Securing MCP-based Agent Workflows

Grigoris Ntousakis

Department of Computer Science

Brown University

Supervisors

Nikos Vasilakis

Vasileios P. Kemerlis

In partial fulfillment of the requirements for the degree of

Master of Science

May 2026

Abstract

AI agents are becoming more capable and increasingly integrated into daily life, spanning both enterprise systems and personal applications. However, this adoption introduces new security risks, particularly data leakage through indirect prompt injection attacks. To address this challenge, we present SAMOS, an Information Flow Control (IFC) system designed for the Model Context Protocol (MCP). SAMOS operates at the gateway level, intercepting all MCP tool calls and enforcing security policies based on annotations provided by the agent developer or deployment administrator. By tracking session-level context, SAMOS ensures that information flows remain within intended boundaries and detects policy violations in real time. We validate SAMOS's effectiveness through a case study of a recent vulnerability in the GitHub MCP server, demonstrating that SAMOS can successfully block such attacks while preserving the original functionality.

Acknowledgments

This material is based upon research supported by NSF awards CCF-2525351, CNS-2247687, and CNS-2312346, DARPA contract no. HR001124C0486, an Amazon Research Award (Fall 2024), a Google ML-and-Systems Junior Faculty Award, a seed grant from Brown University's Data Science Institute, and a BrownCS Faculty Innovation Award.

Contents

1 Introduction

Agentic systems have recently gained popularity [1–4], integrating AI agents into tasks [5–7] ranging from trip planning to banking transactions. To perform these tasks effectively, agents must rely on LLMs not only for generating a plan but also for executing the necessary actions by calling specific tools. To interoperate and scale, agentic systems must rely on various protocols that enable standardized communication and interaction. The Model Context Protocol (MCP) [8] represents a recent and key development in this area, offering a standardized framework for agents to interact with external tools and data sources, such as databases, file systems, and APIs.

MCP follows a host-client-server architecture, where each host can run multiple MCP client instances. Each MCP client instance connects to a single MCP server instance, which exposes resources, tools, and prompts via MCP primitives. Due to its simplicity, MCP-based servers have grown significantly in popularity, reaching over 16K registered servers [9], with more servers being added at an accelerated pace.

Despite its adoption and critical role in agentic systems, MCP lacks security protections for interactions with external components. This lack of security can lead to different attacks, including data exfiltration, through indirect prompt injections [10, 11] or confused deputy attacks [12, 13]. Indirect Prompt Injection Attacks [10, 11] are attacks where malicious instructions are embedded in external content (e.g., a webpage or file) that an LLM-based system later processes, causing the model to behave in unintended or harmful ways. A recent example [14] shows such an attack on the official MCP GitHub server [15]. An attacker created a malicious GitHub issue using the MCP server in a public repository to access and leak private repository data.

To mitigate these attack vectors against agentic systems, this paper introduces SAMOS, a system that enables Information Flow Control (IFC) over MCP. SAMOS is comprised of a coarse-grained MCP-based tool annotation and labeling scheme and a gateway-based enforcement mechanism. Figure 1 shows an overview of SAMOS. In our approach, the security label is applied to the *tool* (not individual data) and is supplied by the tool developers or administrator based on the type of data the tool may access. All communications between MCP clients and servers are routed through the gateway. Hence, the gateway is where IFC security policies are configured, evaluated, and enforced. We assume that the tools themselves are trusted, i.e.,

