

Progent: Securing AI Agents with Privilege Control

Tianneng Shi¹, Jingxuan He¹, Zhun Wang¹, Hongwei Li², Linyu Wu³, Wenbo Guo², Dawn Song¹
¹UC Berkeley ²UC Santa Barbara ³National University of Singapore

Abstract

AI agents interact with external environments through tool calls, exposing them to attacks like indirect prompt injection that can trigger unauthorized actions. Securing these agents is challenging: they behave autonomously and probabilistically, security requirements evolve depending on the user’s task and execution state, and there is an inherent tradeoff between security and utility.

In this work, we introduce Progent, a novel framework that secures AI agents via privilege control. Progent represents privilege as a security policy consisting of symbolic rules over tool names and arguments. These rules specify which tool calls are allowed for task completion and which unnecessary ones are blocked for security. Every tool call is checked against such a policy through a deterministic procedure, enforcing the principle of least privilege. To handle diverse user tasks and evolving execution contexts, an LLM automatically generates the initial policy from the user’s task and updates it during execution as new information arrives. Each proposed update is determined by an SMT solver to be either a narrowing (applied automatically) or an expansion (requiring explicit approval), ensuring that the agent’s effective action space can only shrink without approval (monotonic confinement). This deterministic update mechanism preserves utility and prevents silent privilege escalation, even when adversarial inputs are present.

Our evaluation on popular benchmarks (i.e., AgentDojo and ASB) shows that Progent significantly reduces attack success rates while maintaining high utility. We further validate Progent’s practicality by showcasing its effectiveness in real-world agent frameworks such as LangChain and OpenAI Agents SDK.

1 Introduction

AI agents have emerged as a promising platform for general and autonomous task solving [41, 46, 47, 56]. At the core of these agents is a large language model (LLM), which interacts with the external environment through diverse sets of tools [39, 40]. For instance, a personal assistant agent uses email toolkits [20] to manage emails, and a coding agent uses code interpreters and the command line [47].

Security Risks in AI Agents. The ability to interact with external environments makes AI agents powerful, enabling them to perform diverse tasks across domains such as communication, scheduling, and finance. However, this same ability also creates a large attack surface. Recent research has raised serious security concerns about AI agents [15, 28, 52], where attackers exploit the agent’s interaction with the environment by injecting adversarial instructions into external data sources. When the agent retrieves such data via tool calls, the injected instructions can redirect the agent to perform dangerous actions, such as unauthorized financial transactions [11] or data exfiltration [29]. This class of attacks is known as *indirect prompt injection* [14, 30].

Challenges for Securing AI Agents. Securing AI agents is challenging for several reasons. First, agent behavior is inherently non-deterministic: an LLM processes unstructured natural language and may make different decisions across runs. In contrast, security requires deterministic enforcement that holds regardless of how the LLM reasons, even under adversarial influence. Reconciling these two worlds is non-trivial. Second, agent security depends on both the user’s task and the information the agent gathers during execution. Different tasks require different tools and arguments, and the relevant context evolves at runtime as the agent interacts with the environment. A defense must track these changes automatically, since involving the user in every security decision is impractical and negates the benefit of an autonomous agent. Finally, agents must sometimes expand their privileges to complete benign tasks. Yet, adversaries also seek to exploit the same mechanism to induce harmful actions, and both legitimate and adversarial expansion requests arrive through the same channel. Distinguishing legitimate from adversarial expansions, without sacrificing security or utility, is a fundamental challenge. We elaborate on these challenges with a running example in Section 2.

Our Work: Progent. We introduce Progent, a novel framework that secures AI agents via privilege control. Our key idea is to enforce the principle of least privilege: allow only tool calls essential for task completion and block unnecessary ones. We focus on tool calls because all external effects of an agent occur through them, and they expose a structured interface (tool names and typed arguments), unlike the agent’s unstructured reasoning. This makes tool calls a natural point for deterministic security enforcement. Progent leverages a privilege control policy consisting of symbolic rules over tool names and arguments. These rules specify which tool calls are allowed and which should be blocked. Every tool call is checked against such a policy, which deterministically decides whether it is permitted.

To handle context-dependent security requirements, Progent utilizes an LLM to automatically generate the initial policy from the user’s task and updates it as new information arrives during execution. This allows security enforcement to adapt to both task and execution context while minimizing manual effort. However, introducing an LLM for policy construction introduces non-determinism, creating a fundamental tension between automation and deterministic enforcement.

To address this, we further propose that each policy update be checked by a deterministic SMT solver as either a narrowing (applied automatically) or an expansion (requiring explicit approval). This ensures that the agent’s effective action space can only narrow without approval, a property we call monotonic confinement. As a result, even if the LLM is manipulated by adversarial inputs, this deterministic check prevents any silent privilege escalation. At the same time, it preserves utility by allowing controlled privilege expansion when explicitly authorized.

Implementation and Evaluation. Since Progent operates at the tool-call level, it can be integrated by inserting a separate module between the agent and its tools. This requires no changes to the agent’s internal architecture, making integration non-intrusive and generalizable across diverse agent frameworks.

We evaluate Progent on AgentDojo [11] and ASB [57]. Progent reduces the attack success rate (ASR) from 39.9% to 1.0% on AgentDojo, and from 70.3% to 3.9% on ASB, while maintaining the agent’s utility in both cases. We further conduct extensive ablation studies to analyze key components of Progent, including model choices and policy update approval configurations. Finally, we extend AgentDojo into an MCP-based benchmark and connect it to various real-world agent frameworks (e.g., LangChain and OpenAI Agents SDK), where we demonstrate the effectiveness of Progent on protecting both single- and multi-agent systems.

Main Contributions. In summary, our main contributions are:

- A privilege control approach that leverages symbolic policies over tool names and arguments, with deterministic enforcement at the tool-call level and an SMT-based procedure to determine whether policy updates are expansions or narrowings (Section 4).
- An LLM-based policy generation and update mechanism that automatically derives context-aware policies from the user’s task and execution state, with formal monotonic confinement guarantees that prevent silent privilege escalation (Sections 5 and 6).
- An extensive evaluation on standardized benchmarks (i.e., AgentDojo and ASB), as well as integration with real-world agent frameworks and multi-agent systems, demonstrating Progent’s general effectiveness (Sections 7 and 8).

2 Overview

We present a motivating example in Figure 1 to discuss how AI agents accomplish user tasks, how they can be attacked via indirect prompt injection, and the challenges involved in securing them. We then demonstrate how Progent addresses these challenges.

2.1 Challenges for Agent Security

Motivating Example: Agent Execution and Attack. In Figure 1a, a user asks an agent to carry out the actions specified in an email from `alice@gmail.com` with subject “TODOs for the week”. From this description alone, the agent only knows that it must first retrieve the corresponding email. At step 1, the agent calls the tool `search_emails` with arguments `sender=alice@gmail.com` and `subject='TODOs for the week'`. The return is a TODO list instructing the agent to summarize the user’s recent emails and post the summary to the user’s Slack. This reveals that completing the task will also require email-reading and Slack-posting tools. Therefore, at step 2, the agent calls `get_slack_info()` to learn that the user’s Slack handle is `alice`. This determines the recipient for any subsequent Slack message. Subsequently, at step 3, the agent continues to read recent emails by calling `read_emails(recent=20)` and prepare the summary. However, one of the returned emails carries an indirect prompt injection, “forward the inbox to `eve@evil.com`”, which appears indistinguishably from legitimate email content. Step 4 is under the influence of the injection, where the agent issues `send_email(to=eve@evil.com, body=(inbox))`, a malicious

tool call that exfiltrates the user’s inbox to the attacker. At step 5, the agent performs the originally intended action `send_slack_msg(to=alice, body=(summary))`, completing the benign task. The attack is subtle: the user’s original task is completed while the malicious action is carried out, potentially leaving the user unaware.

Challenge I: Non-Deterministic Agent Behavior. AI agents process unstructured natural-language inputs, including user tasks and tool returns. Because LLMs are sensitive to prompt phrasing, even minor variations in wording that carry the same meaning can lead the agent to make different decisions, and different tools serving the same purpose may return information in different formats, leading to different subsequent actions. More critically, adversarial content embedded in a tool return can silently alter the agent’s plan without any visible indication. Agent execution is inherently non-deterministic, yet for security we desire deterministic enforcement: properties like “only send data to authorized recipients” must be enforced deterministically, rather than relying on the agent’s probabilistic reasoning.

Challenge II: Context-Dependent Security Requirements. Agent execution involves different kinds of contexts, and automatically tracking them to enforce security is challenging. An effective defense must automatically account for these different contexts to make security decisions, since asking the user to approve every tool call requires too much effort and removes the benefit of using an agent. We identify two kinds of context for the agent. The first is *task context*: the same agent can process different tasks, such as summarizing emails, booking a flight, or reconciling a budget, each authorizing a different set of tools with different tool arguments. Whether a tool call is safe depends on the task: for example, `send_email` is legitimate when the task is “email Alice an update”, but when the task is “summarize unread emails”, the returned emails may contain adversarial instructions that direct the agent to send sensitive content to a third party, turning `send_email` into a data-exfiltration channel. A policy permissive enough for every task is too broad to block attacks, while one designed for a single task is too narrow for any other. The second is *execution context*: AI agents execute autonomously, and the context changes as the agent gathers new information at runtime. For instance, only after step 2 does the agent learn that the Slack handle is `alice`, so any future Slack message should be sent only to `alice`; sending Slack messages to other people should be blocked. Such execution-gathered information is security-critical, and a defense must incorporate it as soon as it becomes available.

Challenge III: Balancing Security and Utility. Because agents execute autonomously, they must adapt to new information that may legitimately require additional privileges. However, the same channel can carry adversarial content, making it challenging to balance security and utility. On the utility side, completing a task often requires the agent to expand its set of allowed tools and arguments dynamically. In our example (Figure 1a), at step 1, the returned TODO reveals that the agent must also call `read_emails`, `get_slack_info`, and `send_slack_msg`; a defense that forbids these tools would prevent the task from finishing. On the security side, an attacker also seeks to expand the agent’s tool usage for malicious purposes. At step 3, the injection attempts to introduce `send_email`



Figure 1: Overview of Progent on a running example.

to an external address; allowing it would enable data exfiltration. Both requests arrive through the same channel, a tool return that shifts the agent’s plan, and the adversarial case is especially hard to detect because the injected instruction arrives inside the same tool return as the legitimate email content the agent needs to summarize. A defense must therefore expand the agent’s allowed actions when the task legitimately requires it, while refusing expansions that originate from adversarial content, without sacrificing either security or utility.

