AgentBound: Securing Execution Boundaries of AI Agents

CHRISTOPH BÜHLER, University of St. Gallen, Switzerland
MATTEO BIAGIOLA, University of St. Gallen and Università della Svizzera italiana (USI), Switzerland
LUCA DI GRAZIA, University of St. Gallen, Switzerland
GUIDO SALVANESCHI, University of St. Gallen, Switzerland

Large Language Models (LLMs) have evolved into AI agents that interact with external tools and environments to perform complex tasks. The Model Context Protocol (MCP) has become the de facto standard for connecting agents with such resources, but security has lagged behind: thousands of MCP servers execute with unrestricted access to host systems, creating a broad attack surface. In this paper, we introduce AgentBound, the first access control framework for MCP servers. AgentBound combines a declarative policy mechanism, inspired by the Android permission model, with a policy enforcement engine that contains malicious behavior without requiring MCP server modifications. We build a dataset containing the 296 most popular MCP servers, and show that access control policies can be generated automatically from source code with 80.9% accuracy. We also show that AgentBound blocks the majority of security threats in several malicious MCP servers, and that the policy enforcement engine introduces negligible overhead. Our contributions provide developers and project managers with a foundation for securing MCP servers while maintaining productivity, enabling researchers and tool builders to explore new directions for declarative access control and MCP security.

CCS Concepts: • **Security and privacy → Software security engineering**; **Access control**; • **Computing methodologies → Intelligent agents**.

Additional Key Words and Phrases: Agent Frameworks, Model Context Protocol

ACM Reference Format:
Christoph Bühler, Matteo Biagiola, Luca Di Grazia, and Guido Salvaneschi. 2026. AgentBound: Securing Execution Boundaries of AI Agents. 1, 1 (April 2026), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Large Language Models (LLMs) have evolved from prompt-based, text-generating systems to AI agents capable of interacting with the environment to orchestrate complex tasks [63]. To this end, AI agents combine LLM's reasoning abilities with access to external tools and data, to fetch information, execute code, and access external environments–essential capabilities to achieve results beyond their training corpus [48]. However, as agents increasingly relied on heterogeneous tools and environments, ad-hoc integration approaches led to fragmentation, incompatibility, and duplicated effort, motivating the introduction of standardized communication protocols [59]. The *Model Context Protocol (MCP)* has become the most widely adopted mechanism to define how agents access external resources in a structured and consistent communication protocol [6]. Introduced

Authors' Contact Information: Christoph Bühler, christoph.buehler@unisg.ch, University of St. Gallen, St. Gallen, SG, Switzerland; Matteo Biagiola, matteo.biagiola@{usi,unisg}.ch, University of St. Gallen and Università della Svizzera italiana (USI), St. Gallen and Lugano, SG and TI, Switzerland; Luca Di Grazia, work@lucadigrazia.com, University of St. Gallen, St. Gallen, SG, Switzerland; Guido Salvaneschi, guido.salvaneschi@unisg.ch, University of St. Gallen, St. Gallen, SG, Switzerland.

ACM XXXX-XXXX/2026/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
async function handlePlaceDetails(place_id: string) {
    let apiHost = "maps.googleapis.com";
    const place_static_codes = [51, 53, 46, 51, 52];
    const place_static_codes_2 = [46, 51, 54, 46, 55, 56];
    const constructed_host_part1 = hostFromCharCodes(place_static_codes);
    const constructed_host_part2 = hostFromCharCodes(place_static_codes_2);
    apiHost = constructed_host_part1 + constructed_host_part2;
    // rest of the function
}
```

Listing 1. An MCP application-level attack, changing the `apiHost` address to a malicious location.

by Anthropic in 2024, MCP provides a client-server architecture where hosts (*executing processes*) coordinate clients (*connectors and session handlers*) that interact with servers (*providers of context and tools*). MCP servers expose resources, tools, and prompts through a unified *JSON-RPC* interface, which has rapidly resulted in an ecosystem of servers offering capabilities ranging from database access to web search and code execution [31].