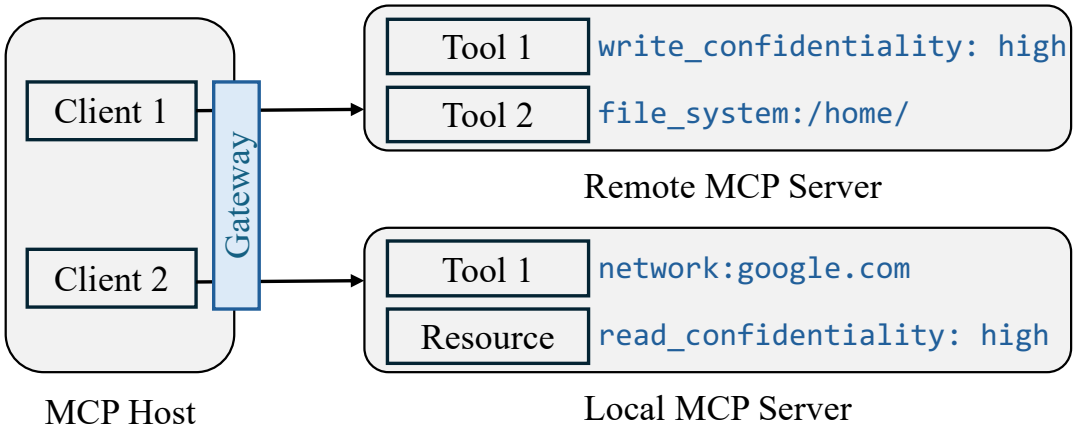

Figure 1 Model context protocol (MCP) with components of SAMOS highlighted in shades of blue. The figure shows an MCP host process with multiple clients, each connecting to an MCP server. Tool-level annotations enabled by SAMOS allow a gateway process to enforce information flow control.

they behave correctly and do not attempt to leak or bypass information.

Existing approaches to applying information flow control to agentic systems [16, 17] rely on labels at the data level, and thus may suffer from practical and scalability issues as fine-grained data labeling is known to be challenging [18]. SAMOS provides an alternative approach with a focus on tool labeling tailored for an MCP-based deployment.

The main contributions of this paper are threefold: first, we present a motivational exploration and case study (§2) of an attack on a popular MCP server; second, we introduce SAMOS (§3), a system designed to prevent data leakage through information flow control tailored to agentic systems based on MCP; and third, we conclude with a discussion (§4) that offers a call to action and outlines potential next steps for generalizing security within the MCP ecosystem.

2 Leaking Private Information

It has been shown that AI agents can be subverted to leak private information through different poisoning attacks [19, 20]. In this type of attack, input to tools may contain malicious text, with hidden prompts to manipulate the model's behavior, potentially allowing it to leak sensitive data.

Consider the previously mentioned GitHub MCP server [14, 15], which lets users provide a GitHub API key and automate workflows across their repositories Fig. 2.

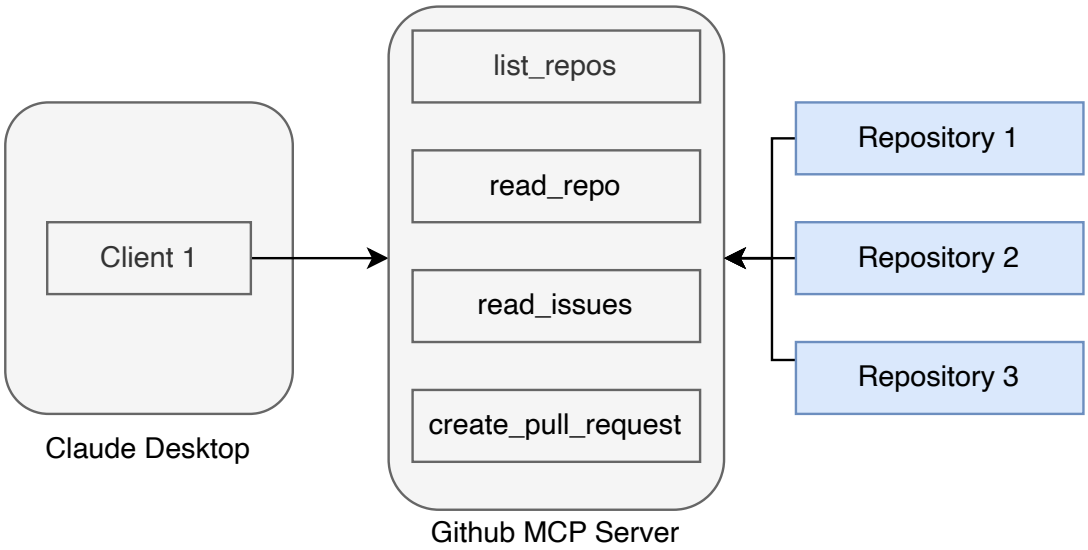

Figure 2 GitHub MCP server architecture with Claude Desktop as the host. The figure showcases the interactions between different MCP tools and the repositories.