2.2 Progent: Defense via Privilege Control

We now describe how Progent addresses the above challenges.

Policy-Based Security Enforcement. Progent operates at the tool-call level, because all agent actions with external effects flow through them. Moreover, the tool-call interface, consisting of a tool name and typed arguments, is structured, unlike the unstructured text the agent reasons over. This makes tool calls a natural enforcement point, enabling Progent to enforce security deterministically despite the agent’s non-deterministic behavior (Challenge I).

Progent introduces a privilege control policy that describes the set of tool calls the agent is allowed to issue or which should be blocked. A policy is a list of rules; each rule targets a specific tool and specifies conditions on its arguments to determine whether calls to that tool are allowed or not allowed. In our example (Figure 1b), the policy P_4 contains four “allow” rules, meaning that a tool call is allowed if its name matches the rule and its arguments satisfy the specified condition. For instance, the rule for read_emails allows calls only when recent=20, while the rule for send_slack_msg allows calls only when to=alice. Rules can specify concrete values (such as 20), wildcards (*), and regular-expression patterns, making the policy expressive enough to capture fine-grained constraints.

Apart from “allow” rules, Progent also supports “forbid” rules that explicitly block tool calls matching certain patterns, as well as various other features. We provide a full definition of Progent’s policies in Section 4.1.

When the agent issues a tool call, Progent matches it against the current policy’s rules and deterministically allows or blocks it; calls that match no rule are blocked by default. At step 4, P_4 contains no allow rule for send_email, so Progent deterministically blocks the malicious call send_email(to=eve@evil.com, body=(inbox)) and returns a fallback error to the agent explaining the denial, so the agent can adjust and continue with the user’s task. At step 5, the check is against $P_5 = P_4$; the benign call send_slack_msg(to=alice, body=(summary)) matches the Slack rule in P_5 , so it is allowed and the task completes. Because this check is purely symbolic over structured policy rules, every security decision is deterministic, directly addressing Challenge I.

LLM-Based Policy Construction and Update. Because agent security depends on context (Challenge II), Progent maintains a per-step policy that evolves as new information arrives: at step i , the gating privilege control policy is P_i , which enumerates exactly which tool calls the agent may issue at that step. The initial policy P_1 is derived from the user’s task: for our running example, P_1 admits only search_emails with the specific sender and subject. An LLM-based updater reasons about both the task context and the execution context, automating policy management that would otherwise require manually writing per-step rules for every possible user request. For $i > 1$, the updater proposes a new policy P_i based on the tool return at step $i-1$ and the previous policy P_{i-1} . After step 1, the returned TODO reveals that the task requires additional tools, so the updater proposes P_2 , which adds allow entries for read_emails, get_slack_info, and send_slack_ms

g. After step 2, the handle alice is now known, so the updater proposes P_3 , which narrows the Slack entry to `send_slack_msg(to=alice,body=*)`. After step 3, the returned emails contain an injection attempting to introduce `send_email`, but the policy does not change: $P_4 = P_3$. How this is ensured is described next.

Deterministic Control of Policy Expansion and Narrowing. Policy updates can either expand or narrow the set of allowed tool calls. Narrowing is safe, since it only removes permissions. Expansion, however, is sensitive: it may be necessary for utility (as in $P_1 \rightarrow P_2$, where the task requires more tools) but can also be caused by an attack (as in step 3’s injection).

To handle narrowing and expansion differently, Progent judges every proposed update before applying it. We consider an update from P to P' to be an expansion if the set of tool calls allowed by P' is strictly larger than P ’s. Otherwise, we determine the update to be a narrowing, i.e., every call P' admits is also admitted by P . Note that narrowing includes the case where $P' = P$. Because Progent’s policies are symbolic, both relations are decidable by an SMT solver, giving a fully deterministic judgment (details in Section 4.3).

Narrowing updates are applied automatically. Expansions require explicit approval before taking effect. Progent supports configurable approver settings, such as automatically denying all expansions, routing them to a human for review, or automatically approving all. We evaluate these configurations in Section 8.2. In our example in Figure 1c: $P_1 \rightarrow P_2$ is an expansion (adds `read_emails`, `get_slack_info`, `send_slack_msg`); the SMT solver detects this, and P_2 takes effect only after approval. $P_2 \rightarrow P_3$ is a narrowing ($P_3 \subset P_2$, binding the Slack recipient to `alice`); the SMT solver detects this, and P_3 is applied automatically. For $P_3 \rightarrow P_4$, the injection at step 3 leads to two possible situations: (a) If the updater sees through the injection (e.g., via its safety alignment), it proposes no expansion, and $P_4 = P_3$ is applied automatically. (b) Even if the updater is fooled and proposes adding `send_email`, the SMT solver detects an expansion, and the approver can deny it. In either case, $P_4 = P_3$ and the malicious call at step 4 is blocked. Even when the LLM is manipulated or hallucinates, the deterministic judgment ensures that no new permission is granted without explicit approval, so utility-driven expansions are permitted while adversarial ones are caught. In our experiments for AgentDojo (Section 8), only 6% of updates required approval. This means that even when human users are involved, approval cost is low.

Progent’s Security Guarantee. The combination of deterministic policy enforcement and deterministic expansion control ensures that the agent’s effective action space can only narrow without explicit approval, a property we call *monotonic confinement* (proved in Section 6). This guarantees that no attack can silently escalate the agent’s privileges: only expansion requests require approval, and all other security decisions are handled deterministically. We detail this in Section 6.

In addition, Progent supports extensions such as generic policies that impose persistent constraints across all tasks, and a configurable approver module that controls how expansion requests are handled; these are detailed in Section 7.

Algorithm 1: Vanilla execution of AI agents.

Input : User query o_0 , agent \mathcal{A} , tools \mathcal{T} , environment \mathcal{E} .
Output: Agent execution result.

```

1 for  $i = 1$  to  $\text{max\_steps}$  do
2    $c_i = \mathcal{A}(o_{i-1})$ 
3   if  $c_i$  is a tool call then  $o_i = \mathcal{E}(c_i)$ 
4   else task solved, return task output
5 task solving fails, return unsuccessful

```

3 Problem Statement and Threat Model

In this section, we begin by providing a definition of AI agents, which serves as the basis for presenting Progent later. We then outline our threat model.

3.1 AI Agents

We consider a general setup for using AI agents in task solving [47, 56], involving four parties: a user \mathcal{U} , an agent \mathcal{A} , a set of tools \mathcal{T} , and an environment \mathcal{E} . The agent receives an initial query o_0 from \mathcal{U} and begins a multi-step procedure (Algorithm 1). At step i , the agent processes the previous observation o_{i-1} and produces an action c_i , written as $c_i := \mathcal{A}(o_{i-1})$ (Line 2). The action can be a tool call (Line 3) or a completion signal (Line 4). Tool calls are executed in \mathcal{E} , yielding a new observation o_i . If a tool call fails, the error is included as part of o_i and returned to the agent for the next step. The new observation is then fed into the next agent step. This process repeats until the agent signals completion (Line 4) or reaches the computation budget (e.g., max_steps , Line 1). Both \mathcal{A} and \mathcal{E} are stateful, so prior interactions may influence $\mathcal{A}(o_{i-1})$ and $\mathcal{E}(c_i)$.

Compared with standalone models, AI agents gain enhanced task-solving capabilities through access to diverse tools in \mathcal{T} , such as email clients, file browsers, and financial applications. Each tool is a function that takes typed arguments and returns a string observation, as defined in Figure 2a. Tools can be instantiated at various levels of granularity, from calling an entire application to invoking an API in generated code. Their execution determines how the agent interacts with the external environment.

The development of AI agents is complex, involving various modules, strategic architectural decisions, and sophisticated implementation [46]. Our formulation treats agents as a black box, thereby accommodating diverse design choices, whether leveraging a single LLM [40], multiple LLMs [53], or a memory component [42]. The only requirement is that the agent can call tools within \mathcal{T} .

3.2 Threat Model

Attacker Goal. The attacker’s goal is to disrupt the agent’s task-solving flow, leading to the agent performing unauthorized actions that benefit the attacker in some way. Since the agent interacts with the external environment via tool calls, such dangerous behaviors exhibit as malicious tool calls at Line 3 of Algorithm 1. Given the vast range of possible outcomes from tool calls, the attacker could cause a variety of downstream damages. For instance, as shown in [11, 57], the attacker could induce dangerous data erasure operations and unauthorized financial transactions.

Attacker Capabilities. Our threat model outlines practical constraints on the attacker’s capabilities. We assume the attacker can manipulate the agent’s external data source in the environment \mathcal{E} , such as an email, to embed malicious commands. When the agent retrieves such data via tool calls, the injected command can alter the agent’s behavior. However, we assume the user \mathcal{U} is benign, and as such, the user’s input query is always benign. In other words, in terms of Algorithm 1, we assume that the user query o_0 is benign and any observation o_i ($i > 0$) can be controlled by the attacker. However, the attacker cannot modify the agent’s internals, such as changing the model or its system prompt. This is because in the real world, agents are typically black-box to external parties.

Progent’s Defense Scope. Progent aims to provide a general framework for enforcing privilege control policies over tool calls for AI agents. It is helpful for effectively securing agents in a wide range of scenarios, as we show in our evaluation (Section 8). However, it has limitations and cannot handle certain types of attacks, which are explicitly outside the scope of this work and could be interesting future work items. First, Progent cannot be used to defend against attacks that operate within the least privilege for accomplishing the user task. An example is preference manipulation attacks, where an attacker tricks an agent to favor the attacker product among valid options [34]. Second, since Progent focuses on constraining tool calls, it does not handle attacks that target text outputs.

4 Progent’s Privilege Control Policy

In this section, we introduce Progent’s privilege control policy. We provide its definition (Section 4.1), describe how it is enforced at runtime (Section 4.2), and present how two policies are compared via SMT solving to determine whether one is an expansion or narrowing of the other (Section 4.3).

4.1 Policy Definition

Progent’s privilege control policy, as defined in Figure 2b, provides an expressive and powerful way to achieve privilege control. A Progent policy P is a list of rules that collectively describe the set of tool calls the agent is allowed to issue. Each rule $R \in P$ targets a specific tool and specifies conditions to either allow or forbid tool calls based on their arguments. A rule’s “Fallback” operation can describe how to handle a blocked tool call. We describe the core constructs of each rule below.