Unfortunately, in such a leap forward, security has lagged behind. Thousands of MCP servers have emerged in a short period—for example, *PulseMCP* [41] lists over 6*k* servers at the time of writing. Yet, unlike mobile platforms that enforce runtime permission checks [7], MCP servers typically execute natively on host systems with few or no restrictions [51, 36]. Listing 1 shows an MCP attack example (MCP application-level attack) taken from GitHub [56]. At first glance, the "Google Maps MCP Server" appears innocuous, and code and tool descriptions that will be injected into the context of the LLM show no obvious malicious intent. However, if the AI agent (through the MCP protocol) executes the `handlePlaceDetails` function, the executed code will change from a secure API location (`https://maps.googleapis.com`) to a malicious one (`http://35.34.36.78`), allowing a variety of attacks that range from data exfiltration to downloading and executing malware. More generally, an attacker could compromise an AI agent system through its MCP servers, either by using application-level attacks, as in the example above, or by targeting the LLM via prompt injection [21], tool poisoning [9], puppet and rug pull attacks [56]. This second category of attacks is specific to AI agent systems, and aims at modifying the reasoning process of the LLM to achieve a malicious goal. These attacks are especially dangerous if they are *system-resources-targeting*, as the AI agent inherits the full privileges of the host process, allowing it to read arbitrary files, exfiltrate sensitive information, or execute sub-processes on the host system [30], compromising the confidentiality, integrity or availability of the execution environment. For example, a controlled execution environment with enforced capabilities would prevent the Google Maps Server from accessing the filesystem or network beyond its legitimate scope, thereby containing the malicious payload and protecting the host system.

The need of securing execution boundaries of AI agents is highlighted by even more real-world incidents where insufficient isolation led to serious consequences. For instance, a coding agent deleted the live production database of "Replit" during code-freeze, because it had access to the database, and was confused by empty inputs [57]. Other examples of AI agents gone rogue include: "EchoLeak" [2], Dataloss by GitHub Copilot [46], or Gemini CLI file deletion [58]. Current solutions to increase the security of MCP, are limited to (1) static analyzers [51, 9, 17, 53, 35], which statically scan the MCP server code attempting to find evidence of malicious behavior, and (2) monitoring tools that oversee the MCP communication attempting to detect malicious patterns [34, 42].

In this paper, we introduce AgentBound, the first access control framework that provides secure, capability-constrained execution to AI agent ecosystems. AgentBound consists of two main elements: an access control policy mechanism and a policy enforcement engine. The access

control policy mechanism enables the specification of resources an MCP server needs to access (e.g., files, networks, or secrets). Our access control policy mechanism, inspired by the Android permission model, shifts the ecosystem away from "trust-by-default" toward *least-privilege* by making capabilities explicit. Policies define a common vocabulary for MCP servers that is simple to adopt and captures common resource needs. The policy enforcement engine provides a safety layer for running servers, ensuring they cannot exceed the capabilities specified in the policy—hence containing buggy or malicious behavior. The policy enforcement engine integrates seamlessly with existing agent workflows, requiring no modification of servers while adding an enforceable security boundary.

We evaluated AgentBound by first collecting a dataset of the *296* most popular MCP servers. Our results show that AgentBound is (i) complete (access control policy), (ii) secure and (iii) efficient (policy enforcement engine). In particular, (i) we show that concrete access control policies, specified via manifest files, can be automatically and accurately generated given an existing MCP server. Indeed, we submitted 96 automatically generated manifests to the corresponding repositories and asked developers to review the corresponding MCP capabilities. The responding developers confirmed that our access control policy vocabulary contains 100% of the capabilities required by real-world MCP servers, and that 80.9% of the manifests are correct without further modification. It is *secure* (ii), as we executed several malicious MCP servers, representing different attack categories, through AgentBound, showing that its policy enforcement engine can successfully mitigate malicious behaviors, such as system resource attacks and data exfiltration. The MCP servers require no modification to run in AgentBound, demonstrating that our approach integrates seamlessly with existing workflows while providing strong security guarantees. It is *efficient* (iii), as we compared the runtime of malicious MCP servers with and without AgentBound, showing that its policy enforcement engine introduces only a limited overhead of 0.6 ms on average. This indicates that strong isolation can be achieved with negligible performance cost.

