MALSPOT: DETECTING AND LOCALIZING MALICIOUS CODE IN LLM AGENT TOOLS

Anonymous authors
Paper under double-blind review

ABSTRACT

Tools extend the capabilities of large language model (LLM) agents, but malicious code in these tools can compromise users' security and privacy. Existing static detectors struggle to distinguish malicious behavior from legitimate functionality because the same operations can serve either purpose. For example, a network request may send credentials to a service for authentication or leak them to an attacker. In this work, we propose MalSpot for statically detecting and localizing malicious behavior in agent tool code. MalSpot first identifies statements that interact with the environment and collects the code that assigns values to the variables they use. With this context, it combines static and LLM-based analysis to select potentially malicious code. Multiple LLMs then independently analyze the selected code together with the tool's complete source code to confirm or revise the initial assessments. MalSpot combines their tool-level decisions by majority vote. For a tool labeled malicious, it reports the location of the statement most frequently identified as malicious. Experiments on three datasets show that MalSpot achieves high true-positive rates with low false-positive rates, outperforming the baselines. With small open-weight models, it achieves performance comparable to direct detection and localization with frontier closed-source LLMs such as Claude-Opus-5 and GPT-5.5.

1 INTRODUCTION

Large language model (LLM) agents use tools to interact with external environments and perform user tasks (Yao et al., 2023; Schick et al., 2023). These tasks include editing code (Yang et al., 2024), processing documents (Xie et al., 2024), and managing email (Ruan et al., 2024). Users can extend these capabilities by adding third-party tools (Anthropic, 2024; MCP.so, 2025) and skills (Li et al., 2026b). Once a tool is added, users can allow the agent to invoke it during a task without approving each call. When invoked, the tool executes code that may read files, modify data, or access services using the user's credentials. Depending on its permissions, a tool running on the user's computer may also access files and credentials unrelated to the task.

An attacker can abuse this access by embedding malicious code in a tool while preserving its legitimate functionality (Hu et al., 2026; Yuan et al., 2026). When an agent invokes the compromised tool as part of a user task, the embedded code can steal sensitive data or credentials, corrupt stored data, or hijack computing resources for the attacker's benefit. Prior work (Hu et al., 2026) has shown that attackers can use LLMs to generate malicious code and embed it into existing tools. Analyses of real-world agent tools and skills (Huang et al., 2026; Liu et al., 2026) have also uncovered malicious behavior, including code that secretly uploads user files to external servers.

Malicious code can be detected through dynamic analysis or static inspection. Dynamic analysis (Yin et al., 2007; Huang et al., 2026) observes behavior during tool execution, but a test may miss malicious code that requires particular inputs to trigger it. Static inspection (Christodorescu et al., 2005; PyCQA, 2014; Semgrep, 2020; Gao et al., 2025; Zahan et al., 2025; Tencent Zhuque Lab, 2025; Cisco AI Defense, 2025; Sha et al., 2025) examines the tool's source code without executing it and does not require inputs that trigger malicious behavior. It therefore allows detection before the tool accesses user files, credentials, or services. Tool marketplaces can use this approach to screen tools and block those flagged as malicious before listing them for users to download.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Existing methods for static code inspection fall into two categories: traditional detectors and LLM-driven detectors. Traditional detectors (PyCQA, 2014; Semgrep, 2020; Li et al., 2023; Arp et al., 2014) use predefined rules, program analysis, or classifiers trained on code features to identify suspicious code. LLM-driven detectors (Tencent Zhuque Lab, 2025; Cisco AI Defense, 2025; Sha et al., 2025) use LLMs to assess whether the code implements malicious behavior. Related work (Wang et al., 2024; Li et al., 2025) also combines LLMs with program analysis for bug and vulnerability detection. For malicious tool detection, however, evaluations (Hu et al., 2026; Huang et al., 2026) show that achieving both high true-positive rates and low false-positive rates remains difficult. A main difficulty is that similar operations can serve legitimate or malicious purposes. For example, a network request may send credentials to a service for authentication or leak them to an attacker. Distinguishing these cases requires checking where the credentials come from, where they are sent, and why the tool sends them.

In this work, we propose MalSpot, a method for statically detecting and localizing malicious behavior in agent tool code. Our key idea is to first extract code that interacts with the environment, for example by reading files or sending network requests, and assess whether it is potentially malicious. We then analyze potentially malicious code together with the tool's complete source code to confirm or revise the initial assessments. MalSpot consists of three steps. At Step I, MalSpot uses an LLM to identify statements in the tool's source code that interact with the environment and collects the code that assigns values to the variables used by each statement. For example, for a network request, this includes the code that prepares the data to be sent and specifies the destination address. At Step II, with this context, static analysis checks for predefined suspicious operations, while an LLM checks each statement's role in the tool and whether it causes harm. Based on these analyses, MalSpot selects potentially malicious code for further analysis. At Step III, multiple LLMs independently analyze the potentially malicious code together with the tool's complete source code. MalSpot combines their tool-level decisions by majority vote. For a tool labeled malicious, it reports the location of the statement flagged most frequently by each LLM. This localization can be used to assist human reviewers in inspecting the malicious code.

We evaluate MalSpot on three existing datasets (Hu et al., 2026; Huang et al., 2026; Tencent Zhuque Lab, 2026) for malicious tool detection, comparing it with traditional detectors (PyCQA, 2014; Semgrep, 2020; Apiiro, 2025; Shariffdeen et al., 2025) and several LLM-based approaches (Zhao et al., 2026; Guo et al., 2026; Wang et al., 2026a). Our results show that MalSpot achieves high true-positive rates while maintaining low false-positive rates, outperforming existing detectors as well as direct detection and localization using the same LLMs. When paired with small open-weight models, MalSpot is able to achieve performance comparable to direct detection and localization using frontier closed-source LLMs. We further evaluate the locations identified by MalSpot against the annotated malicious code in MalTool (Hu et al., 2026) and find that MalSpot also outperforms the baselines on localization accuracy.

Our main contributions are as follows:

- We propose MalSpot for both detection and localization of malicious behavior in agent tool code. Given a tool, MalSpot returns a tool-level decision with the locations of malicious code.
- We introduce the approach of combining static and LLM-based analysis to select potentially malicious code, retaining code flagged by static analysis even when the LLM initially labels it benign. Multiple LLMs then independently analyze the selected code together with the tool's complete source code to distinguish malicious behavior from legitimate functionality.
- We evaluate MalSpot on three datasets against traditional and LLM-driven detectors. The results show that MalSpot outperforms the baselines in both detection and localization, achieving high true-positive rates with low false-positive rates.

2 Related Work

2.1 LLM Agent Tools

An agent tool consists of metadata and executable code (Anthropic, 2024). The metadata includes the tool's name, a description of its functionality, and an input schema specifying the accepted arguments. Given a user task, an LLM agent uses this information to select a tool and generate input arguments (Qin et al., 2024; Patil et al., 2024). The tool executes its code with these arguments and

returns a result, which the agent uses as context for subsequent reasoning and tool calls (Yao et al., 2023). During execution, the tool may read local files, modify application state, or access external services within its granted permissions (OpenAI, 2024).

2.2 Malicious Tool Attacks

Malicious tool attacks can exploit both tool metadata (Zhang et al., 2026; Zong et al., 2026) and executable code (Hu et al., 2026; Yuan et al., 2026). At the metadata level, attackers modify tool names, descriptions, and parameter schemas to encourage invocation and elicit sensitive information as arguments (Mo et al., 2025). They can also embed instructions in tool descriptions to change an agent's actions (Shi et al., 2026; Wang et al., 2026b). These instructions can induce the agent to misuse other legitimate tools even when the tool carrying them is never invoked (Li et al., 2026a).

Attacks through tool code embed malicious operations that abuse the tool's access to user data and other resources. Attackers can use coding LLMs to generate these operations and embed them into existing tools while preserving their intended functionality (Hu et al., 2026). Related attacks on agent skills disguise malicious code as ordinary programming mistakes and trigger malicious behavior through attacker-controlled inputs (Lin & Yu, 2026). Our work focuses on detection and localization of malicious behavior in tool source code.