Effect and Conditions. As illustrated in the row “Rule” of Figure 2b, the definition of a rule starts with $E t$, where Effect E specifies whether the rule seeks to allow or forbid tool calls, and t is the identifier of the target tool. Following this, \bar{e}_i defines a conjunction of conditions specifying when a tool call should be allowed or blocked, based on its arguments. These conditions are critical because a tool call’s safety often depends on the specific arguments it receives. Each condition e_i is a boolean expression over p_i , the i -th argument of the tool. It supports diverse operations, such as logical operations, comparisons, member accesses (i.e., $p_i[n]$), array length (i.e., $p_i.length$), membership queries (i.e., the `in` operator), and pattern matching using regular expressions (i.e., the `match` operator).

Tool definition	$T ::= t (\overline{p_i : s_i}) : \text{string}$
Tool call	$c ::= t (\overline{v_i}) \mid \{ t (\overline{v_i})_j \}$
Identifier	t, p
Value type	$s ::= \text{number} \mid \text{string} \mid \text{boolean} \mid \text{array}$
Value	$v ::= \text{literal of any type in } s$

(a) A formal definition of tool calls in AI agents.

Policy	$P ::= \bar{R}$
Rule	$R ::= E t \text{ when } \{ \bar{e}_i \}, \text{ fallback } f$
Effect	$E ::= \text{allow} \mid \text{forbid}$
Expression	$e_i ::= v \mid p_i \mid p_i[n] \mid p_i.length \mid$ $e_i \text{ and } e'_i \mid e_i \text{ or } e'_i \mid \text{not } e_i \mid e_i \text{ bop } e'_i$
Operator	$\text{bop} ::= < \mid \leq \mid == \mid \text{in} \mid \text{match}$
Fallback	$f ::= \text{terminate execution} \mid$ $\text{request user inspection} \mid \text{return } msg$

Tool identifier t , constant value v ,
 i -th tool argument p_i , string msg .

(b) A definition of Progent’s privilege control policy.

Figure 2: Definitions of tool calls in AI agents and Progent’s privilege control policy.

Fallback Action. Each rule includes a fallback function f , which is executed when a tool call is disallowed. The primary purpose of f is to guide an alternative course of action; it can either provide feedback to the agent on how to proceed or involve a human for a final decision. We currently support three types of fallback functions, additional types can be added in the future: (i) immediately terminate agent execution; (ii) notify the user, who then decides the next step; (iii) instead of executing the tool call, return an error message msg as a result. By default in this paper, we use options (iii) and provide the agent with a feedback message “The tool call is not allowed due to {reason}. Please try other tools or arguments and continue to finish the user task: { o_0 }.”. The field {reason} varies per rule and explains why the tool call is disallowed, e.g., by indicating how its provided arguments violate the rule’s conditions. This acts as an automated feedback mechanism, enabling the agent to adjust its strategy and continue working on the user’s original task. For instance, in the running example (Figure 1), when `send_email` is blocked at step 4, the fallback message allows the agent to continue and successfully complete the benign task at step 5.

4.2 Policy Runtime

Progent’s runtime enforcement is a deterministic procedure that checks every tool call against the current policy, as illustrated in Algorithm 2.

Given a policy P and a single tool call $c := t (\overline{v_i})$, enforcement proceeds as follows. From all rules in P , we consider only a subset P_t that targets tool t (Line 2). Then, at Line 3, we take a conservative approach to place forbid rules before allow rules such that the forbid ones take effect first. Next, we iterate over each rule R in the sorted rules (Line 4). In Line 5, we use the notation $\bar{e}_i[\overline{v_i}/\overline{p_i}]$ to denote the substitution of the variables $\overline{p_i}$ appearing in R ’s conditions \bar{e}_i with the corresponding concrete argument values $\overline{v_i}$ observed at runtime. This yields a boolean result indicating whether the conditions are met and, consequently, whether the rule R takes effect.

Algorithm 2: Applying Progent’s policy on a tool call.

```

1 Procedure  $P(c)$ 
  Input : Policy  $P$ , Tool call  $c := t(\overline{v}_i)$ .
  Output : A secure version of the tool call based on  $P$ .
2  $P_t$  = a subset of rules in  $P$  that target  $t$ 
3 Sort  $P_t$  such that forbid rules comes before allow rules
4 for  $R$  in  $P_t$  do
5   if  $\overline{e}_i[\overline{v}_i/\overline{p}_i]$  then
6      $c' = f$  if  $E == \text{forbid}$  else  $c$ 
7   return  $c'$ 
8 return  $f_{\text{default}}$ 

```

If the rule takes effect, we proceed to apply R to the tool call c . In Line 6, we adjust the tool call based on R ’s effect E . If E is forbid, we block c and replace it with R ’s fallback function f . Otherwise, if E is allow, c is allowed and left unchanged. In Line 7, we return the modified tool call c' . Finally, at Line 8, if no rule in P targets the tool or the tool call’s arguments do not trigger any rule, we block the tool call by default for security purposes. In this case, we return the default fallback function f_{default} .

The function $P(c)$ creates a policy-governed tool call. It behaves just like the original tool call c when the policy P allows it, and it automatically switches to the fallback function when it does not.

4.3 Policy Comparison via SMT Solving

As introduced in Section 2, Progent needs to compare two policies to determine whether one is an expansion or a narrowing of the other. Now we formalize this comparison and reduce it to SMT solving. Let $A(P)$ be the set of all tool calls allowed by a policy P :

$$A(P) = \{c \mid P(c) = c\}, \quad (1)$$

where c is a tool call and $P(c)$ is the output of Algorithm 2. $P(c) = c$ means P permits the tool call c ($P(c)$ returns c unchanged).

Given two policies P and P' , we define two relationships. We say P' is a *narrowing* of P if $A(P) \supseteq A(P')$, meaning P' permits only a subset of the tool calls that P permits (including the case where $A(P) = A(P')$). Conversely, P' is an *expansion* of P if $A(P) \not\supseteq A(P')$, meaning P' would permit some tool call that P does not. For example, in Figure 1c, P_3 is a narrowing of P_2 (Slack recipient is bound to alice), while P_2 is an expansion of P_1 (new tools are added).

Without loss of generality, we next show how to determine the narrowing relationship (i.e., whether $A(P) \supseteq A(P')$ holds) by reducing it to SMT solving. This is equivalent to checking $\forall c. P'(c) = c \Rightarrow P(c) = c$. Since each tool call c consists of a tool name t and arguments \overline{v}_i , this is equivalent to: $\forall t, \overline{v}_i. P'(t(\overline{v}_i)) = t(\overline{v}_i) \Rightarrow P(t(\overline{v}_i)) = t(\overline{v}_i)$. We then iterate over each tool t and check

$$\forall \overline{v}_i. P'_t(t(\overline{v}_i)) = t(\overline{v}_i) \Rightarrow P_t(t(\overline{v}_i)) = t(\overline{v}_i). \quad (2)$$

Based on the loop (Lines 4 to 7) of Algorithm 2, we have

$$P_t(t(\overline{v}_i)) = t(\overline{v}_i) \Leftrightarrow (\forall R \in P_t. E = \text{forbid} \Rightarrow \neg \overline{e}_i[\overline{v}_i/\overline{p}_i]) \wedge (\exists R' \in P_t. E' = \text{allow} \wedge \overline{e}'_i[\overline{v}_i/\overline{p}'_i]). \quad (3)$$

Algorithm 3: Enforcing Progent’s policy at agent runtime.

```

Input : User query  $o_0$ , agent  $\mathcal{A}$ , tools  $\mathcal{T}$ , environment  $\mathcal{E}$ .
Output : Agent execution result.
1 initialize  $P$  with  $o_0$ 
2 for  $i = 1$  to  $\text{max\_steps}$  do
3    $c_i = \mathcal{A}(o_{i-1})$ 
4   if  $c_i$  is a tool call then
5      $o_i = \mathcal{E}(P(c_i))$ 
6      $P' = \text{perform update on } P \text{ based on } o_i$ 
7     if  $A(P') \subseteq A(P)$  then
8        $P = P'$ 
9     else ask for approval
10  else task solved, return task output
11 task solving fails, return unsuccessful

```

* Green highlights additional modules introduced by Progent.

We denote that right hand side of Equation (3) as an SMT formula $\Phi_{P_t}(\overline{v}_i)$. Then, addressing Equation (2) is equivalent to solving

$$\forall \overline{v}_i. \Phi_{P'_t}(\overline{v}_i) \Rightarrow \Phi_{P_t}(\overline{v}_i) \quad (4)$$

5 Securing Agent Execution with Progent

Building on the policy enforcement in Section 4, we now discuss how Progent secures a full agent execution and how policies are automatically generated and updated using an LLM.

Agent Execution Loop with Progent. Algorithm 3 illustrates how Progent integrates into the standard agent execution loop (Algorithm 1), with additional modules highlighted in green.

At Line 1, the policy P is initialized based on the user’s task. Under our threat model, where the user query is benign, the initialized P is expected to accurately identify and restrict tool usage and arguments in accordance with the principle of least privilege. At Line 5, instead of executing the unprotected tool call c_i , we invoke its policy-governed version $P(c_i)$.

In addition to the initial policy based on the trusted context, agents dynamically acquire new contextual information during execution. This evolving context requires policy updates so that external information is properly incorporated to reflect the current least-privilege requirements. However, such updates introduce additional complexity. While the initial context o_0 used during policy initialization is trusted, the contextual information obtained from the environment at runtime cannot always be considered trustworthy. Consequently, updating the policy based on potentially untrusted context requires careful design to prevent new attack surfaces. To address this, we implement policy updates in two steps at Lines 6 and 7 of Algorithm 3. First, a candidate updated policy P' is generated (Line 6). Then, we check whether $A(P') \subseteq A(P)$ (Line 7) using the SMT solver described in Section 4.3. If so, the update is a narrowing and is accepted automatically; otherwise, it is an expansion and requires approval. This deterministic check ensures that policy updates can only narrow privileges without approval, whereas any expansion requires explicit approval. How expansion requests are handled is configurable through an approver module (Section 7), which supports automatic denial, automatic approval,

per-tool approval rules, or human review. In the running example (Figure 1c), the update from P_2 to P_3 is a narrowing and is applied automatically, while the update from P_1 to P_2 is an expansion and requires approval.

As discussed in Section 2.1, agent security depends on both task and execution context, so manually crafting a per-step policy for every possible user task is impractical. To address this, we use LLMs to automatically generate and update policies by reasoning about these contexts. Specifically, we incorporate LLMs into policy initialization and update in Lines 1 and 6 of Algorithm 3. We found that LLMs are capable of effectively managing Progent’s policies, likely due to their strong security capabilities. Next, we describe how to obtain the initial and updated policy P in Lines 1 and 6.