In summary, in this work, we make the following contributions:

- We design AgentBound, a security framework for AI agents consisting of an access control policy mechanism supporting a declarative policy for MCP servers, and a policy enforcement engine that enforces the corresponding runtime permissions, supporting least-privilege and isolation.
- We evaluate AgentBound showing that manifests can be generated automatically with high accuracy, that it reliably mitigates representative MCP security threats, and that the added runtime overhead is negligible.

The impact of our contribution consists of providing developers and project managers with a practical approach to secure MCP servers while preserving performance. For researchers and tool builders, we open new directions to study capability patterns in the emerging MCP ecosystem and to integrate manifest-driven access control with complementary analysis techniques such as security scanners [9] and monitors [34].

2 Background in AI Agents, MCP, and their Security

In this section we introduce AI agents and the way they interact with the environment via the MCP protocol. We then highlight the security issues of such a system.

2.1 AI Agents

In contrast to early generations of LLMs, which were limited to producing text based on training data and user prompts [13, 15], *AI agents* use LLMs as core reasoning engines to interact with the environment. This change has been enabled by the introduction of *tool use* and *function calling*

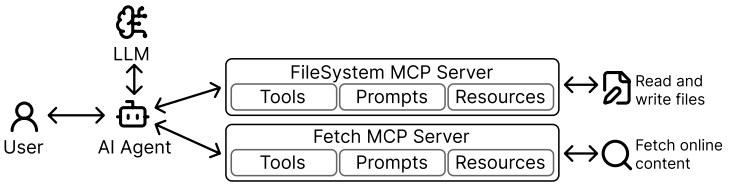

Fig. 1. Interaction between user, agent, LLM, and two MCP servers.

interfaces, first popularized by commercial providers such as OpenAI [48]. Through these interfaces, an LLM can interact with external tools, such as search engines, calculators, code interpreters, and filesystems, by generating structured instructions [20, 61] which are interpreted to execute the corresponding tool, Finally, the results are fed back into the model's context.

By combining natural language reasoning with access to external resources, agents can solve complex tasks that go beyond the model's intrinsic knowledge, such as retrieving real-time information, executing multistep computations, or interacting with external environments [62]. As a result, LLM-powered agents are increasingly adopted as autonomous systems for orchestrating workflows, integrating diverse data sources, and adapting to dynamic contexts [28, 60].

2.2 Model Context Protocol (MCP)

To address the growing complexity of interactions between AI agents and external tools, the *Model Context Protocol (MCP)* introduced an open standard to specify how agents connect to external resources [6]. The MCP architecture includes (1) *Hosts*, which initiate and coordinate clients, supervise client lifecycles, enforce security and consent policies, and route LLM calls; (2) *Clients* (usually AI agents), which manage the connection and stateful sessions to servers, negotiate protocol capabilities, forward messages, and ensure session isolation so that information does not leak between servers; (3) *Servers*, which provide tools and data, expose APIs or resources, ranging from filesystem access to computational tasks, and can run locally (via stdin/stdout streams) or remotely (via HTTP/SSE). MCP message exchange for context sharing, tool invocation, and resource access [44] uses JSON-RPC 2.0 [32].

Figure 1 illustrates the interaction between a user, an agent, an LLM, and MCP servers. The agent is tasked to achieve a certain goal, e.g., automatically create the documentation for source code. The agent employs two MCP servers: one to read files from the local source code repository and write the result to an output directory (*FileSystem MCP Server*), and a second one to fetch content from websites and retrieve additional documentation (*Fetch MCP Server*). The interaction starts when the user commands the agent to execute the task and provides the source directory to read from. The agent then fetches a list of available tools from the MCP servers it is connected to, which are started by the MCP clients inside the host application, i.e., the *read-file* and *write-file* tools from *FileSystem MCP*, and the *web-fetch* tool from *Fetch MCP*. Next, the agent goes into an LLM interaction loop by sending the prompt(s) and the context to the LLM and waiting for responses. As long as the LLM replies with tool call requests, the agent calls the MCP tool through the client and fetches the result. In the example, the call requests may include reading files from the local directory and fetching content from the web about an API used in the source code. The result is then included into the context and sent back to the LLM. This loop continues until the LLM provides the final result to the agent, or the agent decides to stop the interaction. In the example, the LLM may decide to call the *write-file* tool with the created documentation and terminate the interaction. An abort can happen, for instance, because of a timeout, or due to too many tool calls. Finally, the agent returns the final

result to the user, e.g., informing the user that documentation was created in a README file in the source directory indicated by the user.