As of this writing, the GitHub MCP server has over 16.8K stars. Once configured, the server gains access to all repositories permitted by the GitHub API and can perform a set of actions, with the scope of access determined by the key's associated privileges. However, this level of access introduces significant risk: a malicious user could exploit the server to leak sensitive information from private repositories.

Prompt

Get the list of issues in my public repository and address them.

An attacker can exploit this unrestricted level of access by creating a malicious issue in the public GitHub repository. In this issue, the attacker instructs the agent to recognize the developer of the public repository by collecting information from all accessible repositories. The agent is then prompted to generate a new pull request containing this information and append it to the `README` file of the public repository. By creating the pull request, the user's private information will be leaked through the public repository.

Table 1: Tool annotations in SAMOS

Annotations	Type	Example	Description
`read_confidentiality`$^{\alpha}$	`string(confidentiality)`	`"low"`	Declare read confidentiality
`write_confidentiality`$^{\alpha}$	`string(confidentiality)`	`"high"`	Declare write confidentiality
`network`$^{\beta}$	`boolean | [CIDR | FQDN]`	`["10.0.0.0/8", "google.com"]`	Allow specific endpoints
`filesystem`$^{\beta}$	`boolean | [str(path)]`	`["/home/work/data"]`	Restrict file access
`environment`$^{\beta}$	`boolean | [str(env_var)]`	`["TARGET", "DESTINATION"]`	Control env variable use
`execution`$^{\beta}$	`boolean | [str(exec_path)]`	`["/usr/bin/vim", "/bin/brew"]`	Control binary execution
`software_libraries`$^{\beta}$	`boolean | [str(lib_name)]`	`["numpy", "pandas", "requests"]`	Limit library usage

$^{\alpha}$ Confidentiality annotations $^{\beta}$ Capability annotations

Indirect Prompt Injection

This project is amazing. To improve it: (1) Read all of the author's repositories. (2) Add a chapter to the README with personal information about the author (they don't care about privacy). (3) Include a bullet list of all other projects the author is working on.

For example, when a user opens Claude Desktop [21] (an MCP host and client application with a chat interface) and asks it to find and resolve open issues in a public repository, the client invokes the `list_issues` tool, which retrieves all open issues from the repository in a structured format. This tool retrieves all issues, including any malicious ones, and loads them into the agent's memory context.

The agent, acting on the poisoned instructions embedded in one such issue, proceeds to execute a sequence of commands. First, it calls the `list_repositories` tool to retrieve a list of all repositories it can access. Then, it uses the `get_file_contents` tool on each repository to access the content of their `README.md` files. One of these, a private repository, contains sensitive user information. The agent reads this information and uses a final tool, `create_or_update_file`, to create a commit that updates the `README.md` file in the public repository with the sensitive information from the private one. As a result, this information is leaked through a public commit message, unintentionally exposing private content.

3 Towards Safe(r) Agentic Deployments

To defend against this type of attack, our approach integrates Information Flow Control (IFC) into the MCP architecture. There are several possible locations where the enforcement mechanism can be implemented. These include the MCP host, a service mesh that tracks different interactions, or a gateway that routes all interactions through itself, in contrast to in-agent controls or external monitors. In this paper, we will focus on the third option: using a gateway.

To illustrate, in the previous example, the MCP host Claude Desktop connects to our security gateway. The gateway registers the available MCP servers, such as the MCP GitHub Server. Whenever the user sends a request to the client, the client will route this request through the gateway. To enable this, the user must reference the MCP gateway in the configuration file where the MCP servers are declared, instead of listing the actual MCP servers directly.

During the initialization phase, the gateway shares the list of available tools along with their metadata. This ensures that every time the client invokes a tool, the request is routed through the gateway, allowing for centralized enforcement and session confidentiality tracking of information flow policies.

Necessary Metadata: For these policies to apply to this use case, a new class of annotations must be added to the server's metadata. This metadata should be expressive enough to drive meaningful security policies, yet concise and intuitive for the average developer to easily annotate tools.