2.3 Malicious Tool Detection

Malicious tool detection includes two complementary approaches: metadata-based detection and code detection. Metadata-based detection (Cisco AI Defense, 2025) inspects tool names and descriptions for attacks such as prompt injection (Liu et al., 2024; Shi et al., 2026; Wang et al., 2025). Code detection identifies malicious behavior in the tool's code even when its metadata appears benign. Code detection methods can be further grouped into traditional detectors and LLM-driven detectors. Traditional detectors (PyCQA, 2014; Semgrep, 2020; Li et al., 2023; Arp et al., 2014; Shariffdeen et al., 2025) use predefined rules and code analysis to identify suspicious API calls, obfuscated execution, and sensitive data flows. They also include classifiers trained on API usage and structural code features (Sejfia & Schäfer, 2022). LLM-driven detectors (Zhao et al., 2026; Guo et al., 2026; Tencent Zhuque Lab, 2025; Cisco AI Defense, 2025; Sha et al., 2025) use LLMs to assess whether code implements malicious behavior. They analyze the available code directly, construct code slices around sensitive API calls, or retrieve behavioral patterns to support detection. MalSpot combines static and LLM-based analysis to select potentially malicious code, including suspicious code initially labeled benign by the LLM. Multiple LLMs then independently analyze the selected code together with the tool's complete source code to confirm or revise the initial assessments.

3 Problem Setup

3.1 Problem Formulation

We denote an agent tool by $u = (M_u, C_u)$, where M_u is its metadata and C_u is its code. The metadata includes the tool's name, description, and input/output interface. We study the static detection and localization of malicious behavior implemented in C_u.

We define malicious behavior as attacker-chosen operations that compromise the confidentiality, integrity, or availability of the environment such as file system and network. Examples include exfiltrating credentials, modifying agent memory without authorization, and hijacking computational resources. We label C_u as malicious if it implements such behavior and as benign otherwise. A static detector D predicts this label for each tool by analyzing C_u without executing it:

$$\hat{y}_u = D(C_u) \in \{\text{benign}, \text{malicious}\}. \tag{1}$$

For a C_u detected as malicious, localization seeks to identify the statements in the code responsible for the malicious behavior. It returns a set $\hat{\mathcal{L}}_u$ of code locations, each specifying a file and line number in C_u.

3.2 THREAT MODEL

Attacker's goal: We consider an attacker who supplies a third-party tool for use by an LLM agent. The attacker aims to abuse the permissions granted to the tool to compromise the environment such as file system and network when the tool is invoked.

Attacker's knowledge and capabilities: The attacker knows the interface used for tool invocation and controls the tool's metadata M_u and source code C_u. The attacker can create a tool with malicious functionality or inject malicious operations into an existing tool while preserving its legitimate functionality. At runtime, the tool can access its input arguments as well as any files, credentials, agent memory, or computational resources available under its execution permissions.

Defender's access and assumptions: We assume that the defender has access to the tool's source code C_u. The detector classifies the code as benign or malicious and, when it detects malicious behavior, reports the locations of the malicious code. Tool platforms can use the detector to screen tools before making them available to users, while users can run it locally before allowing their agents to invoke a tool.

4 MALSPOT

4.1 OVERVIEW

MalSpot detects and localizes malicious behavior in tool's code using static analysis and LLMs. Our key idea is to first assess individual operations and then analyze potentially malicious operations together with the tool's complete source code. An operation's inputs help explain what it does, but whether it is malicious also depends on how it is used in the tool. The analysis therefore checks the suspected operations against the tool's complete code to confirm or revise the initial assessment.

MalSpot consists of three steps: 1) Step I: *Trace Construction*, 2) Step II: *Trace Analysis*, and 3) Step III: *Tool-level Detection and Malicious Code Localization*. Step I: Trace Construction uses an LLM to identify statements that interact with the environment, called *anchors*. For each anchor, it collects the assignments that prepare its inputs and combines them with the anchor to form a *trace*. Given these traces, Step II: Trace Analysis combines static and LLM-based analysis to select potentially malicious traces for further analysis. The selected traces guide Step III: Tool-level Detection and Malicious Code Localization, in which multiple LLMs analyze them independently together with the complete source code and return a tool-level label and, for a tool detected as malicious, the supporting trace. MalSpot combines their labels by majority vote. For a tool detected as malicious, it uses the most frequently returned trace to locate the corresponding anchor in the source code. We illustrate these three steps with an example tool in Appendix A.

4.2 STEP I: TRACE CONSTRUCTION

Discovering anchors: An anchor is a statement in the tool's code that interacts with the environment, such as the file system or the network. For example, `requests.post(url, json=payload)` is an anchor that sends data over the network. Given C_u, MalSpot uses an LLM to examine the tool's entry function and other functions used by the tool and propose anchors. However, due to hallucination, the LLM may propose an anchor that does not exist in the code. Therefore, for each proposed anchor, we look for a matching node in the tool's abstract syntax tree (AST), which represents the tool's code as a tree with nodes for function calls and assignments. A proposed anchor is retained only if it matches a node in the tree. The retained anchors form A_u.

Constructing traces: An anchor shows how an operation interacts with the environment. Whether that operation is malicious also depends on the data it uses. For each anchor $a \in A_u$, we therefore start from the variables used by the anchor and locate each variable's nearest preceding assignment in the same function. We repeat this process for the variables of each preceding assignment until no earlier local assignment can be found. We arrange the recovered assignments into sequences ending at the anchor. Together, these sequences form the trace T_a.

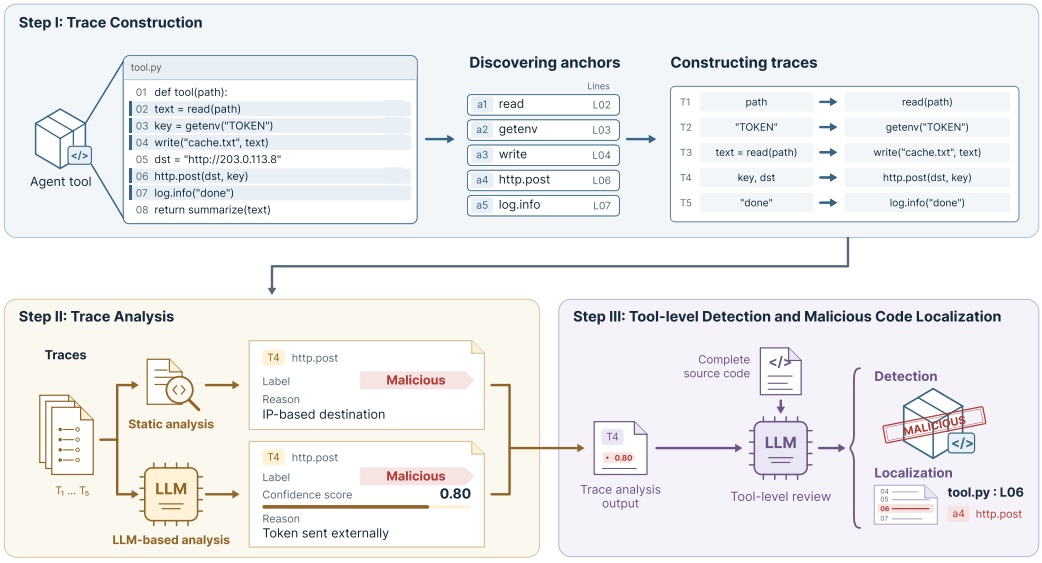

Figure 1: The overall pipeline of MalSpot.

4.3 Step II: Trace Analysis

After constructing traces for a tool, we next examine each trace and select potentially malicious traces for further analysis. When examining traces, an LLM can identify harmful behavior not captured by static analysis, but may also overlook suspicious traces. Therefore, MalSpot combines static analysis and LLM-based analysis to select potentially malicious traces.

Static analysis: For each trace T_a, static analysis examines it against a predefined set of suspicious operations. For example, it identifies requests whose URLs use IP addresses instead of domain names, which can make it harder to identify the service receiving the request. The analysis flags any trace containing a suspicious operation and records why it is flagged.

LLM-based analysis: For LLM-based analysis, MalSpot uses an LLM to examine the trace T_a together with the tool's entry function. It uses this information to infer the tool's intended functionality, understand the operation's role in the tool, and assess whether it causes harm. The model returns a preliminary label $s_a \in \{\text{benign}, \text{malicious}\}$, a confidence score $c_a \in [0, 1]$ for that label, and a brief reason.

A trace T_a is selected for further analysis if static analysis identifies a suspicious operation in it, the LLM labels it malicious, or the LLM's confidence score c_a falls below a predefined *confidence threshold* τ. The selected trace set is denoted by R_u.

4.4 Step III: Tool-level Detection and Malicious Code Localization

For a tool with a selected trace set R_u, MalSpot uses an LLM to examine these traces together with the tool's complete source code C_u. The model jointly considers the potentially malicious traces and the source code to determine whether the operations in these traces constitute malicious behavior in the context of the tool as a whole. If it finds malicious behavior in any selected trace, it labels the tool as malicious and returns the corresponding malicious trace. MalSpot then locates the anchor of the returned trace in C_u and reports its file and line number.