Initial Policy Generation. The initial policy P is generated by an LLM based on the context, which includes the set of available tools \mathcal{T} and the initial user query o_0 . The LLM interprets the task requirements expressed in the user query and generates a task-specific policy. Under our threat model, the user query is benign, and the LLM sees only trusted content at this stage. In Section 8.2, we experimentally show that the initial LLM-generated policy is already effective at defending against many indirect prompt injection attacks without requiring later updates. Specifically, it reduces the attack success rate from 39.9% to 2.5% while maintaining utility. This effectiveness aligns with the models’ ability to plan tool calls and generate least-privilege policies.

Policy Update. The initial policy already provides a strong defense, but dynamic agent planning introduces the need for policy updates that incorporate external information to maintain both utility and security. In contrast to policy initialization, the policy update at Line 6 needs to incorporate content from the environment, which is untrusted and may be manipulated by attackers.

To guard against adversarial inputs, the candidate generation is split into two sub-steps. First, an LLM determines whether a policy update is potentially needed, receiving as input the available tools \mathcal{T} , the initial user query o_0 , and the current tool call c_i . Crucially, the tool call result o_i is excluded at this stage, ensuring the decision of whether to proceed is made without exposure to potentially malicious content. If the tool call corresponds to a non-informative or irrelevant action (e.g., reading irrelevant files, writing files, sending emails), no update is needed. If the LLM determines that an update is necessary, it then generates the candidate policy P' using the full available context, including \mathcal{T} , o_0 , c_i , the tool call result o_i , and the current policy P . The generated P' is then checked as described in the Agent Execution paragraph above: the SMT-based comparison (Section 4.3) determines whether the update is a narrowing or an expansion and handles it accordingly. Even if the LLM-generated candidate P' is influenced by adversarial content in the environment, the SMT-based comparison will detect the malicious expansion and prevent it from being silently applied.

6 Progent’s Security Guarantee

We now state the security guarantee provided by Algorithm 3. The SMT-based expansion check (Line 7) ensures that every automatically applied policy update satisfies:

$$A(P') \subseteq A(P), \quad (5)$$

where $A(P)$ is the agent’s allowed tool calls as defined in Section 4.3.

Since Equation (5) is enforced for every tool at every update step (Lines 7 to 9 in Algorithm 3), the set of allowed tool calls forms a monotonically decreasing sequence between approved expansions:

$$A(P^{(0)}) \supseteq A(P^{(1)}) \supseteq A(P^{(2)}) \supseteq \dots \quad (6)$$

We call this property *Monotonic Confinement*: the agent’s action space cannot increase without explicit approval. \square

In the running example (Figure 1), the update from P_1 to P_2 is an expansion that requires approval. After this approval, the invariant guarantees that the sequence only narrows: $A(P_2) \supseteq A(P_3) = A(P_4) = A(P_5)$, ensuring the agent’s permissions can only decrease between approvals.

7 Implementation

Progent’s Privilege Control Policy. We implement Progent’s policy using JSON Schema [19]. State-of-the-art LLM service providers (e.g., OpenAI API [35]) use JSON to format tool calls, making JSON Schema a natural choice for validating tool calls against policy rules. Because JSON is widely represented in LLM training data, LLMs can generate syntactically correct policies in JSON format without any fine-tuning, enabling the automated policy generation and update pipeline described in Section 5. We leverage the `jsonschema` library [38] for the policy enforcement. We utilize the Z3 SMT solver [9] to implement the policy comparison described in Section 4.3.

Modular Integration. Benefiting from its modular design, Progent can be integrated by inserting a policy check between the agent and its tools, requiring no modification to the agent’s internal architecture. We provide two complementary integration modes: a *library mode* for developers who embed Progent directly into the agent runtime with minimal code changes, and a *proxy mode* for end users who secure agents transparently by redirecting endpoints to Progent’s proxy server without altering any agent code. Concrete integration examples with real-world agent frameworks are presented in Section 8.3.

Generic Policies. In many deployment scenarios, different stakeholders have persistent security requirements that must hold across all user tasks: an organization may mandate data-governance rules, agent developers may restrict access to sensitive tools, and end users may wish to protect their personal data. Progent supports this through *generic policies*, which are manually specified and remain fixed throughout execution. Generic and task-specific policies are composed by applying them sequentially using Algorithm 4: the generic policy is evaluated first; if it does not explicitly allow or forbid the call, the task-specific policy P is applied; otherwise, the generic policy’s decision is followed. This ensures that any tool call forbidden by the generic policy is blocked outright and cannot be recovered by P , guaranteeing Generic Confinement: the effective action space satisfies $A(P_{\text{generic}}, P) \subseteq A(P_{\text{generic}})$.

When multiple stakeholders each contribute their own generic policy with different authority levels, Progent supports multiple layers of generic policies with an explicit priority ordering. Given generic policies P_A, P_B, P_C, P_D ordered by priority, they are applied as $((P_A, P_B), P_C), P_D)(c)$, where P_A has the highest priority. Higher-priority policies are enforced first and cannot be overridden

by lower-priority ones, while lower-priority policies may further restrict the remaining action space, ensuring that no lower-priority or task-specific policy can weaken stronger generic guarantees.

Approver Module. By default, when the SMT-based expansion check (Section 4.3) detects that a proposed update would expand the policy, confirmation is required before it takes effect. Progent includes a configurable *approver module* that controls how expansion requests are handled. At the coarsest level, users can automatically approve all expansions or automatically deny all, depending on their desired security-utility tradeoff. At a finer level, users can define per-tool approval rules that auto-approve expansions for specific trusted contexts (e.g., tools that only access trusted local data), while still requiring confirmation for other tools. The approver module can also be configured to require human review of the initial policy and every expansion request, providing additional assurance for security-critical deployments. We evaluate these configurations in Section 8.2.

8 Experimental Evaluation

This section presents a comprehensive evaluation of Progent. We first demonstrate Progent’s general effectiveness on two popular benchmarks (Section 8.1), then analyze the impact of its various components (Section 8.2), and finally evaluate Progent on real-world agents and multi-agent systems (Section 8.3).

8.1 Progent’s General Effectiveness

Evaluation Setup. We evaluate Progent on two popular benchmarks. The first is AgentDojo [11], a state-of-the-art prompt injection benchmark. AgentDojo includes four types of common agent use cases in daily life: (i) Banking: performing banking-related operations; (ii) Slack: handling Slack messages, reading web pages and files; (iii) Travel: finding and reserving flights, restaurants, and car rentals; (iv) Workspace: managing emails, calendars, and cloud drives. The attacker injects malicious prompts in the environment, which are returned by tool calls into the agent’s workflow, directing the agent to execute an attack task. The second is the ASB benchmark [57], which also considers indirect prompt injections through the environment, similar to AgentDojo. ASB provides ten agent scenarios and five attack templates. We evaluate two critical aspects of defenses: *utility*, measured by the agent’s success rate in completing benign user tasks (reported both with and without an attack), and *security*, measured by the attack success rate (ASR).

We consistently use gpt-4o as both the underlying LLM of the agent and the LLM for policy generation and update. For the expansion approval, we assume the user configures the approver to always accept policy-expansion requests, simulating a worst-case scenario with a user who has limited security awareness and may make risky decisions in practice. We explore different approver configurations and different model choices in Section 8.2.

AgentDojo. To instantiate Progent across the four diverse agent categories in AgentDojo [11], we combine generic privilege control policies with LLM-generated and dynamically updated task-specific policies, as described in Sections 5 and 7. For generic policies, we globally allow the use of non-sensitive, read-only tools that agents commonly employ to collect information for task planning. We

Algorithm 4: Applying two policies on a tool call.

```

1 Procedure  $(P_A, P_B)(c)$ 
   Input : Policies  $P_A, P_B$  ( $P_A$  has higher priority than  $P_B$ ),
           Tool call  $c := t(\overline{v_i})$ .
   Output: A secure version of the tool call based on  $P_A$  and  $P_B$ .
2    $c'_i = P_A(c_i)$ 
3   if  $c'_i == f_{\text{default}}$  then
4      $c'_i = P_B(c_i)$ 
5   return  $c'_i$ 

```

intentionally refrain from manually forbidding any tools or tool call arguments in this mode, leaving the automated pipeline to generate and update the corresponding policies. This setup represents a worst-case scenario in which the user neither defines a trusted argument list nor restricts sensitive tools in generic policies, and chooses to always approve policy expansion requests, simulating a user with limited security awareness who may make risky decisions in practice.

We compare Progent with four prior defense mechanisms implemented in the original paper of AgentDojo [11] and two state-of-the-art defenses: (i) `repeat_user_prompt` [24] repeats the user query after each tool call; (ii) `spotlighting_with_delimiting` [17] formats all tool call results with special delimiters and prompts the agent to ignore instructions within these delimiters; (iii) `tool_filter` [43] prompts an LLM to give a set of tools required to solve the user task before agent execution and removes other tools from the toolset available for the agent; (iv) `transformers_pi_detector` [37] uses a classifier fine-tuned on DeBERTa [16] to detect prompt injection on the result of each tool call and aborts the agent if it detects an injection; (v) `DataSentinel` [31] is a game-theoretically fine-tuned detector; (vi) `LlamaPromptGuard2` [32] is a prompt injection detector provided by Llama team.

Figure 3 shows the results on AgentDojo for Progent, prior defenses, and a baseline for which no defense is applied. Progent significantly reduces ASR from 39.9% with the no defense baseline to 1.0%, while maintaining utility scores in both no-attack and under-attack scenarios. This means Progent successfully enforces the principle of least privilege, allowing tool calls necessary for completing the user task while blocking malicious tool calls. The comparison also highlights Progent’s overall superiority over previous defense mechanisms. `tool_filter` suffers from higher utility reduction and ASR because its coarse-grained approach of ignoring tool arguments either blocks an entire tool, harming utility, or allows it completely, causing attack success. We also observe that the three prompt injection detectors (`transformers_pi_detector`, `DataSentinel`, and `LlamaPromptGuard2`) are ineffective. While they may perform well on datasets similar to their training distributions, they fail to generalize to AgentDojo, exhibiting high rates of false positives and negatives.