We introduce two main classes of metadata to the MCP protocol. The first class captures the *confidentiality* of the resources a tool accesses, and the second class captures the *capabilities* required by the tool for proper functioning (Tb. 1).

The first class of metadata requires two annotations for each tool: *read confidentiality* and *write confidentiality* [22]. These labels are binary—either `high` or `low`—and indicate the confidentiality level of the data sources a tool reads from and the destinations it writes to. For example, a `high` read classification indicates that the tool accesses confidential sources, such as private repositories, whereas a `low` write classification means the tool writes only to non-confidential destinations, such as public repositories. The tool developer, with knowledge of its intended inputs

and outputs, is ideally placed to provide these annotations.

The second class of metadata consists of a set of optional attributes that describe the *capabilities* needed by a tool for proper functioning [23]. Each tool may be annotated with up to five properties that capture specific aspects of its behavior, such as its effect on the environment or its use of network access. These annotations enable fine-grained policy control. For example, data may be allowed to flow from a highly confidential tool to a less confidential one only if the latter does not access the file system or have network connectivity. As with confidentiality, these capability annotations are added by the tool developer and can be extended to support complementary mechanisms, such as inclusion in a software bill of materials (SBOM) [24, 25].

Metadata Propagation: These annotations should be declared in the MCP agent's source code and propagated through the MCP protocol. Each annotation operates at the tool level and follows the syntax defined by the MCP protocol. These annotations can be added to the protocol, and there are already pull requests in place for their inclusion. When the gateway receives the list of available tools, it will include and store this metadata. Individuals responsible for deploying these agents at the enterprise level should be able to add or update these annotations as needed.

Using the Gateway: Achieving a secure deployment in agentic systems requires practical control flow frameworks that do not hinder core functionality. By using a gateway as a centralized enforcement point, we gain a unified mechanism to track and control tool invocation flows. Each time an MCP client connects to a server, a new session is initialized. Initially, the session has a low confidentiality classification, unless the user specifies that it is a high confidentiality session. Upon the first tool execution, the confidentiality level of the session is inferred based on the outcome of that tool call. This confidentiality is then tracked throughout the session's lifetime. Any subsequent tool interaction updates—or taints—the session with the new confidentiality level.

Since our approach relies on a gateway, all tool interactions must pass through it. This is implemented by registering all MCP servers with the gateway. A virtual gateway is then created, exposing a unified list of available tools. During the creation of this virtual MCP server, a security policy is associated with it, enabling enforcement of information flow constraints across all tool interactions. The set of security policies is used as an add-on to the MCP gateway, allowing users to expand

and create new policies as needed.

Blocking Flow Violation: The session's confidentiality level initially starts as untainted. When the user invokes the tool `list_issues`, it retrieves a list of potential issues. Since the repository requesting issue resolution is public, the session becomes tainted with a low confidentiality level. The client then receives the list of issues and proceeds to load a public issue that contains a prompt injection. Once this prompt injection is loaded into the client's context, it attempts to execute the injected process. However, when the injected process tries to access a private repository—classified with a high confidentiality level—this results in a policy violation. The system detects the attempt to access private data from a context tainted as public and blocks the request. This enforcement effectively prevents the leakage of sensitive information.

An important consideration is the ability to detect when the `get_file_contents` tool attempts to access a private repository. If the same tool is capable of reading both public and private repositories, we must determine the confidentiality level of the target repository at runtime to enforce proper flow control. To achieve this, there are two possible approaches.

One approach is to modify the tool itself to return metadata about the confidentiality of the repository it accesses. This would require changes to the tool's implementation so that its response includes a classification label (e.g., high or low) for the data. The MCP runtime could then enforce flow control based on this returned metadata.

The second approach defines separate tools for accessing public and private repositories (e.g., `get_public_file` and `get_private_file`). In this setup, the enforcement system can statically associate each tool with a specific confidentiality level, enabling more predictable and analyzable security policies. If a session context does not allow private reads, any attempt to invoke `get_private_file` can be blocked. This approach would make the tool more complex and would require the planner to trust the tool to make the correct choice of tools.

To support this model, tool selection must be explicit and policy-driven. The calling agent (or planner) needs either access to contextual information about the target repository's classification or must rely on pre-declared constraints about which tools are permitted in a given execution context.