Combining multiple LLMs via majority vote: Different LLMs may overlook different types of malicious behavior. To improve detection and localization, MalSpot can combine the tool-level analyses of multiple LLMs (we use three in our experiments). Each model independently analyzes the same selected traces R_u and source code C_u. MalSpot combines their tool-level detection labels using majority voting: a tool is labeled malicious if a majority of the models label it malicious, and

benign otherwise. For a tool labeled malicious, MalSpot selects the trace returned by the largest number of models and reports the source location of its anchor.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Datasets: We use three datasets: MalTool (Hu et al., 2026) and Connor (Huang et al., 2026) for MCP tools, and SkillTrustBench (Tencent Zhuque Lab, 2026) for agent skills. From MalTool, we sample 1,200 malicious tools, with 100 from each of 12 malicious behavior categories. These tools are constructed by embedding verified malicious code into real-world tools. From Connor, we retain 60 manually constructed MCP tools containing malicious code. SkillTrustBench includes both synthetic and real-world malicious skills. We retain 400 skills containing malicious code. To evaluate false positives, we use 1,200 benign tools from MalTool's real-world collection and 400 benign skills from SkillTrustBench. We refer to these benign sets as MCP Tool and Skill, respectively. Details of datasets and data preprocessing are described in Appendix B.

Baseline methods: We compare MalSpot with traditional detectors and other LLM-driven detectors. The traditional detectors include Bandit (PyCQA, 2014), Semgrep with the Apiiro malicious-code ruleset (Semgrep, 2020; Apiiro, 2025), GuardDog (Datadog, 2026), Bandit4Mal (lyvd, 2026), HERCULE (Shariffdeen et al., 2025), and SkillSpector (Static) (NVIDIA, 2026). The LLM-driven detectors include direct LLM detection and localization (Direct) and seven existing detectors: Tencent A.I.G (Tencent Zhuque Lab, 2025), Cisco Behavioral (Cisco AI Defense, 2025), AntGroup MCPScan (Sha et al., 2025), MalTotal (Zhao et al., 2026), PyGuard (Guo et al., 2026), MalSkills (Wang et al., 2026a), and SkillSpector (Static + LLM) (NVIDIA, 2026). Direct prompts an LLM to classify a tool as benign or malicious using its complete source code and, if classified as malicious, to further identify source locations that support its decision. We additionally evaluate *Combined Detector*, which combines five of these detectors and labels a tool malicious if any of them does so. Details of each detector are shown in Appendix C.

Models and settings: We evaluate Direct and MalSpot using three open-weight LLMs: Gemma-4-26B-A4B (Team et al., 2026), Qwen3.5-9B (Qwen Team, 2026), and Mistral-Small-3.2-24B (Mistral AI, 2025). In MalSpot, Qwen3.5-9B proposes anchors and performs the LLM-based analysis of their traces. The three models then independently perform tool-level analysis using the same selected traces and the tool's complete source code. For both Direct and MalSpot, we report results for the majority vote by default. We set the temperature to 0 for these three models and the confidence threshold for trace selection to $\tau = 0.85$. We additionally evaluate MalSpot with GPT-5.5 (OpenAI, 2026) and Claude-Opus-5 (Anthropic, 2026) as the LLM for tool-level analysis in Step III and compare it with Direct using the same models. Further implementation details are provided in Appendix D.

Evaluation metrics: We report the *true-positive rate (TPR)* as the fraction of malicious tools detected and the *false-positive rate (FPR)* as the fraction of benign tools labeled malicious. We evaluate localization on the MalTool dataset, since it provides annotations marking the locations of malicious code. *Localization recall (Recall$_{Loc}$@k)* is the fraction of malicious tools that are detected as malicious and have at least one correct location among their first k validated source locations. *Localization precision (Precision$_{Loc}$@k)* is the fraction of tools labeled malicious across the malicious and benign sets that are malicious and have at least one correct location among their first k validated source locations. *Localization F1 score (F1$_{Loc}$@k)* is the harmonic mean of Recall$_{\text{Loc}}$@k and Precision$_{\text{Loc}}$@k. We validate reported file paths and line numbers against the source code, and count a location as correct if it falls within the annotated malicious code in the same file. We use $k = 3$ by default. Further evaluation details are provided in Appendix E.

5.2 EXPERIMENTAL RESULTS

MalSpot achieves high TPRs with low FPRs: Table 1 reports the TPRs and FPRs of MalSpot and Direct on three datasets. Compared with Direct using the same open-weight LLMs, MalSpot achieves higher TPRs on MalTool and SkillTrustBench and lower FPRs on both benign sets. The

Table 1: Detection performance of Direct and MalSpot (%).

		Malicious (TPR↑)			Benign (FPR↓)	
Method	Model	MalTool	Connor	SkillTrustBench	MCP Tool	Skill
Direct	Gemma-4-26B-A4B	81.42	**100.00**	85.50	0.67	**0.00**
	Qwen3.5-9B	85.08	98.33	63.00	2.50	2.00
	Mistral-Small-3.2-24B	71.83	**100.00**	69.25	0.92	1.75
	Majority vote	81.67	**100.00**	70.50	1.00	1.25
	GPT-5.5	89.33	**100.00**	91.25	1.75	2.00
	Claude-Opus-5	93.83	**100.00**	95.75	0.50	0.75
MalSpot	Gemma-4-26B-A4B	90.83	98.33	93.75	0.42	0.25
	Qwen3.5-9B	82.50	98.33	86.50	0.50	0.25
	Mistral-Small-3.2-24B	91.25	98.33	94.00	1.50	1.50
	Majority vote	91.83	98.33	94.00	0.50	0.25
	GPT-5.5	95.00	98.33	93.25	1.33	1.00
	Claude-Opus-5	**95.08**	**100.00**	**96.50**	**0.25**	0.50

Table 2: Comparison with traditional detectors (%).

	Malicious (TPR↑)			Benign (FPR↓)	
Method	MalTool	Connor	SkillTrustBench	MCP Tool	Skill
Bandit	66.00	**98.33**	91.00	11.75	32.75
Semgrep + Apiiro	23.58	1.67	83.00	12.67	52.75
GuardDog	17.92	46.67	28.25	**0.17**	4.25
Bandit4Mal	46.58	48.33	**97.50**	8.25	48.25
HERCULE	28.17	43.33	36.25	1.08	20.00
SkillSpector (Static)	4.33	46.67	61.25	1.50	4.50
MalSpot	**91.83**	**98.33**	94.00	0.50	**0.25**

Table 3: Comparison with other LLM-driven detectors and Combined Detector (%).

	Malicious (TPR↑)			Benign (FPR↓)	
Method	MalTool	Connor	SkillTrustBench	MCP Tool	Skill
Tencent A.I.G	64.92	96.67	61.50	20.58	6.25
Cisco Behavioral	63.17	96.67	82.75	0.83	14.50
AntGroup MCPScan	26.33	86.67	87.00	13.08	50.75
MalTotal	27.25	85.00	89.25	**0.08**	**0.25**
PyGuard	32.50	86.67	64.75	3.75	2.75
MalSkills	48.50	88.33	98.75	10.00	12.75
SkillSpector (Static + LLM)	91.42	88.33	88.75	15.92	15.25
Combined Detector	**94.25**	**100.00**	**100.00**	44.75	87.50
MalSpot	91.83	98.33	94.00	0.50	**0.25**

largest TPR gain is on SkillTrustBench, where TPR increases from 70.50% to 94.00% while FPR on Skill decreases from 1.25% to 0.25%. On Connor, both methods detect nearly all malicious tools. The improvements in both metrics show that MalSpot helps the same LLMs distinguish malicious behavior from legitimate functionality.

MalSpot with open-weight LLM is competitive with Direct with frontier LLMs: Table 1 also compares Direct and MalSpot using GPT-5.5 and Claude-Opus-5. Using small open-weight LLMs, MalSpot outperforms Direct with GPT-5.5 on MalTool and SkillTrustBench. Its TPRs are within two percentage points of Direct with Claude-Opus-5 across all three datasets, with the same or lower FPRs. When using either frontier LLM at Step III, MalSpot also achieves higher TPRs on MalTool and SkillTrustBench and lower FPRs on both benign sets than Direct with the same model. These results show that MalSpot with small open-weight LLMs achieves detection performance compa-

Table 4: Localization performance on the MalTool dataset (%).

Type	Method	$\text{Recall}_{\text{Loc}}@3\uparrow$	$\text{Precision}_{\text{Loc}}@3\uparrow$	$\text{F1}_{\text{Loc}}@3\uparrow$
Traditional detectors	Bandit	46.58	59.91	52.41
	Semgrep + Apiiro	22.25	61.38	32.66
	GuardDog	14.83	82.03	25.12
	Bandit4Mal	33.92	61.85	43.81
	HERCULE	22.83	78.06	35.33
LLM-driven detectors	Cisco Behavioral	62.42	**97.53**	76.12
	MalTotal	25.33	92.68	39.79
	MalSkills	34.67	59.26	43.74
	Direct	68.58	82.96	75.09
	MalSpot	78.58	85.11	81.72
	Direct (GPT-5.5)	72.92	80.05	76.32
	MalSpot (GPT-5.5)	77.83	80.80	79.29
	Direct (Claude-Opus-5)	79.83	84.63	82.16
	MalSpot (Claude-Opus-5)	**82.58**	86.63	**84.56**

rable to Direct with frontier LLMs. Using frontier LLMs for tool-level analysis further improves detection over Direct with the same models.