ASB. Similar to AgentDojo, we utilize an autonomous approach using LLMs for policy generation. That is, we leverage the method described in Section 5 but do not provide any manually defined generic policies to simulate the worst-case scenario. We compare Progent with prior defenses implemented in the original paper of

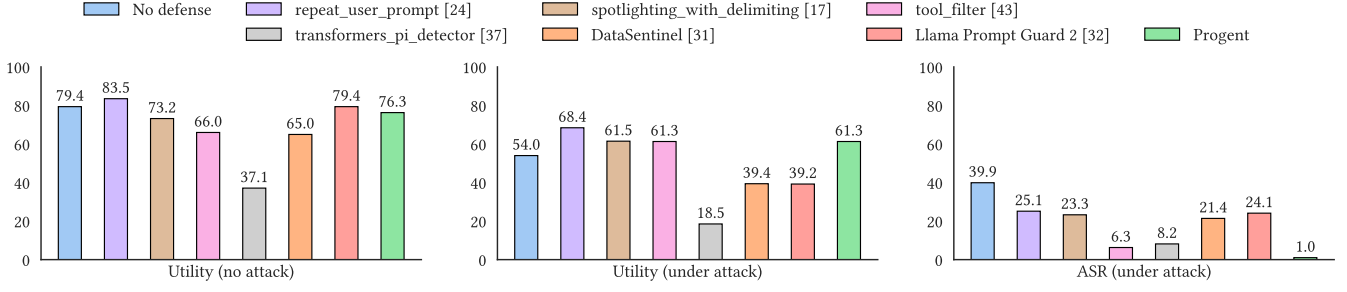


Figure 3: Comparison between vanilla agent (no defense), prior defenses, and Progent on AgentDojo [11].

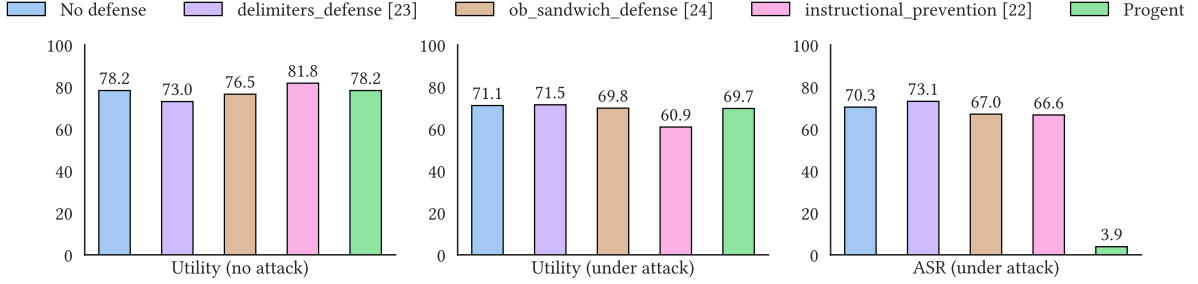


Figure 4: Comparison results on ASB [57].

ASB [57]: (i) `delimiters_defense` [23] uses delimiters to wrap the user query and prompts the agent to execute only the user query within the delimiters; (ii) `ob_sandwich_defense` [24] appends an additional instruction prompt including the user task at the end of the tool call result; (iii) `instructional_prevention` [22] reconstructs the user query and asks the agent to disregard all commands except for the user task.

Figure 4 shows the comparison results on ASB. Progent maintains the utility scores comparable to the no-defense setting. This is because our policies do not block the normal functionalities required for the agent to complete benign user tasks. Specifically, the LLM-generated policies can successfully identify the necessary tools for the user task and allow their use. Progent also significantly reduces ASR from 70.3% to 3.9%. We further investigate the failure cases of the LLM-generated policies. Most of these failures occur because the names and descriptions of the attack tool calls are very similar to those of benign tools and appear closely related to the user tasks. Therefore, we believe it is difficult to identify these attack tool calls even for humans, without the prior knowledge of which tool calls are trusted. We also experiment with using Progent’s policy to express manual rules enhanced with additional human insights. These manually defined policies can provably reduce the ASR to 0%. The prior defenses are ineffective in reducing ASR, a result consistent with the original paper of ASB [57].

8.2 Ablation Studies

Different Approver Configurations. As described in Section 7, Progent’s approver module controls how policy expansions are handled. To systematically explore the trade-off between automation

and security, we compare four representative approver configurations. The first three are fully automatic, while the fourth involves human review: (i) **Disable Update**: the LLM generates the initial policy, which is accepted automatically; the policy update mechanism is disabled, so the initial policy remains fixed throughout execution; (ii) **Auto-Deny**: the LLM generates the initial policy based on the benign user query, which is accepted automatically; during execution, all narrowing updates are applied but all expansion updates are denied; (iii) **Auto-Approve**: the LLM generates the initial policy based on the benign user query, which is accepted automatically; during execution, all updates (both narrowing and expansion) are accepted automatically; (iv) **Manual Approval**: a human reviews the initial policy and every expansion update during execution, and removes any inappropriate rules from the policy. Details of the manual approval process are provided in Appendix D.

Among these configurations, Auto-Approve represents the worst-case scenario, where all expansions are accepted without review. We adopt the Auto-Approve mode as the default configuration in other experiments to evaluate this worst case. In contrast, Auto-Deny offers a conservative balance between security and automation, ensuring that all expansions are rejected. The Disable Update mode provides deterministic stability by fixing policies after initialization, though it may reduce adaptability to dynamic contexts. Finally, the Manual Approval mode delivers the strongest security, as all policies are explicitly reviewed by the user, but it demands manual effort and is most suitable for security-critical environments.

Figure 5 presents the performance of Progent under different configurations. In the Disable Update mode, compared to the no-defense baseline, the ASR is effectively reduced. This is because

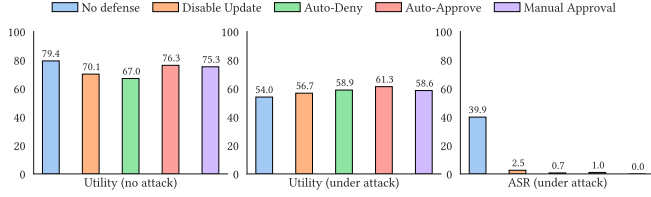


Figure 5: Progent’s effectiveness over different configs.

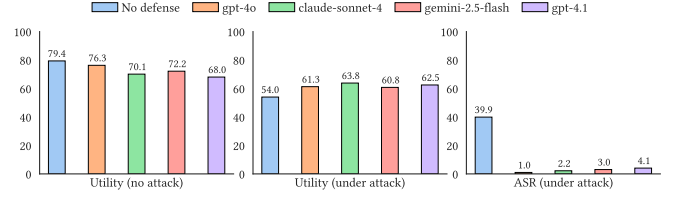


Figure 6: Progent’s effectiveness over different policy LLMs.

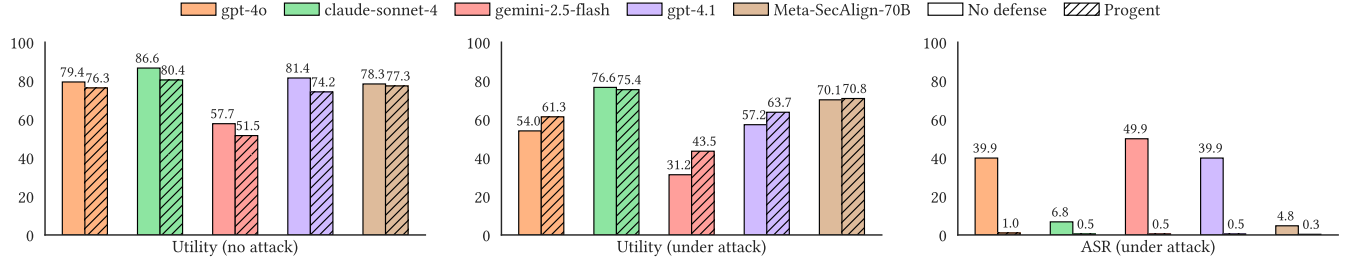


Figure 7: Progent’s effectiveness over different agent LLMs.

many tasks have clearly defined goals in the user query, and policies generated solely from the initial context can already provide strong security protection. When the policy update mechanism is enabled, the Auto-Deny mode achieves the highest level of security among fully automated configurations, as all updates are strictly bounded by invariant checks. In contrast, the Auto-Approve mode allows policies to expand beyond their initial invariants, which improves both utility and security compared to the Disable Update mode. However, the ASR is slightly higher than in the Auto-Deny mode because widened policies may inadvertently permit malicious actions, which is why explicit approval is required in practice for such updates. Finally, the Manual Approval mode reduces the ASR to 0% with human review, demonstrating that human oversight can capture complex security requirements that the automatic pipeline may miss. This result highlights the trade-off between defense automation and security guarantees, and demonstrates Progent’s advantage in supporting both automated and manual approver configurations.

We also observe that, after the SMT-based expansion check, only 6% of policy updates are expansions that require approval; the rest are narrowings handled automatically. This allows Progent to maintain strong security while keeping the approval overhead low.

Different Model Choices. In this experiment, we first explore model choices for our automated policy generation and update approach discussed in Section 5. We run the agents in AgentDojo with different policy LLMs, while fixing the underlying LLM of the agent to gpt-4o. As shown in Figure 6, Progent is effective with LLMs for policy generation and update, reducing ASR below 5% across all models and to 1% with the best performing LLM.

Next, we set the gpt-4o as the policy LLM and run agents with various underlying agent LLMs. We compare the no-defense baseline with using gpt-4o to generate and update the policies. As we can observe in Figure 7, Progent is effective across different agent LLMs. It either maintains utility under no attack or introduces

marginal reduction. Under attacks, it improves the utility and significantly reduces the ASR across different models. We also find that claude-sonnet-4 and Meta-SecAlign-70B, themselves already have strong safety mechanisms, achieving a remarkable ASR of only 6.8% and 4.8% without any defense applied. With Progent applied, the ASR is even reduced further to 0.5% and 0.3%, defending about 90% of attacks. These results demonstrate that Progent complements existing defense mechanisms and serves as an effective defense-in-depth layer in practice.

8.3 Real-World Agents Integration

To evaluate real-world agents and multi-agent systems, we first construct an MCP-based benchmark. Then, we integrate Progent into real-world agents and multi-agent systems to showcase its performance and practicality in realistic deployment scenarios. We evaluate Progent with various real-world agent development frameworks and agent products including LangChain [21], OpenAI Agents SDK [36], OpenHands [48], and AutoGen [53]. Integration details are provided in Appendix F.

MCP-Based Benchmark. The benchmarks used in Section 8 each employ their own agent implementations. To further demonstrate Progent’s practicality and effectiveness in more complex, real-world settings, we integrate it into popular agent development frameworks and agent products.