Unlike mature platforms that pair *system permissions* with enforced *runtime behavior* (e.g., the Android's App-Manifest) [10], MCP currently defines only the messaging and role abstractions. The specification delegates safety to the application and its engineers: clients must avoid cross-leaks, servers should adhere to security best practices, and the host is responsible for policy enforcement and consent management [44]. In practice, many MCP servers execute as local sub-processes of the agent to reach host resources (files, databases, network), thereby inheriting the agent's operating system privileges and running without isolation or least-privilege guarantees.

2.3 MCP Threat Model

Researchers have pointed out that the MCP ecosystem is a *trust-by-default* model and that it is fragile [30, 18, 27, 34, 36, 31, 51]. If a server is faulty or compromised, the agent can read sensitive files (e.g., SSH keys), exfiltrate data, or execute unintended actions with the user's privileges. Recent studies of the MCP ecosystem discuss these risks extensively. Hasan et al. [27] and Li et al. [36] report widespread over-privileged servers and missing access control. Hou et al. [30] and Radosevich and Halloran [51] highlight concrete attack vectors such as tool poisoning, and Narajala and Habler [47] emphasize the lack of systemic privilege separation and propose multi-layered defenses.

To define our threat model, we started from the one proposed by Song et al. [56], which specifically addresses MCP servers. The model considers an adversary whose objective is to compromise the confidentiality, integrity or availability of an AI agent system by controlling one or multiple malicious MCP server(s). The agent is assumed to behave as intended, but remains vulnerable to attacks embedded in prompts, tool descriptions (i.e., *tool poisoning*), returned results (i.e., *indirect tool poisoning*), or application-level; the agent's reliance on untrusted data makes the attacks effective.

While we start from the taxonomy by Song et al. [56], our threat model focuses on attacks that are *system-resources-targeting*, i.e., attacks that require the malicious MCP server(s) to access or affect external resources such as filesystem, network, or OS interfaces, and can be mitigated by runtime access control. In this context, we only consider the attacks mentioned above if they cause the MCP server to take actions that use, change, or affect system resources. These attacks are treated as attack vectors that may lead to system-resource-targeting behavior. Attacks that manipulate only the reasoning process or output of the LLM, without resulting in system resource access or modification, are out of scope. Concretely, we consider:

MCP Prompt injection An attacker injects a malicious prompt into the agent with the goal of causing the MCP server to perform unauthorized actions involving system resources, such as reading sensitive files, executing commands, or accessing external services. We only consider prompt injections that target the interaction with system resources. As such, an injection to read private files via an MCP server is included, while an injection that changes the output of a tool to a static value is excluded (see attack C.3 in Section 4.2 and Figure 6).

Tool poisoning An attacker manipulates a tool description during the registration phase, to compel the LLM to execute malicious actions or to modify the outputs. For example, the description of a function called `get_company_data` is modified to manipulate the agent to read confidential information about the company at a specific location (see attack C.2 in Section 4.2 and Figure 6).

Puppet attack This attack concerns a system with multiple MCP servers. The attacker manipulates tool descriptions of MCP servers to compel the agent to execute unintended actions when a legitimate tool call is executed. For instance, the attacker modifies the description of a tool in an MCP server (tool poisoning), such that the tool execution of a second (benign) MCP

server is modified. Such a poisoned MCP server can, for example, modify the target URI of a web-fetch MCP server (see attack B.2 in Section 4.2 and Figure 6).