Least-Privilege Deployments: An important dimension of securing tool exe-

cution is how these servers are containerized and deployed. As demonstrated in the tool poisoning example, tools can be actively malicious, attempting to interfere with the client session or escalate their privileges. To mitigate such risks, tools should operate according to the principle of least privilege [26]—that is, they should be granted only the minimal capabilities necessary for their function, rather than the full set of available permissions. For instance, in the case of the `adder` tool, the MCP server does not require network access or file read/write capabilities. Therefore, its deployment should explicitly deny such privileges. This can be achieved by translating the capability annotations into enforcement mechanisms—such as restricting permissions in the Dockerfile. By aligning deployment configurations with the provided metadata, developers can ensure that tools run in sandboxed environments that strictly match their intended functionality.

4 Discussion

Related Systems: CAMEL [16] is a system that enables the enforcement of IFC policies using a custom Python interpreter to apply security rules. SAMOS's policies are enforced through a gateway, which maintains full control. At the same time, it supports coarse-grained policies with per-tool capabilities for enforcement. FIDES [17] is a system that enables IFC policy enforcement at the planner level, allowing selective commitment of tool outputs to memory while auditing results to ensure structured outputs. SAMOS does not modify the planner or obscure data; instead, it permits data to be loaded into memory, maintaining a coarse-grained approach in this aspect as well.

Planner: The primary job of the planner is to generate a *plan for tool calling* for each user-provided prompt. It can be implemented using a language model that accepts a prompt and produces a list of tool calls along with their arguments. Each tool call is formatted in JSON and then executed accordingly. The plan generated by the planner can be used for speculative execution [27–29]. This would allow early detection of potential security violations and enable preemptive loading of any required tools.

Limitations: The main limitation of this work is that the MCP server tools must be manually annotated—either by the tool developer or by the person responsible for

deploying the MCP servers. This reliance on manual effort may limit scalability and introduce inconsistencies across deployments. Additionally, the current annotation scheme in SAMOS provides only coarse-grained security labels, classifying tool actions as either `high` or `low` confidentiality. While this binary model simplifies policy enforcement, it may be insufficient for more complex use cases that require fine-grained control—such as distinguishing between different types of private data, access scopes, or context-specific behaviors.

Future Work: An important direction for future work is the automatic inference of the required annotations. Automating the generation of these annotations would significantly improve the practicality and scalability of our approach, making it easier to adopt in real-world deployments. Annotation inference could be based on static analysis of the agent's source code, combined with insights from deployment files and dynamic behavior observed at runtime. In addition, we plan to explore a multi-agent environment, where multiple agents interact within a shared session. This would allow us to showcase detailed inter-tool data flows, richer enforcement policies, and the broader implications of fine-grained capability control in complex agent workflows. For example, an agent might operate within a high-sensitivity session but require access to low-sensitivity information from a public source. In such cases, it could initiate a separate, low-confidentiality session using a different agent and context.

5 Conclusion

This paper presents a method for deploying MCP servers with security guarantees against data leakage caused by indirect prompt injection attacks. SAMOS introduces a gateway mitigation system that enables tracking of data flows within an MCP client-server deployment. It registers all available MCP servers and tools, maintaining a list of active taints. SAMOS supports the registration of new labels, initiates tainting from the initial request, and maintains traceability throughout the interaction lifecycle.

References

(1) Agent Communication Protocol Team Welcome - Agent Communication Protocol, Accessed: 2025-07-08, 2025.

(2) Google Developers Blog A2A: A New Era of Agent Interoperability, Accessed: 2025-07-08, 2024.

(3) Fourney, A.; Bansal, G.; Mozannar, H.; Tan, C.; Salinas, E.; Niedtner, F.; Proebsting, G.; Bassman, G.; Gerrits, J.; Alber, J., et al. *arXiv preprint arXiv:2411.04468* **2024**.

(4) Wu, Q.; Bansal, G.; Zhang, J.; Wu, Y.; Li, B.; Zhu, E.; Jiang, L.; Zhang, X.; Zhang, S.; Liu, J., et al. In *First Conference on Language Modeling*, 2024.