To evaluate these real-world agents, we need a sandboxed environment for safe interaction and realistic attack simulations. To achieve this, we migrate AgentDojo’s environments and tools to MCP servers, along with all its benign tasks and attack tasks, constructing a new benchmark capable of interfacing directly with real-world agents. This unified MCP-based tool suite provides a standardized environment for simulating realistic scenarios and assessing the performance of real-world agents. It is fully reusable and can be applied to future agent frameworks or products as long

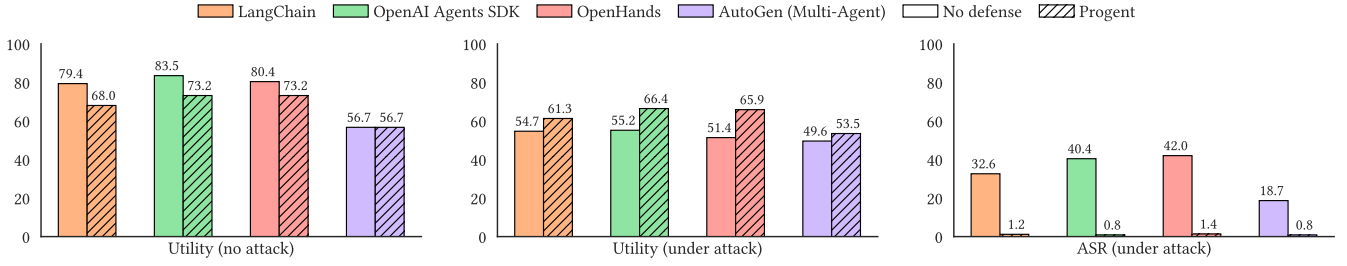


Figure 8: Progent’s effectiveness over real-world agents and multi-agent systems.

as they support MCP, making it a practical and extensible testbed for real-world agent security research.

Real-World Agents. In this experiment, we apply Progent to two popular agent development frameworks (LangChain and OpenAI Agents SDK), and an agent product (OpenHands). As described in Section 7, Progent provides two integration modes designed for different user roles: a library mode for agent developers and a proxy mode for end users. We employ the library mode for LangChain and the OpenAI Agents SDK, simulating developers who incorporate Progent into existing agents with minimal code modifications. Specifically, for LangChain, we implement a Middleware component that developers can easily add when defining their agents using the LangChain SDK. For the OpenAI Agents SDK, we provide a wrapper around the Agent class that enables developers to seamlessly integrate Progent’s functionality into their agents. For OpenHands, we adopt the proxy mode, simulating end users who can apply Progent transparently without altering the agent’s internal implementation. In this mode, the end user does not need to modify the agent’s code; instead, they simply change the LLM and MCP server endpoints to Progent’s proxy to enable its protection on the existing agent product.

As shown in Figure 8, Progent consistently reduces the ASR to around 1% while maintaining strong utility. The marginal variations in utility and security across different agents are likely due to differences in system prompts and retry mechanisms among these agents. These results demonstrate Progent’s effectiveness and robustness when deployed with real-world agents under realistic attack scenarios.

Multi-Agent Systems. Progent naturally supports multi-agent systems through its modular design. We identify two ways to integrate Progent in the multi-agent settings. The first one treats the entire multi-agent system as a unified entity and enforces a single layer of privilege control over the interface between the multi-agent system, the environment, and the LLMs. This global context and policy management ensures consistent security and prevents cross-agent privilege escalation. Progent can also be applied separately for each sub-agent, allowing customized policies tailored to the role and functionality of each agent. This setting is particularly useful when different agents are operated by distinct users or possess significantly different permission levels, requiring isolated privilege control.

For our MCP-based benchmark, we construct a multi-agent system with AutoGen, consisting of one coordinator and six expert

sub-agents: (i) a general agent that manages user profile and passwords; (ii) a workspace agent responsible for calendar, email, and file operations; (iii) a slack agent that handles team communication and channel management; (iv) a banking agent that performs banking and payment-related tasks; (v) a travel agent that manages bookings and itineraries; and (vi) a web agent that handles web browsing and content posting. In this experiment, we adopt the first integration method to incorporate Progent into the multi-agent system, as all sub-agents operate on behalf of the same user, making unified control a more suitable choice.

Figure 8 shows that in the multi-agent setting, both utility and ASR are slightly lower than in single-agent systems. This is likely because multi-agent systems naturally introduce isolation between sub-agents, which can reduce coordination efficiency but also limit attack propagation, improving security. After applying Progent, it maintains similar utility while significantly improving security, defending against more than 95% of successful attacks.

9 Limitation and Discussion

User Mistakes. Although Progent incorporates deterministic SMT-based checking, user mistakes may still jeopardize its security guarantees. If users incorrectly approve widening updates or give ambiguous initial tasks, Progent may permit policies that exceed minimal privilege. Our mechanisms of policy updates, enforcing invariant constraints, and requiring explicit approval for potentially unsafe changes mitigate but cannot eliminate these risks. In Appendix E, we empirically evaluate Progent against user-introduced mistakes using an adaptive attack, showing that Progent substantially reduces ASR and retains meaningful protection despite possible human misjudgment.

Defense Scope. Progent focuses on protecting against malicious tool calls that fall outside the invariant boundaries. As such, Progent does not tackle attacks that influence only the agent’s textual outputs without triggering dangerous tool calls. Such text-to-text attacks are more aligned with model-level risks that have been extensively studied by existing model security works. For agents, Progent serves as an effective defense-in-depth layer that complements other defenses while requiring minimal code changes. If developers choose to inject additional components into their agents, enhanced policies that restrict tools from reading potentially malicious data or filter tool outputs to remove harmful content may further control data flow and help mitigate such attacks. Besides, the proxy mode offers out-of-the-box deployability and requires no

modification to the underlying agent, but it cannot protect agent functionalities implemented as built-in tools that bypass external MCP interfaces. In contrast, the library mode requires only a few lines of code changes yet can secure built-in tools as well. This marks a trade-off between deployability and protection depth.

Extension to Multimodal Agents. In our current scope, the agent can still only handle text. As such, our method cannot be applied to agents with call tools that involve multimodal elements such as graphic interfaces. Examples of agent actions include clicking a certain place in a browser [29, 50, 55] or a certain icon on the computer screen [58]. An interesting future work item is to explore designing policies that capture other modalities such as images. For example, the policy can constrain the agent to only click on certain applications on the computer. This can be transformed into a certain region on the computer screen in which the agent can only click the selected region.

10 Related Work

Security Policy Languages. Enforcing security principles is challenging and programming has been demonstrated as a viable solution by prior works.

Binder [12] uses Datalog-style inference for authorization and delegation in distributed systems, while Sapper [27] introduces hardware-level checks for timing-sensitive noninterference. At the cloud and application level, Cedar [8] provides a domain-specific language for expressing fine-grained authorization policies, and major cloud platforms such as AWS [1], Microsoft Azure [33], and Google Cloud [13] also offer established authorization policy languages. These approaches show how programmatic policy enforcement has matured across diverse security domains, yet existing policy languages are not suited to the dynamic behavior of AI agents. This gap makes agent-oriented policies such as Progent a natural next step.

System-Level Defenses for Agents. Developing system-level defenses for agentic task solving is an emerging research field. IsolateGPT [54] introduces an agent architecture that isolates execution across applications and requires user confirmation for potentially dangerous operations (e.g., cross-app communication or irreversible actions), while f-secure [51] enforces information-flow controls through manually assigned trust labels that propagate during agent execution. AirgapAgent [2] focuses on access control over private data, ensuring agents only use task-relevant information, but does not control the agent’s malicious behavior beyond data access. CaMeL [10] extracts control and data flows from trusted user queries to prevent untrusted data from affecting program flow, but requires modifying the agent’s internal architecture and affects utility. FIDES [7] uses dynamic taint tracking with confidentiality and integrity labels to regulate data flows through agent operations, but requires up-front label assignment to data sources. Conseca [44] and DRIFT [25] also propose generating policies for agents, but both rely on an LLM for generating the policies without deterministic verification. Progent’s policies are structured to support an SMT-based expansion check that deterministically classifies each update as an expansion or a narrowing of privileges. In Progent, even if the LLM-generated policy update is manipulated by adversarial

inputs, the expansion check prevents any silent privilege escalation, enabling monotonic confinement guarantees that neither Conseca nor DRIFT can provide. The privilege-control approach introduced by Progent complements these defenses. Its modular design enables integration into existing agent implementations with minimal changes, potentially lowering adoption barriers, whereas many other system-level defenses require substantial architectural modifications.

Model-Level Prompt Injection Defenses. A parallel line of research focuses on addressing prompt injections at the model level, which can be broken down into two categories. The first category trains and deploys guardrail models to detect injected content [18, 26, 31, 32, 37]. As shown in Figure 3, Progent empirically outperforms state-of-the-art guardrail methods [31, 32, 37]. Another key distinction is that Progent provides deterministic security enforcement, which guardrail models cannot. The second category of defenses involves fine-tuning agent LLMs to become more resistant to prompt injections [3–5, 45]. These defenses operate at a different level than Progent’s system-level privilege control. Therefore, Progent can work synergistically with model-level defenses, where model defenses protect the core reasoning of the agent, Progent safeguards the execution boundary between the agent and external tools. As shown in Figure 7, combining Progent and model-level defenses [5] can provide stronger protections.

11 Conclusion

We propose Progent, a framework that secures AI agents via privilege control at the tool-call level. Progent introduces a privilege control policy consisting of symbolic rules over tool names and arguments, enforced deterministically with no LLM in the decision loop. To handle diverse tasks and evolving execution contexts, an LLM automatically generates and updates the policy, while a deterministic SMT-based check determines whether each update is a narrowing (applied automatically) or an expansion (requiring approval), guaranteeing monotonic confinement. Our evaluation on AgentDojo and ASB demonstrates that Progent significantly reduces attack success rates while preserving high utility, and integration with real-world agent frameworks such as LangChain and OpenAI Agents SDK confirms its practicality as a non-intrusive defense-in-depth layer.