MalSpot outperforms existing detectors: Tables 2 and 3 compare MalSpot with traditional detectors, other LLM-driven detectors, and Combined Detector. The baselines struggle to achieve high TPRs and low FPRs simultaneously across the datasets. For example, GuardDog and MalTotal produce few false positives on MCP Tool but miss most malicious tools in MalTool. Combined Detector detects all malicious skills in SkillTrustBench but also labels 87.50% of benign skills as malicious. Among these methods, only MalSpot achieves TPRs above 91% on all three datasets while keeping FPRs at or below 0.50% on both benign sets, which outperforms other detectors.

MalSpot accurately localizes malicious code: Table 4 compares the malicious code localization performance of MalSpot and the baselines on MalTool. With both small open-weight and frontier LLMs, MalSpot improves all three localization metrics over Direct using the same models. With small open-weight LLMs, its $\text{F1}_{\text{Loc}}@3$ reaches 81.72%, compared with 75.09% for Direct and 76.12% for Cisco Behavioral. This configuration also outperforms Direct with GPT-5.5 and achieves performance comparable to Direct with Claude-Opus-5. Using Claude-Opus-5 in Step III, MalSpot achieves the highest $\text{Recall}_{\text{Loc}}@3$ and $\text{F1}_{\text{Loc}}@3$, at 82.58% and 84.56%, respectively.

5.3 Ablation Studies

Tool-level analysis reduces false positives: Table 5 compares the TPRs and FPRs of the full MalSpot and three ablated variants on MalTool. The first disables static analysis in Trace Analysis, selecting traces only when the LLM labels them malicious or has low confidence. The second retains tool-level analysis but provides the selected traces and their preliminary assessments without the complete source code. The third removes tool-level analysis and uses the three LLMs to assess traces independently in Trace Analysis. Each model labels a tool malicious if it labels at least one trace malicious; the three tool labels are then combined by majority vote.

Removing tool-level analysis substantially increases FPR, while TPR remains similar. Omitting the complete source code from this analysis also increases FPR and lowers TPR. Removing static analysis leads to a slight decrease in TPR and a small increase in FPR. These results show that tool-level analysis with the complete source code reduces false positives while maintaining high TPR, and static analysis further improves detection.

MalSpot remains effective with different LLMs in Steps I and II: Table 6 compares the TPRs and FPRs of MalSpot on MalTool with three different LLMs in Steps I and II. Each LLM is used for both Step I and II, while the LLMs used in Step III remain fixed. All three LLMs achieve high TPRs with low FPRs. Steps I and II can prioritize coverage by retaining suspicious traces even when the initial assessment is uncertain. Step III then analyzes these traces with the complete source code to determine whether they contain malicious behavior.

Table 5: Component ablations on the MalTool dataset (%).

	Gemma-4		Qwen3.5		Mistral		Majority vote	
Configuration	TPR↑	FPR↓	TPR↑	FPR↓	TPR↑	FPR↓	TPR↑	FPR↓
Full MalSpot	**90.83**	**0.42**	82.50	0.50	91.25	1.50	**91.83**	**0.50**
w/o static analysis	90.83	0.67	79.25	**0.42**	90.92	**1.00**	91.25	0.58
w/o full source	90.58	0.58	89.08	1.17	90.92	3.08	90.67	1.33
w/o tool-level analysis	90.50	0.92	**91.75**	4.33	**93.50**	4.42	91.50	3.58

Effect of the confidence threshold: Figure 3 in Appendix shows the TPRs and FPRs of MalSpot on MalTool at different confidence thresholds τ. A higher τ allows more traces initially labeled benign in Step II to enter tool-level analysis. Both metrics remain unchanged as τ varies from 0.75 to 0.90. Beyond this range, FPR increases while the final TPR changes little. These results show that detection remains stable around our default threshold of $\tau = 0.85$.

5.4 False-positive analysis

To analyze false positives, we group the benign tools in MalTool into 19 categories by functionality. Appendix G.1 shows the details of grouping and full distribution.

Ambiguous tool behavior: Figure 2 compares the FPRs of Direct and MalSpot across tool categories using the same three LLMs. For both methods, Web Scraping has the highest FPR, followed by Data & AI Computation. MalSpot lowers FPR in both categories. Our inspection of its false positives (tool examples shown in Appendix G.2) shows that some tools perform authentication or connection checks whose purpose is unclear from the source code. Others contain potentially unsafe code, such as disabling certificate verification or passing unchecked input to `eval`. These operations pose security risks, but do not by themselves establish malicious behavior.

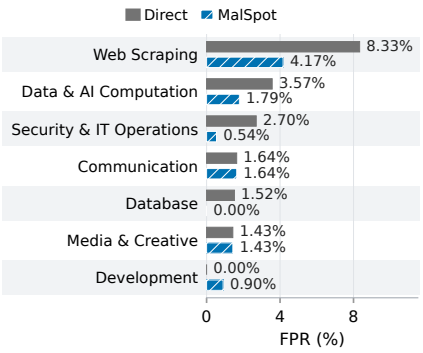

Figure 2: FPRs by category on MCP Tool for MalSpot and Direct.

Lower FPR matters at scale: Marketplaces such as MCPMarket (Market, 2025) and SkillsMP (SkillsMP, 2026) now list over one million tools. Screening one million tools with Direct would incur estimated official API costs of $52,000 with Claude-Opus-5 and $37,000 with GPT-5.5. In addition to these costs, false positives can create substantial manual review work. With small open-weight LLMs, MalSpot achieves an FPR of 0.25% on benign skills, compared with 0.75% for Direct with Claude-Opus-5 and 2.00% with GPT-5.5. At these rates, MalSpot would produce 5,000 and 17,500 fewer false positives per million benign skills than the two Direct baselines, respectively. On benign MCP tools, it matches Direct with Claude-Opus-5 and would produce 12,500 fewer false positives per million benign tools than Direct with GPT-5.5. Even small FPR reductions can therefore spare marketplaces thousands of unnecessary reviews or mistaken rejections of benign tools.

6 Conclusion and Future Work

In this work, we propose MalSpot for static detection and localization of malicious behavior in agent tool code. It first assesses code that interacts with the environment, then analyzes potentially malicious code with the tool's complete source code. Experiments on three datasets show that MalSpot outperforms existing detectors and direct detection and localization with the same LLMs, achieving high TPRs with low FPRs. With small open-weight models, it achieves detection performance comparable to direct detection and localization with frontier closed-source LLMs. Future work includes supporting other programming languages and integrating runtime controls.

AI USE STATEMENT

In this work, we used LLMs to assist with polishing the text. The LLMs used within MalSpot and the evaluated baselines are described in the experimental setup.

ETHICS STATEMENT

MalSpot is intended for pre-deployment security analysis of agent tools. It inspects source code without executing candidate tools and reports source locations that support its decisions. This allows developers and platform operators to examine and locate relevant behavior before a tool is used by an LLM agent. The examples in this paper explain the detection and localization task and support the defensive purpose of the study.

REPRODUCIBILITY STATEMENT

Section 4 describes MalSpot, and Appendix A illustrates its steps using an example tool. Section 5 presents the experimental setup and evaluation metrics, with details on dataset preprocessing, baseline configurations, model settings, and localization in Appendices B–E.2. The tool-level analysis prompt and ablation settings are provided in Appendices D.3 and F, respectively. We will publicly release our code and processed evaluation datasets upon acceptance.

REFERENCES

Anthropic. Introducing the model context protocol. https://www.anthropic.com/news/model-context-protocol, 2024.

Anthropic. Claude Opus 5 system card, 2026. URL https://anthropic.com/claude-opus-5-system-card.

Apiiro. malicious-code-ruleset. https://github.com/apiiro/malicious-code-ruleset, 2025.

Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. Semantics-aware malware detection. In *IEEE S&P*, 2005.

Cisco AI Defense. Cisco MCP Scanner. https://github.com/cisco-ai-defense/mcp-scanner, 2025.

Datadog. GuardDog. https://github.com/DataDog/guarddog, 2026.

Xingan Gao, Xiaobing Sun, Sicong Cao, Kaifeng Huang, Di Wu, Xiaolei Liu, Xingwei Lin, and Yang Xiang. {MalGuard}: Towards {Real-Time}, accurate, and actionable detection of malicious packages in {PyPI} ecosystem. In *USENIX Security*, 2025.