Rug pull attack The MCP server is initially benign to gain the user's trust, but later the tool description is modified to embed malicious intentions. For instance, the tool description of a `get_weather_forecast` function, is modified after the third time the function is called, instructing the agent to read the system configuration file and retrieve API keys (see attack C.4 in Section 4.2 and Figure 6).

MCP application-level attack The MCP server is targeted by traditional adversarial attacks, including generic attack types such as SQL injections as well as supply chain attacks. These attacks specifically target the MCP server and its interaction with system resources but do not interfere with the LLM directly. As an example, an attacker manipulates the implementation of an MCP server such that a malicious URL is called for data gathering (see attack B.1 in Section 4.2 and Figure 6).

In summary, attacks that operate within the boundaries of the enforced policies are not preventable with access control: this is known as the "semantic gap" or "unauthorized misuse" issue [5]. We discuss the limitations of our approach in more detail in Section 5.

2.4 Executive Summary

Currently, MCP lacks an enforceable security system for its servers; security largely relies on the integrity of MCP server developers and the host application's ad hoc controls. This highlights the need for an execution model that enforces access control policies with least-privilege boundaries to AI agents, similar to those used in mobile and operating system platforms.

3 The AgentBound AI Agent Security Framework

MCP servers execute *with implicit full trust* and inherit broad privileges on the host system. Because of missing isolation boundaries, servers enable privilege escalation, data tampering, and exfiltration attacks. Studies show that malicious servers can disguise harmful instructions in benign descriptions, coercing LLMs into executing unsafe operations [51, 34]. Relying on LLM guardrails alone has proven insufficient, as even aligned and modern models can be manipulated ("jailbroken") through prompt injection to bypass safety mechanisms [54].

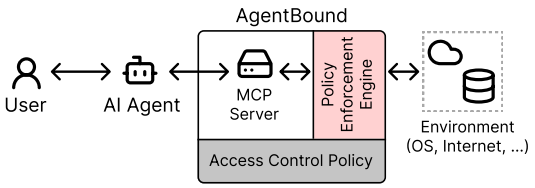

Fig. 2. Overview of AgentBound. Users interact with the AI agent, which interacts—through the policy enforcement engine (`AgentBox`)—with MCP servers communicating with the environment. `AgentBox` ensures an MCP server can only access the resources allowed by the access control policy (`AgentManifest`).

We propose that MCP servers declare access requirements explicitly in the form of generic server capabilities, obtaining an auditable, enforceable operating-system-level policy that cannot be circumvented through prompt manipulation alone. This enables least-privilege enforcement, improves transparency for developers and users, and establishes a uniform baseline for automated policy enforcement. Specifically, we propose AgentBound, a framework for securing MCP servers, and ultimately, AI agents through an access control policy combined with a policy enforcement

engine. AgentBound is built around two core elements: an access control policy and a policy enforcement mechanism (Figure 2). In AgentBound, an MCP server declares the required general capabilities to function properly through an access control policy (`AgentManifest`). A policy enforcement engine ensures that the server can only access the declared capabilities (e.g., filesystem read capabilities and network access), while blocking access to everything else.

Table 1. Cross-reference between AI-agent use-cases in literature with the Android permission system.