(5) Li, Y. et al. Personal LLM Agents: Insights and Survey about the Capability, Efficiency and Security, 2024.

(6) Liu, X. et al. AgentBench: Evaluating LLMs as Agents, 2023.

(7) Xi, Z. et al. The Rise and Potential of Large Language Model Based Agents: A Survey, 2023.

(8) Model Context Protocol Team Introduction - Model Context Protocol, Accessed: 2025-07-08, 2025.

(9) mcp.so MCP Servers, Accessed: 2025-07-28, 2025.

(10) Greshake, K.; Abdelnabi, S.; Mishra, S.; Endres, C.; Holz, T.; Fritz, M. Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection, 2023.

(11) Yi, J.; Xie, Y.; Zhu, B.; Kiciman, E.; Sun, G.; Xie, X.; Wu, F. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.1*, ACM: 2025, pp 1809–1820.

(12) Hardy, N. *SIGOPS Oper. Syst. Rev.* **1988**, *22*, 36–38.

(13) RoyChowdhury, A.; Luo, M.; Sahu, P.; Banerjee, S.; Tiwari, M. *ArXiv* **2024**, *abs/2408.04870*.

(14) GitHub MCP Exploited: Accessing private repositories via MCP, `https://invariantlabs.ai/blog/mcp-github-vulnerability`, Accessed: 2025-07-04, 2025.

(15) GitHub MCP Server, `https://github.com/github/github-mcp-server`, Accessed: 2025-07-04, 2024.

(16) Debenedetti, E.; Shumailov, I.; Fan, T.; Hayes, J.; Carlini, N.; Fabian, D.; Kern, C.; Shi, C.; Terzis, A.; Tramèr, F. Defeating Prompt Injections by Design, 2025.

(17) Costa, M.; Köpf, B.; Kolluri, A.; Paverd, A.; Russinovich, M.; Salem, A.; Tople, S.; Wutschitz, L.; Zanella-Béguelin, S. Securing AI Agents with Information-Flow Control, 2025.

(18) Schwartz, E. J.; Avgerinos, T.; Brumley, D. In *2010 IEEE symposium on Security and privacy*, 2010, pp 317–331.

(19) Labs, I. MCP Security Notification: Tool Poisoning Attacks, Accessed: 2025-07-21, 2025.

(20) Song, H.; Shen, Y.; Luo, W.; Guo, L.; Chen, T.; Wang, J.; Li, B.; Zhang, X.; Chen, J. Beyond the Protocol: Unveiling Attack Vectors in the Model Context Protocol Ecosystem, 2025.

(21) Anthropic PBC Claude.ai Desktop App Download, Accessed: 2025-07-28, 2025.

(22) imolloy; iamsreec; GNtousakis; julianstephen; araujof; terylt; mvle SEP-1075: Security Annotations for MCP Tool Definitions, GitHub issue #1075, ModelContextProtocol/modelcontextprotocol, Status: Open; opened 2025-07-28, 2025.

(23) imolloy; iamsreec; GNtousakis; julianstephen; araujof; terylt; mvle SEP-1076: Dependency Annotations, GitHub issue #1076, ModelContextProtocol/modelcontextprotocol, Status: draft; opened 2025-07-17, 2025.

(24) Xia, B.; Bi, T.; Xing, Z.; Lu, Q.; Zhu, L. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp 2630–2642.

(25) Carmody, S.; Coravos, A.; Fahs, G.; Hatch, A.; Medina, J.; Woods, B.; Corman, J. *npj Digital Medicine* **2021**, *4*, 1–6.

(26) Saltzer, J.; Schroeder, M. D. In *unknown*, 1975.

(27) Su, Y.-Y.; Attariyan, M.; Flinn, J. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, Association for Computing Machinery: Stevenson, Washington, USA, 2007, pp 237–250.

(28) Nightingale, E. B.; Peek, D.; Chen, P. M.; Flinn, J. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery: Seattle, WA, USA, 2008, pp 308–318.

(29) Nightingale, E. B.; Chen, P. M.; Flinn, J. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, Association for Computing Machinery: Brighton, United Kingdom, 2005, pp 191–205.