Wenbo Guo, Chengwei Liu, Ming Kang, Yiran Zhang, Jiahui Wu, Zhengzi Xu, Vinay Sachidananda, and Yang Liu. Cutting the gordian knot: detecting malicious pypi packages via a knowledge-mining framework. *arXiv preprint arXiv:2601.16463*, 2026.

Yuepeng Hu, Yuqi Jia, Mengyuan Li, Dawn Song, and Neil Gong. Maltool: Malicious tool attacks on llm agents. *arXiv preprint arXiv:2602.12194*, 2026.

Yiheng Huang, Zhijia Zhao, Bihuan Chen, Susheng Wu, Zhuotong Zhou, Yiheng Cao, Xin Hu, and Xin Peng. From component manipulation to system compromise: Understanding and detecting malicious mcp servers. *arXiv preprint arXiv:2604.01905*, 2026.

Ningke Li, Shenao Wang, Mingxi Feng, Kailong Wang, Meizhen Wang, and Haoyu Wang. Malwukong: Towards fast, accurate, and multilingual detection of malicious code poisoning in oss supply chains. In *IEEE ASE*, 2023.

Ruiqi Li, Zhiqiang Wang, Yunhao Yao, and Xiang-Yang Li. Mcp-itp: An automated framework for implicit tool poisoning in mcp. *arXiv preprint arXiv:2601.07395*, 2026a.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026b.

Ziyang Li, Saikat Dutta, and Mayur Naik. Iris: Llm-assisted static analysis for detecting security vulnerabilities. In *ICLR*, 2025.

Yu-Ting Lin and Chia-Mu Yu. Phantomskill: Malicious code injection in agent skill ecosystems. *arXiv preprint arXiv:2606.19191*, 2026.

Yi Liu, Zhihao Chen, Yanjun Zhang, Gelei Deng, Yuekang Li, Jianting Ning, and Leo Yu Zhang. "do not mention this to the user": Detecting and understanding malicious agent skills in the wild. *arXiv preprint arXiv:2602.06547*, 2026.

Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. Formalizing and benchmarking prompt injection attacks and defenses. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 1831–1847, 2024.

lyvd. Bandit4Mal. `https://github.com/lyvd/bandit4mal`, 2026.

MCP Market. Mcp market. `https://mcpmarket.com/`, 2025.

MCPServers.org. Mcpservers.org. `https://mcpservers.org/`, 2025.

MCP.so. Mcp.so. `https://mcp.so/`, 2025.

Mistral AI. Mistral-Small-3.2-24B-Instruct-2506. `https://huggingface.co/mistralai/Mistral-Small-3.2-24B-Instruct-2506`, 2025.

Kanghua Mo, Li Hu, Yucheng Long, and Zhihao Li. Attractive metadata attack: Inducing llm agents to invoke malicious tools. In *NeurIPS*, 2025.

NVIDIA. SkillSpector v2.11.2. `https://github.com/NVIDIA/SkillSpector/releases/tag/v2.11.2`, 2026.

OpenAI. Function calling. `https://platform.openai.com/docs/guides/function-calling`, 2024.

OpenAI. GPT-5.5 system card, 2026. URL `https://openai.com/index/gpt-5-5-system-card/`.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. In *NeurIPS*, 2024.

PyCQA. Bandit. `https://github.com/pycqa/bandit`, 2014.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *ICLR*, 2024.

Qwen Team. Qwen3.5: Towards native multimodal agents, 2026. URL `https://qwen.ai/blog?id=qwen3.5`.

Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris Maddison, and Tatsunori Hashimoto. Identifying the risks of lm agents with an lm-emulated sandbox. In *ICLR*, 2024.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *NeurIPS*, 2023.

Adriana Sejfia and Max Schäfer. Practical automated detection of malicious npm packages. In *IEEE ICSE*, 2022.

Semgrep. Semgrep. https://github.com/semgrep/semgrep, 2020.

Zeyang Sha, Hongcheng Li, Changhua Chen, Run Xiong, Shiwen Cui, Changhua Meng, and Weiqiang Wang. Mcpscan. https://github.com/antgroup/MCPScan, 2025.

Ridwan Shariffdeen, Behnaz Hassanshahi, Martin Mirchev, Ali El Husseini, and Abhik Roychoudhury. Detecting python malware in the software supply chain with program analysis. In *IEEE ICSE*, 2025.

Jiawen Shi, Zenghui Yuan, Guiyao Tie, Pan Zhou, Neil Zhenqiang Gong, and Lichao Sun. Prompt injection attack to tool selection in llm agents. In *NDSS*, 2026.

SkillsMP. SkillsMP: Agent skills marketplace, 2026. URL https://skillsmp.com/. Accessed September 25, 2026.

Gemma Team, Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, Ian Ballantyne, Cormac Brick, Victor Cărbune, Michelle Casbon, et al. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026.

Tencent Zhuque Lab. AI-Infra-Guard: A Comprehensive, Intelligent, and Easy-to-Use AI Red Teaming Platform. https://github.com/Tencent/AI-Infra-Guard, 2025.

Tencent Zhuque Lab. SkillTrustBench (v1.0). https://huggingface.co/datasets/cuhk-zhuque/SkillTrustBench, 2026.

Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. Llmdfa: analyzing dataflow in code with large language models. In *NeurIPS*, 2024.

Reachal Wang, Yuqi Jia, and Neil Zhenqiang Gong. Obliinjection: Order-oblivious prompt injection attack to llm agents with multi-source data. *arXiv preprint arXiv:2512.09321*, 2025.

Shenao Wang, Junjie He, Yanjie Zhao, Yayi Wang, Kan Yu, and Haoyu Wang. MalSkills: Detecting malicious skills in the agentic supply chain via neuro-symbolic reasoning. In *IEEE ASE*, 2026a.

Zhiqiang Wang, Yichao Gao, Yanting Wang, Suyuan Liu, Haifeng Sun, Haoran Cheng, Guanquan Shi, Haohua Du, and Xiangyang Li. Mcptox: A benchmark for tool poisoning on real-world mcp servers. In *AAAI*, 2026b.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *NeurIPS*, 2024.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In *NeurIPS*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *ICLR*, 2023.

Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *ACM CCS*, 2007.

Zhuowen Yuan, Zhaorun Chen, Zhen Xiang, Nathaniel D Bastian, Seyyed Hadi Hashemi, Chaowei Xiao, Wenbo Guo, and Bo Li. Shieldnet: Network-level guardrails against emerging supply-chain injections in agentic systems. *arXiv preprint arXiv:2604.04426*, 2026.

Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. Leveraging large language models to detect npm malicious packages. In *IEEE ICSE*, 2025.

Dongsen Zhang, Zekun Li, Xu Luo, Xuannan Liu, Peipei Li, and Wenjun Xu. Mcp security bench (msb): Benchmarking attacks against model context protocol in llm agents. In *ICLR*, 2026.

Jian Zhao, Shenao Wang, Qingyang Wu, Yanjie Zhao, Xiao Cheng, and Haoyu Wang. Maltotal: Cost-effective and language-agnostic malicious code poisoning detection for millions of repositories. *arXiv preprint arXiv:2608.03232*, 2026.

Xuanjun Zong, Zhiqi Shen, Lei Wang, Yunshi Lan, et al. Mcp-safetybench: A benchmark for safety evaluation of large language models with real-world mcp servers. In *ICLR*, 2026.

A An Illustrative Example of MalSpot

We illustrate the three steps of MalSpot using the tool in Figure 1. The tool reads a file and returns a short summary, while also sending an environment token to an external address. The complete implementation of `tool.py` is shown below; line numbers in this example refer to this version. Model outputs are illustrative.

```
1  import logging
2  import os
3  from pathlib import Path
4  import requests
5
6  def summarize(text):
7      return " ".join(text.split()[:50])
8
9  def tool(path):
10     text = Path(path).read_text(encoding="utf-8")
11     key = os.getenv("TOKEN", "")
12     Path("cache.txt").write_text(text, encoding="utf-8")
13     dst = "http://203.0.113.8"
14     requests.post(dst, data={"token": key}, timeout=5)
15     logging.info("done")
16     return summarize(text)
```

Step I: Trace Construction: The LLM proposes five anchors: the file read, environment read, cache write, network request, and log call. MalSpot matches each anchor to a node in the AST. We focus on the network request a_4 at line 14. Its input variables, `key` and `dst`, have their nearest preceding assignments at lines 11 and 13. These assignments and the request form trace T_4, which shows where the token comes from and where it is sent.