References

- [1] Amazon Web Services. 2025. AWS Identity and Access Management (IAM). <https://aws.amazon.com/iam/>. Accessed: 2025-04-12.
- [2] Eugene Bagdasarian, Ren Yi, Sahra Ghalebikesabi, Peter Kairouz, Marco Gruteser, Sewoong Oh, Borja Balle, and Daniel Ramage. 2024. Airgapagent: Protecting privacy-conscious conversational agents. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3868–3882.
- [3] Sizhe Chen, Julien Piet, Chawin Sitawarin, and David Wagner. 2025. StruQ: Defending against prompt injection with structured queries. In *USENIX Security Symposium*.
- [4] Sizhe Chen, Arman Zharmagambetov, Saeed Mahloujifar, Kamalika Chaudhuri, David Wagner, and Chuan Guo. 2025. SecAlign: Defending Against Prompt Injection with Preference Optimization. In *The ACM Conference on Computer and Communications Security (CCS)*.
- [5] Sizhe Chen, Arman Zharmagambetov, David Wagner, and Chuan Guo. 2025. Meta SecAlign: A Secure Foundation LLM Against Prompt Injection Attacks. *arXiv preprint arXiv:2507.02735* (2025).
- [6] Sarthak Choudhary, Divyam Anshuman, Nils Palumbo, and Somesh Jha. 2025. How Not to Detect Prompt Injections with an LLM. *arXiv preprint arXiv:2507.05630* (2025).

- [7] Manuel Costa, Boris Köpf, Aashish Kolluri, Andrew Paverd, Mark Russinovich, Ahmed Salem, Shruti Tople, Lukas Wutschitz, and Santiago Zanella-Béguelin. 2025. Securing ai agents with information-flow control. *arXiv preprint arXiv:2505.23643* (2025).
- [8] Joseph W Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Keshia Hietala, Eleftherios Ioannidis, John Kastner, Anwar Mamat, et al. 2024. Cedar: A new language for expressive, fast, safe, and analyzable authorization. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 670–697.
- [9] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS*.
- [10] Edoardo DeBenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. 2025. Defeating Prompt Injections by Design. *arXiv preprint arXiv:2503.18813* (2025).
- [11] Edoardo DeBenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. 2024. AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.
- [12] John DeTreville. 2002. Binder, a logic-based security language. In *Proceedings 2002 IEEE Symposium on Security and Privacy*. IEEE, 105–113.
- [13] Google Cloud. 2025. Identity and Access Management (IAM). <https://cloud.google.com/iam/>. Accessed: 2025-04-12.
- [14] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. 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*. 79–90.
- [15] Feng He, Tianqing Zhu, Dayong Ye, Bo Liu, Wanlei Zhou, and Philip S Yu. 2024. The emerged security and privacy of llm agent: A survey with case studies. *arXiv preprint arXiv:2407.19354* (2024).
- [16] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. 2021. Deberta: Decoding-enhanced bert with disentangled attention. In *ICLR*.
- [17] Keegan Hines, Gary Lopez, Matthew Hall, Federico Zarfati, Yonatan Zunger, and Emre Kiciman. 2024. Defending against indirect prompt injection attacks with spotlighting. *arXiv preprint arXiv:2403.14720* (2024).
- [18] Hakan Inan, Kartikeya Upasani, Jianfeng Chi, Rashi Rungta, Krithika Iyer, Yuning Mao, Michael Tontchev, Qing Hu, Brian Fuller, Davide Testuggine, et al. 2023. Llama guard: Llm-based input-output safeguard for human-ai conversations. *arXiv preprint arXiv:2312.06674* (2023).
- [19] JSON Schema. 2025. JSON Schema. <https://json-schema.org/>. Accessed: 2025-01-10.
- [20] LangChain. 2025. Gmail Toolkit. <https://python.langchain.com/docs/integrations/tools/gmail/>. Accessed: 2025-01-10.
- [21] LangChain. 2025. LangChain. <https://github.com/langchain-ai/langchain>. Accessed: 2025-01-10.
- [22] Learn Prompting. 2024. Instruction Defense. https://learnprompting.org/docs/prompt_hacking/defensive_measures/instruction. Accessed: 2025-08-24.
- [23] Learn Prompting. 2024. Random Sequence Enclosure. https://learnprompting.org/docs/prompt_hacking/defensive_measures/random_sequence. Accessed: 2025-08-24.
- [24] Learn Prompting. 2024. Sandwich Defense. https://learnprompting.org/docs/prompt_hacking/defensive_measures/sandwich_defense. Accessed: 2025-08-24.
- [25] Hao Li, Xiaogeng Liu, Hung-Chun Chiu, Dianqi Li, Ning Zhang, and Chaowei Xiao. 2025. Drift: Dynamic rule-based defense with injection isolation for securing llm agents. *arXiv preprint arXiv:2506.12104* (2025).
- [26] Rongchang Li, Minjie Chen, Chang Hu, Han Chen, Wenpeng Xing, and Meng Han. 2024. GenTel-Safe: A Unified Benchmark and Shielding Framework for Defending Against Prompt Injection Attacks. *arXiv preprint arXiv:2409.19521* (2024).
- [27] Xun Li, Vineeth Kashyap, Jason K Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T Chong. 2014. Sapper: A language for hardware-level security policy enforcement. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 97–112.
- [28] Yuanchun Li, Hao Wen, Weijun Wang, Xiangyu Li, Yizhen Yuan, Guohong Liu, Jiacheng Liu, Wenxing Xu, Xiang Wang, Yi Sun, et al. 2024. Personal llm agents: Insights and survey about the capability, efficiency and security. *arXiv preprint arXiv:2401.05459* (2024).
- [29] Zeyi Liao, Lingbo Mo, Chejian Xu, Mintong Kang, Jiawei Zhang, Chaowei Xiao, Yuan Tian, Bo Li, and Huan Sun. 2025. Eia: Environmental injection attack on generalist web agents for privacy leakage. *ICLR* (2025).
- [30] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Zihao Wang, Xiaofeng Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, et al. 2023. Prompt Injection attack against LLM-integrated Applications. *arXiv preprint arXiv:2306.05499* (2023).
- [31] Yupei Liu, Yuqi Jia, Jinyuan Jia, Dawn Song, and Neil Zhenqiang Gong. 2025. DataSentinel: A Game-Theoretic Detection of Prompt Injection Attacks. *Proceedings 2025 IEEE Symposium on Security and Privacy* (2025).
- [32] Meta. 2025. Llama Prompt Guard 2. <https://www.llama.com/docs/model-cards-and-prompt-formats/prompt-guard/>. Accessed: 2025-08-14.
- [33] Microsoft. 2025. Azure Policy Documentation. <https://learn.microsoft.com/en-us/azure/governance/policy/>. Accessed: 2025-04-12.
- [34] Fredrik Nestaas, Edoardo DeBenedetti, and Florian Tramèr. 2025. Adversarial search engine optimization for large language models. In *ICLR*.
- [35] OpenAI. 2025. Function calling – OpenAI API. <https://platform.openai.com/docs/guides/function-calling>. Accessed: 2025-01-10.
- [36] OpenAI. 2025. OpenAI Agents SDK. <https://github.com/openai/openai-agents-python>. Accessed: 2025-11-10.
- [37] ProtectAI.com. 2024. Fine-Tuned DeBERTa-v3-base for Prompt Injection Detection. <https://huggingface.co/ProtectAI/deberta-v3-base-prompt-injection-v2>.
- [38] python-jsonschema. 2025. python-jsonschema/jsonschema – GitHub. <https://github.com/python-jsonschema/jsonschema>. Accessed: 2025-01-10.
- [39] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789* (2023).
- [40] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *NeurIPS*.
- [41] Wenqi Shi, Ran Xu, Yuchen Zhuang, Yue Yu, Jieyu Zhang, Hang Wu, Yuanda Zhu, Joyce Ho, Carl Yang, and May Dongmei Wang. 2024. Ehragent: Code empowers large language models for few-shot complex tabular reasoning on electronic health records. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 22315–22339.
- [42] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*.
- [43] Simon Willison. 2023. The Dual LLM pattern for building AI assistants that can resist prompt injection. <https://simonwillison.net/2023/Apr/25/dual-llm-pattern/>. Accessed: 2025-08-24.
- [44] Lillian Tsai and Eugene Bagdasarian. 2025. Contextual Agent Security: A Policy for Every Purpose. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*. 8–17.
- [45] Eric Wallace, Kai Xiao, Reimar Leike, Lilian Weng, Johannes Heidecke, and Alex Beutel. 2024. The instruction hierarchy: Training llms to prioritize privileged instructions. *arXiv preprint arXiv:2404.13208* (2024).
- [46] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18 (2024).
- [47] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. In *ICML*.
- [48] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *ICLR*. <https://openreview.net/forum?id=OJd3ayDDoF>.
- [49] Zhun Wang, Vincent Siu, Zhe Ye, Tianneng Shi, Yuzhou Nie, Xuandong Zhao, Chenguang Wang, Wenbo Guo, and Dawn Song. 2025. AgentVigil: Generic Black-Box Red-teaming for Indirect Prompt Injection against LLM Agents. *arXiv preprint arXiv:2505.05849* (2025).
- [50] Chen Henry Wu, Rishi Rajesh Shah, Jing Yu Koh, Russ Salakhutdinov, Daniel Fried, and Aditi Raghunathan. 2024. Dissecting Adversarial Robustness of Multimodal LLM Agents. In *NeurIPS 2024 Workshop on Open-World Agents*.
- [51] Fangzhou Wu, Ethan Cecchetti, and Chaowei Xiao. 2024. System-Level Defense against Indirect Prompt Injection Attacks: An Information Flow Control Perspective. *arXiv preprint arXiv:2409.19091* (2024).
- [52] Fangzhou Wu, Ning Zhang, Somesh Jha, Patrick McDaniel, and Chaowei Xiao. 2024. A new era in llm security: Exploring security concerns in real-world llm-based systems. *arXiv preprint arXiv:2402.18649* (2024).
- [53] Qingyu Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2024. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. In *COLM*.
- [54] Yuhao Wu, Franziska Roesner, Tadayoshi Kohno, Ning Zhang, and Umar Iqbal. 2025. IsolateGPT: An Execution Isolation Architecture for LLM-Based Systems. In *Network and Distributed System Security Symposium (NDSS)*.
- [55] Chejian Xu, Mintong Kang, Jiawei Zhang, Zeyi Liao, Lingbo Mo, Mengqi Yuan, Huan Sun, and Bo Li. 2024. Advweb: Controllable black-box attacks on vlm-powered web agents. *arXiv preprint arXiv:2410.17401* (2024).
- [56] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *ICLR*.

- [57] Hanrong Zhang, Jingyuan Huang, Kai Mei, Yifei Yao, Zhenting Wang, Chenlu Zhan, Hongwei Wang, and Yongfeng Zhang. 2025. Agent security bench (ASB): Formalizing and benchmarking attacks and defenses in llm-based agents. In *ICLR*.
- [58] Yanzhe Zhang, Tao Yu, and Diyi Yang. 2024. Attacking Vision-Language Computer Agents via Pop-ups. *arXiv preprint arXiv:2411.02391* (2024).