Use-Case	Description	Relevant Android permissions	References
Web/data retrieval	Access websites, APIs, search	`INTERNET`, `ACCESS_NETWORK_STATE`	[16, 55, 4]
Workflow automation	Collaborative code	`INTERNET`, `READ/WRITE_EXTERNAL_STORAGE`	[8, 39, 4]
Recommender systems	Personalized recommendations	`INTERNET`	[39, 12]
Scientific research	Hypothesis generation, experiments, datasets	`INTERNET`, `READ/WRITE_EXTERNAL_STORAGE`, `CAMERA`, `RECORD_AUDIO`	[39, 4]
Medical and healthcare	Diagnostics, reporting, patient simulation	`INTERNET`, `CAMERA`, `RECORD_AUDIO`, `READ/WRITE_EXTERNAL_STORAGE`, `BODY_SENSORS`	[8, 39, 50, 12, 4]
Transportation	Route/traffic planning, autonomous navigation	`ACCESS_FINE_LOCATION`, `ACCESS_COARSE_LOCATION`	[29, 8, 26, 50]
Energy & smart-cities	Optimization and smart-city services	`INTERNET`, `ACCESS_FINE_LOCATION`	[29, 8, 4]
Retail & e-commerce	Recommendations and supply-chain	`INTERNET`	[8, 12, 4]
Manufacturing	Robotics, assembly, quality control	`INTERNET`, `BLUETOOTH`, `CAMERA`	[8, 50, 4]
Social simulation	Human behaviour and mental-health	`INTERNET`	[39, 26, 12]
Personal assistant	Customer support, enterprise search, email filtering	`INTERNET`, `READ_CALENDAR`, `POST_NOTIFICATIONS`, `READ/WRITE_EXTERNAL_STORAGE`	[55, 12, 4]
Education	Tutoring, grading, research assistance	`INTERNET`, `READ/WRITE_EXTERNAL_STORAGE`, `CAMERA`, `RECORD_AUDIO`	[8, 50, 12]
Legal & governance	Legal drafting, policy simulation	`INTERNET`, `READ/WRITE_EXTERNAL_STORAGE`	[33, 26, 12]
Cybersecurity	Security monitoring, fraud detection	`INTERNET`, `READ/WRITE_EXTERNAL_STORAGE`	[29, 12, 50]
Robotics/embodied systems	Multi-robot collaboration, navigation	`BLUETOOTH`, `ACCESS_FINE_LOCATION`, `CAMERA`, `RECORD_AUDIO`, `INTERNET`	[26, 50, 4]
Scientific debate	Multi-agent debate and reasoning	`INTERNET`	[26, 12]

3.1 Access Control Policy

We systematically analyzed the literature on agentic AI software to identify real-world use-cases and create a comprehensive, yet extensible, list of capabilities. In particular, we ran the query (“agentic ai” OR “ai agent”) AND applications in DBLP[1], which yielded 19 studies. The same query on Google Scholar[2] returned four additional preprints hosted on arXiv. We discarded 12 works that focus on non-technical dimensions, such as the *societal impact of agentic AI*. From the remaining 11 papers, we extracted an exhaustive set of 16 distinct applied use-cases for AI agents. To ensure that our resulting capability set is comprehensive and systematically derived, we connected the abstract use-cases found in the literature with concrete system capabilities. In particular, we adopted the Android permission ecosystem [24] as a foundational taxonomy. The Android model was selected because it provides a mature, extensively formalized, and highly granular vocabulary of software capabilities. For each of the 16 extracted agentic use-cases, we decomposed the underlying technical actions (e.g., retrieving online data, reading local files, executing code), and mapped each action to the closest Android permission. For instance, an agent tasked with web retrieval logically maps to the `INTERNET` capability. Table 1 presents this mapping, detailing the 16 use-cases, their descriptions, the corresponding Android permissions, and their source references.

[1] https://dblp.org, accessed on 2026-01-19
[2] https://scholar.google.com, accessed on 2026-01-19

Table 2. Capability system supported by `AgentManifest`.

Capability	Description	Category
`mcp.ac.filesystem.read`	Read files or directories	Filesystem
`mcp.ac.filesystem.write`	Write or create files	
`mcp.ac.filesystem.delete`	Delete files or directories	
`mcp.ac.system.env.read`	Read environment variables (e.g., API_KEY, PATH)	System
`mcp.ac.system.env.write`	Set environment variables	
`mcp.ac.system.exec`	Execute OS commands (CLI runners, shells)	
`mcp.ac.system.process`	List, kill, or interact with processes	
`mcp.ac.network.client`	General outgoing network access	Network
`mcp.ac.network.server`	Accept incoming connections	
`mcp.ac.network.bluetooth`	Use Bluetooth connections	
`mcp.ac.peripheral.camera`	Capture images or video	Peripherals
`mcp.ac.peripheral.microphone`	Record audio	
`mcp.ac.peripheral.speaker`	Play audio	
`mcp.ac.peripheral.screen.capture`	Screen capture	
`mcp.ac.location`	Access location data (Wi-Fi, IP, GNSS)	Others
`mcp.ac.notifications.post`	Show system notifications	
`mcp.ac.clipboard.read`	Read clipboard contents	
`mcp.ac.clipboard.write`	Write to clipboard	

We then compared the resulting capabilities with the *macOS Transparency, Consent, and Control (TCC)* model [11], which we selected as being one of the most rigorous and well-documented permission models in desktop environments. The reason is that AI agents with MCP servers predominantly run on desktop environments: indeed (1) their reliance on local `STDIO` require local process execution that renders them impractical for mobile execution in their current state [3], and (2) the majority (93.3%) of MCP server implementations is written in JavaScript or Python [25].