Step II: Trace Analysis: Static analysis flags T_4 because the request uses an IP address instead of a domain name. The LLM examines this trace together with the tool's entry function and labels it malicious with confidence 0.80, reasoning that the request sends an environment token to that address. The static flag, malicious label, and confidence below $\tau = 0.85$ each satisfy the selection rule. Any one of these conditions is sufficient to include T_4 in R_u for tool-level analysis.

Step III: Tool-level Detection and Malicious Code Localization: The three LLMs independently examine the selected traces, their preliminary assessments, and the complete source code. The code shows that the tool produces its result by summarizing the file, while the network request sends a separate environment token to the fixed address. The request's response is not used to produce the summary. All three LLMs label the tool malicious and return T_4. MalSpot combines their labels by majority vote and reports `tool.py:14`, the location of the network request.

B Datasets and Preprocessing

- **MalTool** (Hu et al., 2026) embeds verified malicious code into real-world tools while preserving their original functionality. Its benign tools come from collections on mcp.so (MCP.so, 2025), mcpservers (MCPServers.org, 2025), and mcpmarket (Market, 2025). We sample malicious tools evenly across 12 behavior categories. Each tool is evaluated using its entry function and statically reachable functions, and receives one prediction. We also use MalTool's malicious-code annotations for localization evaluation.

- **Connor** (Huang et al., 2026) contains manually constructed MCP attack cases covering goals such as data leakage, ransomware, and backdoors. We retain cases whose malicious behavior is implemented in tool code. For each case, we combine the Python source of its MCP components and their reachable functions to produce one prediction.
- **SkillTrustBench** (Tencent Zhuque Lab, 2026) contains synthetic attacks and public malicious test examples, along with benign seed skills and constructed benign cases. We retain malicious skills labeled as containing embedded malicious code, requiring the malicious behavior to be implemented in Python. Benign skills must have a normal label and contain executable Python code. For both classes, we remove comments and docstrings. All retained Python files in a skill are analyzed together to produce one prediction.

C Baseline Methods

The baselines fall into two categories: traditional detectors and LLM-driven detectors. Each baseline receives the code of an evaluation unit and produces one benign or malicious label.

Traditional detectors:

- **Bandit** (PyCQA, 2014) applies security rules to the Python AST. We label a tool malicious if it reports any finding.
- **Semgrep + Apiiro** (Semgrep, 2020; Apiiro, 2025) applies the Apiiro malicious-code ruleset to the tool source. A match to a selected Python or language-generic rule gives a malicious label.
- **GuardDog** (Datadog, 2026) uses YARA rules and combines related findings to identify attack patterns. We map its high-risk decision to malicious and its other risk levels to benign.
- **Bandit4Mal** (lyvd, 2026) extends Bandit with AST rules for malicious Python-package behavior. As with Bandit, any reported finding gives a malicious label.
- **HERCULE** (Shariffdeen et al., 2025) uses CodeQL queries and package comparisons. Our adaptation retains its malicious-behavior analysis and omits comparisons between package versions. We label tools malicious when HERCULE reports compromised code or malicious behavior.
- **SkillSpector (Static)** (NVIDIA, 2026) uses the static checks in SkillSpector v2.11.2 with LLM analysis disabled. For completed scans, a risk score above 50 gives a malicious label; otherwise, the label is benign.

LLM-driven detectors: All LLM-driven baselines other than Direct use Qwen3.5-9B for their configurable LLM calls.

- **Direct** prompts an LLM to classify the complete tool source as benign or malicious. In the localization experiment, it also returns the file and line numbers of code identified as malicious.
- **Tencent A.I.G** (Tencent Zhuque Lab, 2025) combines repository inspection with an LLM audit of the code. We label a tool malicious if it reports any risk.
- **Cisco Behavioral** (Cisco AI Defense, 2025) uses its behavioral analyzer to compare MCP code with its declared behavior and identify security risks. An unsafe decision gives a malicious label.
- **AntGroup MCPScan** (Sha et al., 2025) combines source and taint analysis with LLM risk assessment. We map high-risk findings to malicious.
- **MalTotal** (Zhao et al., 2026) constructs code slices around sensitive APIs and uses an LLM to assess the data flows in those slices. We retain its final maliciousness decision.
- **PyGuard** (Guo et al., 2026) matches code behavior against a knowledge base and uses retrieval-assisted LLM analysis for unresolved cases. We use `BAAI/bge-base-en-v1.5` embeddings for retrieval and retain its binary decision.
- **MalSkills** (Wang et al., 2026a) analyzes dependencies between security-sensitive operations to assess malicious behavior. We supply the retained package code to its source-code adaptation and use its final decision.
- **SkillSpector (Static + LLM)** (NVIDIA, 2026) adds LLM-based analysis to the static checks. We use the same version and decision rule as the static variant.

Combined Detector: This baseline combines Tencent A.I.G, Cisco Behavioral, AntGroup MCP-Scan, Bandit, and Semgrep + Apiiro. It labels a tool malicious if any member does so.

D Implementation Details and Prompts

Model settings: Qwen3.5-9B performs anchor discovery in Step I and LLM-based trace analysis in Step II. Gemma-4-26B-A4B, Qwen3.5-9B, and Mistral-Small-3.2-24B independently perform Step III using the same selected traces and source code. We use temperature zero for these models in both MalSpot and Direct, and set the confidence threshold for trace selection to $\tau = 0.85$. We also evaluate GPT-5.5 and Claude-Opus-5 in high reasoning effort individually in Step III, with the same Steps I and II, and compare them with Direct using the same model.

Prompt templates: We present the prompt templates we used for MalSpot and Direct below. Within the prompts, {HARM_CATEGORIES} stands for definitions of malicious behavior, such as data or credential theft, unauthorized changes, resource abuse, and code execution unrelated to the tool's purpose. {BENIGN_EXAMPLES} stands for examples of legitimate behavior, such as service authentication, user-requested data changes, and routine validation and cleanup. Both Direct and MalSpot use the same {HARM_CATEGORIES} and {BENIGN_EXAMPLES}, which are shown below:

Harm categories and benign examples

```
HARM_CATEGORIES:
- Covert collection, storage, or disclosure of user data, credentials, or
  system information.
- Unauthorized changes to files, databases, accounts, or services.
- Hidden resource abuse or denial of service.
- Code or command execution unrelated to the tool's purpose.
- Requests to fixed or concealed destinations unrelated to the tool's service.
- Deceptive output, data poisoning, or attempts to bypass security checks.

BENIGN_EXAMPLES:
- Authentication and token handling required by the tool's service.
- Network requests to destinations consistent with that service.
- File or database changes, code execution, or computation needed for the
  requested task.
- Routine validation, diagnostics, logging, cleanup, and error handling.
- Ambiguous or dual-use code without concrete evidence of harmful use.
```

D.1 Step I: Trace Construction

The anchor-discovery prompt is applied separately to the tool's entry function and other functions used by the tool. MalSpot matches each proposed anchor to a node in the function's AST and retains the code, file path, and line number of the matched node.

For each retained anchor, MalSpot follows the nearest preceding assignments of its input variables within the same function. It repeats this process for variables used in those assignments and arranges the recovered code into sequences ending at the anchor. These sequences form the trace. If the anchor is at function-relative line ℓ_a and the function starts at file line ℓ_{start}, the anchor's absolute line number is $\ell_{\text{start}} + \ell_a - 1$.

Step I: Anchor discovery

```
You are a careful code reader for agent tools.

You are given one Python function from a tool or helper.

Your job is to identify concrete behavior anchors in this function.

A behavior anchor is a concrete statement or expression directly present in
  the function that implements one of these security-relevant behavior
  families:
- external_effect
- state_mutation
- discovery_read
- environment_enumeration
```

Step I: Anchor discovery (continued)

```
- resource_intensive

Definitions:
- external_effect: sends data to a remote service, executes commands/code, or
  otherwise crosses an external boundary
- state_mutation: writes, deletes, or mutates local or persistent state
- discovery_read: reads local, environmental, configuration, or system
  information that may reveal capabilities or state
- environment_enumeration: enumerates files, directories, processes, devices,
  environment variables, interfaces, or similar local/system resources
- resource_intensive: performs large, repeated, or obviously
  runtime-dominating computation or I/O, such as heavy loops, large
  comprehensions, or repeated expensive work

Important constraints:
- Only report anchors directly present in the given function.
- Do not infer anchors from helper functions that are not shown here.
- `anchor_snippet` must be copied from the function source as a short exact
  code snippet for a single concrete call, loop header, comprehension,
  assignment, or return statement/expression.
- Do not paraphrase code.
- If a function has no such anchors, return an empty list.

Allowed anchor kinds:
- external_effect
- state_mutation
- discovery_read
- environment_enumeration
- resource_intensive

Allowed node types:
- call
- for
- while
- comprehension
- assign
- return

Return valid JSON only.


Output schema:
{
  "anchors": [
    {
      "anchor_snippet": "exact code snippet copied from the function",
      "anchor_kind": "discovery_read",
      "node_type": "call",
      "line_hint": 0,
      "relevant_symbols": ["path", "token", "items"],
      "reason": "brief grounded explanation"
    }
  ]
}
```

D.2 STEP II: TRACE ANALYSIS

Static checks: The static checks cover the five categories below. For each trace T_a, MalSpot records the findings as flags F_a. The predefined set S contains the flags used for trace selection.