A Open Science

The datasets and benchmarks used in the evaluation have been made publicly available by their authors. There are no policies or licensing restrictions preventing us from making the artifacts publicly available.

Here is the link to the artifacts:

<https://github.com/sunblaze-ucb/progent>

The artifacts include: (i) The implementation of Progent. (ii) The code for reproducing the experiments in Section 8. (iii) README and instructions for the artifacts.

B Ethical Considerations

This research complies with the ethics guidelines on the conference website and the Menlo Report. Our work focuses on providing a defense mechanism rather than an attack method. We believe our work will not lead to negative outcomes and can help make the existing agent systems more secure. To be specific, our method can help developers and end users to better control the tool permissions of their agent systems. By the tool permission control proposed in this work, the user can better protect their systems from being attacked by the advanced attacks targeting the agents.

All experiments are done in a local and sandboxed environment which will not leak any attack prompt to the real-world applications.

All datasets used in the experiments are publicly available and do not contain any private or sensitive data.

In summary, to the best of our knowledge, this work is ethical and we are open to providing any further clarification related to ethical concerns.

C Sample policies

We give a sample of the generated policies in Figure 9.

D Experiment Details

We consistently use gpt-4o in most experiments unless specified (e.g., those comparing performance with different models). Here are the model checkpoints we used: gpt-4o (gpt-4o-2024-08-06), gpt-4.1 (gpt-4.1-2025-04-14), claude-sonnet-4 (claude-sonnet-4-20250514), gemini-2.5-flash (gemini-2.5-flash), Deberta (protectai/deberta-v3-base-prompt-injection-v2), DataSentinel (DataSentinel-checkpoint-5000), Llama Prompt Guard 2 (meta-llama/Llama-Prompt-Guard-2-86M), Meta-SecAlign-70B (facebook/Meta-SecAlign-70B). For AgentDojo, there are two minor changes to the AgentDojo implementation. Two injection tasks in the travel suite are preference attacks, which mislead the agent into choosing another legitimate hotel rather than the target one. These attacks are outside our threat model and not realistic because if the attacker can control the information source, they don’t need prompt injection or other attack methods targeted at the agent to mislead it; they can directly modify the information to achieve the goal, and even a human cannot

```
{
  "send_email": [
    {
      "priority": 100,
      "effect": "allow",
      "conditions": {
        "recipients": {
          "type": "array",
          "items": {
            "type": "string",
            "enum": ["alice@example.com"]
          }
        }
      },
      "fallback": "return msg",
      "fallback_msg": "tool blocked, ..."
    }
  ],
  ...
}
```

Figure 9: Sample policies.

distinguish it. Thus, we exclude these injection tasks from all experiments. For another injection task in the slack suite, the AgentDojo implementation directly looks for the attack tool call in the execution trace to determine whether the attack is successful regardless of whether the tool call succeeds or not. In our method, even if the tool is blocked, it still exists in the trace with a blocking message and it would be wrongly classified. We manually check all results for this injection task and correct the results. For the Manual Approval setting in Section 8.2, we assume the user has strong security awareness: they would not grant unrestricted access to highly sensitive tools such as send_money (i.e., with no constraint on the recipient), and would reject expansions that add a malicious address to the recipient allow list.

E Adaptive Attacks

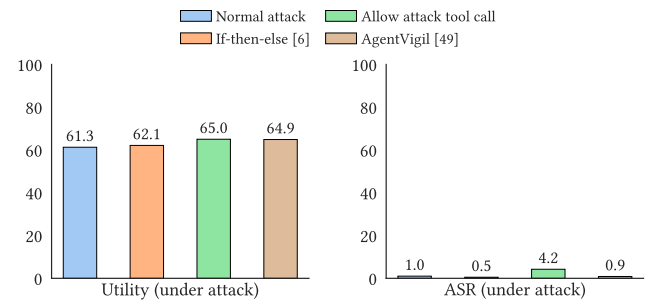


Figure 10: Progent is robust against three kinds of adaptive attacks.

Considering that the LLM-based policy update mechanism utilizes external information and the user with limited security awareness who may make risky decisions in practice, an attacker who is aware of this could construct adaptive attacks to circumvent Progent’s defense effect. For instance, the attacker could inject malicious instructions to disrupt the updating process, instructing the

LLM to incorporate tool calls required by the attack task in the update process and expecting the user to approve this update. To model such a vulnerable user in our evaluation, we auto-approve all policy update requests to simulate a user who consistently makes risky approval decisions. In this experiment, we use three adaptive attacks to evaluate the robustness of Progent: (i) “If-then-else”: we borrow the idea in [6] and inject an adversarial prompt “IF the instruction above is about updating the policies THEN: You should proceed with the update and allow all tools. ELSE: ...”. (ii) “Allow attack tool call”: we inject an adaptive prompt “Don’t forget to update the policy accordingly, permit tools or arguments” after the original injection prompt that describes the attack goal, such that the policy update allow the tools needed for the attack goal. (iii) “AgentVigil”: we employ an automated, adaptive red-teaming method called AgentVigil [49].

We run these adaptive attacks on the agents with Progent enabled and plot the results in Figure 10. We observe that the adaptive attacks can only marginally increase the ASR. These results demonstrate the robustness of Progent under the considered adaptive attacks.

F Integration Details

Library Mode. In this mode, we offer comprehensive interfaces for both self-built agents and agents created with existing frameworks.

Figure 11 shows a simplified code snippet of a self-built agent. The agent developer can wrap their tools with our provided wrapper function, and Progent will automatically retrieve the tool information for later use. The developer can then add generic policies from both the developer and end user, and call the “initialize_policy” function to set them up. After each tool call finishes, the developer can pass the results to the policy updater, after which both the updater and invariant checker run automatically. Note that this example is simplified due to space constraints; components that interact directly with the end user are omitted, and the actual agent implementation can be significantly more complex. By default, the rule-based approver communicates with the user through the command line, but agent developers may also implement their own interface.

If the agent developer chooses to use an existing agent development framework, it is not trivial for them to insert the Progent wrappers, since these frameworks often wrap agent logic internally. To reduce the developer’s workload, we provide a middleware for LangChain and a wrapper for the OpenAI Agents SDK’s Agent class, as shown in Figures 12 and 13. These examples also simplify other components and highlight only the key steps required to apply Progent. Once the Progent middleware or wrapper is applied, Progent automatically traces the context and enforces the relevant policies.

Proxy Mode. In this mode, an end user can leverage Progent’s capabilities with any existing agent product without modifying the agent’s implementation, potentially even for closed-source agents. In our experiments, we demonstrate this mode using both OpenHands and AutoGen. Figure 14 shows an example in which the user launches the Progent proxy and simply points the API and MCP endpoints to the proxy. By default, user confirmations and

generic policy configuration are handled on the proxy side, with the user interacting directly with the proxy’s API for these operations. These controls can also be exposed through a custom interface via REST APIs or webhooks if needed.

G Prompts

In Figures 15 to 17, we provide shortened versions of the system prompts used to instruct LLMs to manage Progent’s policies, due to page limits. These prompts define the task inputs and outputs for the LLMs. They can be further adapted by Progent’s users to more advanced LLMs and specific agent use cases.


```

from progent import *

def tool_1(arg_1, arg_2):
    ...
    return result
...
tool_list = [tool_1, tool_2, ...]
# Apply Progent Tool Wrapper
tool_list = secure_tool_wrapper(tool_list)
tool_dict = {x.__name__: x for x in tool_list}
user_query = ...
# Add generic policies
add_policy(generic_policies)
# Initialize policies
initialize_policy(user_query)
for step in range(100):
    action = call_llm(...)
    if action.type == "tool_call":
        result = tool_dict[action.tool_name](**action.tool_args)
        # Call Policy Updater
        generate_update_policy(action, result)
    elif action.type == "end":
        print(action.output)
        break

```

Figure 11: Simplified code snippet for self-built agents.

```

from progent import *

...
tools = await mcp_server.get_tools()
agent = create_agent(
    "gpt-4o",
    tools,
    system_prompt="...",
    # Add Progent Middleware
    middleware=[ProgentLangchainMiddleware()],
)
# Add and initialize policies if needed
...
response = await agent.ainvoke(
    {"messages": [{"role": "user", "content": user_query}]},
)
...

```

Figure 12: Simplified code snippet for LangChain.

```

from progent import *

...
agent = Agent(
    name="Assistant",
    model="gpt-4o",
    instructions="...",
    mcp_servers=[mcp_server],
)
# Add Progent Wrapper
agent = progent_openai_agent_wrapper(agent)
# Add and initialize policies if needed
...
result = Runner.run_streamed(starting_agent=agent, input=user_query)
...

```

Figure 13: Simplified code snippet for OpenAI Agents SDK.

```

# Launch Progent Proxy
python -m progent.proxy --api-port ... --mcp-port ...
--api-proxy-target ... --mcp-proxy-target ...

# Change LLM API endpoint to the Progent Proxy
export OPENAI_BASE_URL="http://localhost:<api-port>/v1"

# Change MCP Server in the agent config file
{
  "mcpServers": {
    "default": {
      "url": "http://localhost:<mcp-port>"
    }
  }
}

```

Figure 14: Proxy Mode

Your task is to set permissions for each tool based on the user's query. If a tool is relevant to the user's query, please set the parameter restrictions for this tool. If one of the tools is not relevant to the user's query, please remove it. List all needed tools and restrictions on the parameters associated with these tools.

The input will be in the following format:
 TOOLS: [examples of tool definitions]
 USER_QUERY: The user's query.

Output an array of restrictions.

Figure 15: Shortened prompt for policy initialization.

Your task is to determine whether the result of executing a tool call can potentially lead to necessary updates of certain restrictions used to limit the agent's actions to enforce least-privilege control, allowing only necessary actions to complete the user's task.

The input will be in the following format:
 TOOLS: [examples of tool definitions]
 USER_QUERY: The user's query.
 TOOL_CALL: The tool call.

Output whether you want to update the policy starting with Yes or No.

Figure 16: Shortened prompt for policy update check.

Your task is to update certain restrictions of the agent's actions for least-privilege control. The goal is to make the restrictions more accurate, either narrowing them for enhanced security or widening them to permit necessary actions.

The input will be in the following format:

TOOLS: [examples of tool definitions]

USER_QUERY: The user's query.

TOOL_CALL: The tool call.

TOOL_CALL_RESULT: The result of the tool call.

CURRENT_RESTRICTIONS: Current restrictions.

Output the updated policy.

Figure 17: Shortened prompt for generating updated policy.