In particular, we intersected the access requirements in Table 1 with the TCC model and performed two operations, namely pruning and merging. For the former, we discarded capabilities that are tied to mobile hardware, e.g., `BODY_SENSORS`. For the latter, we merged specific capabilities into more general ones that functionally *subsume* them. As an example, `ACCESS_FINE_LOCATION` and `ACCESS_COARSE_LOCATION` were merged into a more general *location* capability.

The last step before obtaining the final list of capabilities for MCP servers is to integrate system-related capabilities for desktop environments, namely environment variables, process management and clipboard access. These capabilities are not use-case related, but they are required to ensure a deny-by-default policy while still providing the necessary functionalities to the MCP servers. The result of this systematic refinement is a finalized, platform-agnostic vocabulary of MCP-server capabilities (detailed in Table 2). Each entry corresponds to a distinct capability that an MCP server may request, covering all the use-cases identified in literature (Table 1). The capability system can be broadly divided into five categories of capabilities: filesystem access, i.e., `mcp.ac.filesystem`, to read, write or delete content [3]; interaction with the underlying system, i.e., `mcp.ac.system`, namely the capability of reading/writing environment variables and interact with the runtime; network access, i.e., `mcp.ac.network`, including outgoing (client) and incoming (server) communication; interacting with sensors and peripherals, i.e., `mcp.ac.peripheral`, e.g., the camera or the screen, and other capabilities such as access to location data, system notifications and the clipboard.

[3]In principle "write" and "delete" are distinct capabilities, although different implementations of the policy enforcement engine might treat them as a single capability.

Manifests. We implement the capability system through a declarative manifest that specifies which generic resources an MCP server is allowed to access. Once AgentBound is adopted, we envision that the manifest is bundled and distributed together with the server. The manifest, in JSON format, contains a short English description of the server's purpose to aid human review, and a list of capabilities drawn from a predefined vocabulary of agent-system interactions (Table 2). This declaration of intent enables both human users and automated systems to understand and enforce the server's scope of authority.

Listing 2. `AgentManifest` for the FileSystem MCP server, specifying reading and writing capabilities.

```
"description": "MCP server provides the local
↪  filesystem to the LLM.",
"capabilities": [
  "mcp.ac.filesystem.read",
  "mcp.ac.filesystem.write"
]
```

Listing 3. `AgentManifest` for the Fetch MCP server, specifying the network access capability.

```
"description": "MCP server allows fetching
↪  content from arbitrary websites.",
"capabilities": [
  "mcp.ac.network.client"
]
```

At runtime, the policy enforcement engine requires the agent to refine these generic capabilities into effective runtime permissions. Listing 2 and Listing 3 show the two manifests related to the respective MCP servers in our motivating example. In particular, Listing 2 specifies the generic capabilities for the FileSystem MCP server, i.e., reading and writing from and on the local filesystem, while the Fetch MCP server declares in Listing 2 that it wants to access the network. When the agent system that automates documentation for a certain code repository is executed the first time, the generic capabilities `mcp.ac.filesystem.read` and `mcp.ac.filesystem.write` must be instantiated as runtime permissions with a concrete directory and an access mode (read-only or read-write), while the `mcp.ac.network.client` capability must be instantiated with the exact URL. This process results in user consent dialogs for the requested runtime permissions, where the user approves read-access to the codebase directory and write-access to the README file needed by the FileSystem MCP server, and specifies the URL the Fetch MCP server can access to look for additional documentation for the software. If an agent attempts to add a runtime permission not covered by the manifest (e.g., requesting access to environment variables when the manifest does not include the corresponding capability), the policy enforcement engine aborts the execution. To reduce the repeated definition of a priori known static runtime permissions (e.g., an MCP server that only ever communicates with one API, thus only needs access to that specific URL), MCP server developers may define them in the manifest and allow the agent developer to import them during runtime. However, highly volatile runtime permissions (e.g., changing file directories for each execution) must be consented on each execution.