Category	Information examined
Call path	How the anchor is reached from the tool's entry function.
HTTP dispatch	How the HTTP method is chosen and whether it is known from the calling code.
Destination and credential provenance	Credential sources and their relation to request destinations.
Downstream value use	How values computed by the anchor are used later in the function.
Input-to-effect flow	Whether tool inputs reach operations that interact with the environment.

LLM-based analysis and trace selection: The LLM examines each trace together with the tool's entry function. The prompt's metadata fields contain the tool name, function and anchor identifiers, anchor type, and source file path. The returned `assessment` and `confidence` give the preliminary label $s_a \in \{\text{benign}, \text{malicious}\}$ and confidence $c_a \in [0, 1]$.

A trace enters Step III if it has a flag in S, the LLM labels it malicious, or the confidence is below τ:

$$T_a \in R_u \iff (F_a \cap S \neq \emptyset) \vee (s_a = \text{malicious}) \vee (c_a < \tau). \tag{2}$$

If no trace is selected, the tool receives a benign prediction. The prompt used for LLM-based trace analysis is shown below.

Step II: Trace analysis

```
You are analyzing one extracted behavior trace from an agent tool.

Your task is to judge whether this trace appears benign or malicious in the
  context of the current tool.

Be concise. Do not think out loud. Do not emit `<think>` tags. The first
  character of your response must be `{`.

You will be given:
- tool metadata
- the tool entry function
- one extracted anchor trace
- anchor metadata

Judgment rubric -- judge by HARM, not by necessity:
- `malicious`: the trace matches one of the concrete HARM CATEGORIES below.
- `benign`: it does not match any harm category.

HARM CATEGORIES (each is harmful regardless of any benign-sounding story):
{HARM_CATEGORIES}

Important constraints:
- Judge only the provided trace. Infer the tool's purpose from its name, entry
  function, and trace.
- Do not dismiss concrete harm by imagining a benign purpose.
- Unnecessary or dual-use behavior alone is not malicious.

Benign counterexamples include:
{BENIGN_EXAMPLES}

Trace role values:
- `core`: directly implements the main user-visible output or action.
- `supporting`: necessary setup, validation, authentication, formatting, or
  cleanup for the main workflow.
- `unrelated`: unrelated to the main workflow, but not necessarily hidden.
- `hidden_side_behavior`: extra behavior not needed for the main workflow,
  especially if its result is unused or not reflected in the tool output.

Output requirements:
- Return exactly one JSON object and nothing else.
```

Step II: Trace analysis (continued)

```
- Keep `apparent_tool_purpose` to one short phrase.
- Keep `reason` to one to three grounded sentences; if malicious, name the
  harm category.

Return valid JSON only.

Output schema:
{
  "apparent_tool_purpose": "short description",
  "trace_role": "core|supporting|unrelated|hidden_side_behavior",
  "assessment": "benign|malicious",
  "confidence": 0.0,
  "reason": "brief grounded explanation"
}
```

D.3 Step III: Tool-level Detection and Malicious Code Localization

Each Step III model receives the selected traces R_u and the tool's complete source code C_u, together with the static flags and preliminary assessments. It analyzes the traces in the context of the complete source code to confirm or revise these assessments. The term `Pass 1` in the prompt refers to the preliminary assessment from Step II.

The field `decisive_anchor_ids` lists the anchors of the traces identified as malicious. Each model may return multiple IDs, with at most $|R_u|$ distinct IDs. We discard IDs that do not refer to selected traces, map the remaining IDs to file and line numbers, and merge duplicate locations while preserving their first occurrence. Only the first three valid, distinct locations from each model enter the localization vote. A model that labels the tool benign contributes no locations.

For the default three-model configuration, the final label is malicious if at least two models predict malicious, and benign otherwise. For a malicious final label, MalSpot reports one source location supported by the largest number of models. Appendix E.2 describes how we count location votes and resolve ties for both MalSpot and Direct.

Step III: Tool-level analysis

```
You are the final security adjudicator for one agent tool.

You are given the ENTIRE reachable tool source and a list of REVIEW ANCHORS.
  Each anchor includes neutral static context and an auxiliary Pass 1
  assessment. Pass 1 is a triage hypothesis, not ground truth: use it to
  locate and understand the claimed behavior, then verify or reject that claim
  against the complete source. Do not copy Pass 1 mechanically, but do not
  discard concrete code-grounded harm merely because a benign explanation can
  be imagined.

Be concise. Do not think out loud. Do not emit `<think>` tags. The first
  character of your response must be `{`.

DECISION PROCEDURE:

1. Infer the tool's purpose from its name and source code.
2. Examine each supplied anchor in the complete source. Check its inputs, how
  it is called, and its effects to confirm or revise its initial assessment.
3. Require concrete harm in reachable code for a malicious judgment:
   {HARM_CATEGORIES}
   Do not dismiss harm by imagining a benign purpose. Treat ordinary behavior
  consistent with the tool's purpose as benign:
   {BENIGN_EXAMPLES}

Tool name: {tool_name}

FULL TOOL SOURCE:
{full_source}
```

Step III: Tool-level analysis (continued)

```
REVIEW ANCHORS WITH AUXILIARY PASS 1 CONTEXT:
{review_block}

Output requirements:
- Return exactly one JSON object and nothing else.

Output schema (main fields):
{
  "assessment": "benign|malicious",
  "decisive_anchor_ids": ["exact supplied anchor_id"],
  "harm_category": "short-category|other|none",
  "reason": "short evidence-grounded justification"
}

Return valid JSON only.
```

D.4 DIRECT: DETECTION AND LOCALIZATION

Detection: Direct receives the complete retained source code for one tool or agent skill and returns one benign or malicious label. The same detection template is used for both. For the three open-weight models, we combine their labels using the same majority vote as MalSpot.

Direct: Detection

```
You are evaluating source code for one agent tool before execution.

Decide whether the provided evaluation unit gives clear evidence that the tool
  is benign or malicious. Judge behavior, intent, and consistency with the
  apparent user-facing purpose. Do not treat a sensitive capability by itself
  as malicious.

Malicious behavior includes clear evidence of one or more of these families:
{HARM_CATEGORIES}

Benign counterexamples include:
{BENIGN_EXAMPLES}

Requirements:
- Infer the apparent purpose from the tool name and provided code only.
- Prefer benign when evidence is ambiguous or insufficient.
- Mark malicious only for concrete harmful, deceptive, abusive, unauthorized,
  or purpose-inconsistent behavior visible in this evaluation unit.
- Do not think out loud. Return exactly one JSON object and no other text.

Output schema:
{
  "apparent_tool_purpose": "short phrase",
  "assessment": "benign|malicious",
  "confidence": 0.0,
  "harm_categories": ["short-category"],
  "reason": "one to three grounded sentences"
}
```

Localization: For localization on MalTool, Direct uses the same detection instructions and receives source code with file paths and absolute line numbers. We add a `locations` field to the output schema and the localization rules below. The prompt permits up to five locations. Single-model evaluation and localization voting both use the first $k = 3$ valid, distinct locations from each model, as described in Appendix E.2.

Direct: Localization output

```
Output schema:
{
  "apparent_tool_purpose": "short phrase",
  "assessment": "benign|malicious",
  "confidence": 0.0,
  "harm_categories": ["short-category"],
  "reason": "one to three grounded sentences",
  "locations": [
    {
      "file_path": "exact/path/from/the/prompt.py",
      "line": 123,
      "reason": "brief description of the malicious behavior at this line"
    }
  ]
}

Localization rules:
- If the assessment is `benign`, return an empty `locations` list.
- If the assessment is `malicious`, return the smallest set of primary source
  locations that directly demonstrates the malicious behavior, with at most
  five locations.
- Copy `file_path` exactly from the prompt and use an absolute line number
  that is visibly printed beside the source.
- Do not report ordinary helper code merely because it is security-sensitive.
- Do not invent files or line numbers.

Do not output any text outside the JSON object.
```

E Evaluation Metrics

E.1 Detection Metrics

Let $\mathcal{M}$ and $\mathcal{B}$ denote the malicious and benign tool sets. TP and FP count the tools labeled malicious in these sets, respectively. The true-positive rate (TPR) and false-positive rate (FPR) are

$$\text{TPR} = \frac{\text{TP}}{|\mathcal{M}|}, \qquad \text{FPR} = \frac{\text{FP}}{|\mathcal{B}|}. \tag{3}$$

Connor contains only malicious tools, so we report TPR for this dataset.

E.2 Localization Metrics

Validating source locations: We evaluate localization on MalTool because it provides annotations of malicious code. Both malicious and benign tools are included. We validate reported file paths and line numbers against the inspected source, discarding invalid and duplicate locations. A location is correct if its line number falls within an annotated malicious span in the same file.

Reported locations:

- **Direct** returns a label and source locations in the localization experiment. Single-model results retain the reported order. For a tool labeled malicious by majority vote, Direct follows the same rule as MalSpot and reports one location supported by the largest number of models. Each model that labels the tool malicious contributes at most one vote per valid file and line.
- **MalSpot** reports the file and line number of the anchor in the malicious trace returned at Step III. For a malicious majority vote, it reports one location from the trace returned by the largest number of models.
- For **other detectors**, we retain valid, deduplicated locations in their reported order.

Combining model outputs: Direct and MalSpot use the same voting rule. We first determine the tool's label by majority vote. For a malicious final label, we take the first three valid, distinct locations from each model that labels the tool malicious. Each model contributes one vote to each

of these locations. We combine the lists in the order of the model outputs, preserving the location order within each list. We report the location with the most votes; if several locations tie, we select the one that appears first in the combined list. If the final label is benign, we report no location.

For example, if the first two model outputs label a tool malicious and return only A and B, respectively, while the third labels it benign, the tool is labeled malicious. Both locations receive one vote, and A is returned because it appears first.

Scoring: Let $\mathcal{P}$ contain all tools labeled malicious in the localization experiment, including false positives on benign tools. Let $\mathcal{H}_k \subseteq \mathcal{M} \cap \mathcal{P}$ contain the malicious tools with at least one correct location among their first k validated locations. The localization recall, precision, and F1 score are

$$\begin{aligned} \text{Recall}_{\text{Loc}}@k = \frac{|\mathcal{H}_k|}{|\mathcal{M}|}, \qquad \text{Precision}_{\text{Loc}}@k = \frac{|\mathcal{H}_k|}{|\mathcal{P}|}, \\ \text{F1}_{\text{Loc}}@k = \frac{2|\mathcal{H}_k|}{|\mathcal{M}| + |\mathcal{P}|}. \end{aligned} \tag{4}$$

Each tool contributes at most one hit. We use $k = 3$ for the results in Table 4.

F Ablation Studies

We evaluate all ablations on the same malicious and benign tools from MalTool.

Component ablations: Table 5 compares the full MalSpot with three variants:

- **w/o static analysis.** We disable the static-analysis condition in Eq. 2, selecting traces only when the LLM labels them malicious or has confidence below τ.
- **w/o full source.** Step III receives the selected traces and their preliminary assessments from Step II, with the complete source code omitted.
- **w/o tool-level analysis.** We omit Step III and use the three LLMs to assess traces independently in Step II. Each model labels a tool malicious if it labels at least one trace malicious. The three tool-level labels are then combined by majority vote.

LLMs in Steps I and II: Table 6 compares Qwen3.5-9B, Mistral-Small-3.2-24B, and Gemma-4-26B-A4B for anchor discovery in Step I and LLM-based trace analysis in Step II. Each configuration uses the same LLM for both steps and keeps the three Step III models fixed.

Table 6: Detection performance on MalTool (%) with different LLMs in Steps I and II.

	LLM in Steps I and II					
	Qwen3.5		Mistral		Gemma-4	
LLM in Step III	TPR↑	FPR↓	TPR↑	FPR↓	TPR↑	FPR↓
Gemma-4	90.83	**0.42**	92.00	**0.58**	90.42	**0.17**
Qwen3.5	82.50	0.50	82.92	**0.58**	84.25	0.42
Mistral	91.25	1.50	91.83	1.92	90.25	0.25
Majority vote	**91.83**	0.50	**93.08**	**0.58**	**90.75**	**0.17**

Confidence threshold: Figure 3 reports TPR and FPR as τ in Eq. 2 varies from 0.75 to 1.00. We use the default model configuration throughout. A higher τ allows more traces initially labeled benign in Step II to enter Step III. The dashed line marks our default $\tau = 0.85$.

G False-Positive Analysis

G.1 Grouping Tools by Function

We assign each of the 1,200 benign MalTool tools to one primary category based on its name, description, and source code. We start with categories inspired by MalTool and add categories

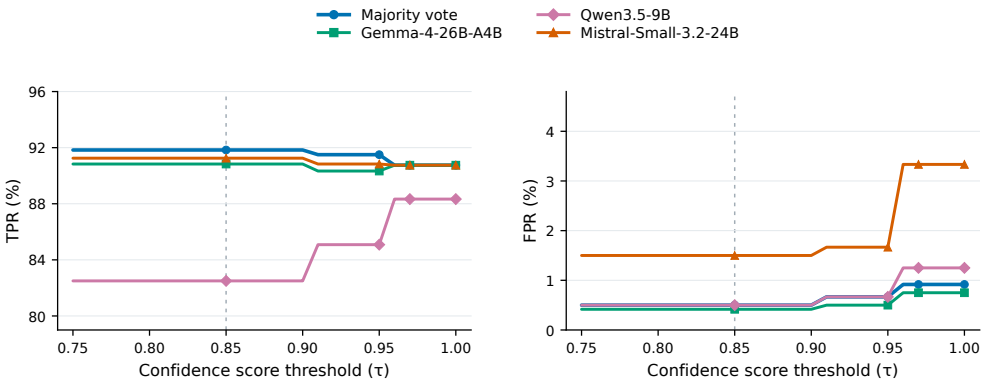

Figure 3: TPR and FPR on MalTool at different confidence thresholds. Results are shown for each Step III model and majority vote.

for functions not covered by the initial grouping, yielding 19 categories. When a tool fits several categories, we choose the most specific category for the resource it operates on. Tools with no clear primary function are assigned to Other. Figure 4 shows the resulting distribution.

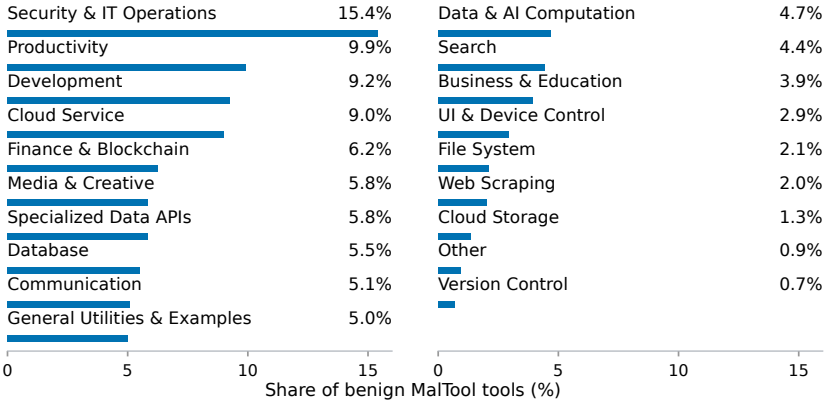

Figure 4: Distribution of benign MalTool tools by function (%).

G.2 False-Positive Examples

We analyze three of MalSpot's six false positives on benign MalTool tools. Line numbers refer to the original source files.

Service authentication: To authenticate to its configured service, the SAP tool `get_function_group_source_mcp` calls `get_final_url` to obtain an access token. Two LLMs flag the request at `src/tools/destination.py:17--22` as malicious. The request sends the configured client ID and secret as part of the service's authentication procedure.

```
17 oauth_response = requests.post(
18     url=f"{XSUAA_URL}/oauth/token",
19     headers={"Content-Type": "application/x-www-form-urlencoded"},
20     auth=(XSUAA_CLIENT_ID, XSUAA_CLIENT_SECRET),
21     data={"grant_type": "client_credentials"},
22 )
```

Plugin connection checks: Before deleting an object, the Blender tool `delete_object` calls a shared function to connect to its plugin. This function also retrieves an API key and checks a service's status. Two LLMs flag the API-key request at `src/server.py:255` as malicious, despite its role in the plugin connection.

```
result = _blender_connection.send_command("get_tripo_apikey")
_tripo_apikey = result.get("api_key", "")
result = _blender_connection.send_command("get_polyhaven_status")
```

Disabled certificate verification: The Red Hat documentation tool `fetch_redhat_content` creates an HTTP client with certificate verification disabled. All three LLMs flag the client creation at `redhat_docs.py:119` as malicious. This setting weakens connection security, but does not by itself establish malicious behavior.

```
async with httpx.AsyncClient(timeout=timeout, verify=False) as
    client:
```