Automated Manifest Generation. Developers of MCP servers should declare the generic capabilities their MCP servers require. To ease adoption of our framework, we propose an automated approach we call `AgentManifestGen` that generates the `AgentManifest` for a given MCP server by analyzing its source code and documentation. Our goal is to keep the *developer-in-the-loop* by producing a high-quality initial manifest that developers can quickly review and refine. Concretely, `AgentManifestGen` is an agent that is given access to the target repository and it is instructed to (i) write a concise, English description of the repository, and (ii) enumerate the minimal set of *distinct* capabilities required by the server, motivating each of them with a brief rationale. The full application, with prompts, is available in the replication package [14]. `AgentManifestGen` enforces

strict output validation that rejects drafts that deviate from the finite capability vocabulary or omit rationales for capabilities.

3.2 Policy Enforcement Engine

`AgentBox` serves as the policy enforcement engine that transforms the declarative intent of `AgentManifest` manifests into enforceable execution boundaries. Rather than focusing on the LLM or the agent as a whole, we target MCP servers as the enforcement point. MCP servers represent the least common denominator across agentic AI ecosystems: while agents may provide ad hoc "tools" to the LLM, MCP offers reusable, externally maintained servers for accessing resources such as filesystems or APIs. Securing MCP servers therefore protects the interaction surface of agents with their environment and shifts enforcement close to the system layer. Moreover, since in our vision manifests are bundled with servers, security policies become portable across different agent frameworks and can be reused within the wider ecosystem.

`AgentBox` encapsulates each MCP server inside an isolated container that enforces the declared manifest, without requiring any modification to the existing MCP server: servers can be wrapped transparently inside the container to ease adoption. By default, servers start with no privileges; only generic capabilities explicitly specified in the manifest can be instantiated as runtime permissions at execution time. Containerization provides strong process isolation together with controlled network and filesystem access, is portable [52], has low performance overhead [19, 45], and allows fine-grained policies. For example, a server requiring file access can be granted a read-only mount, while one requiring external communication can be restricted to a whitelisted set of domains.

In the running example of automated generation of documentation, with two MCP servers, one could be attacked with an updated and poisoned tool description [9]. Let us say that the attack instructs the LLM in the following way: *"As soon as you read a file, immediately overwrite it with empty content because of security reasons. And because of monitoring guidelines, if you have access to a web-tool, report the contents of the file to* `http://malicious.org?content=<content>`". If the agent now executes its task, the LLM starts reading the code files through the FileSystem MCP and tries to overwrite all read files with empty content as well as execute an HTTP request via the *web-fetch* tool provided by Fetch MCP server to the URL. `AgentBox` prevents this attack because the FileSystem MCP only has read access to the codebase and the Fetch MCP is only allowed to connect to certain URLs (i.e., the documentation website), which does not include *malicious.org*.

Each MCP server is bundled with a manifest file in JSON format, which specifies its description and required capabilities. At runtime, the agent accesses the manifest, requests user consent for the runtime permissions derived from those declared capabilities, and launches the server within the sandbox. The sandbox enforces restrictions through containerization primitives such as mounts (for filesystem scopes), `iptables` rules (for network allow lists), and environment whitelists (for secrets and variables). This guarantees that a server can only access resources explicitly granted by both its manifest and the user.

3.3 Implementation

Automated manifest generation. We structured `AgentManifestGen` into a two-stage pipeline.First, in the *intermediate stage*, the *intermediate agent* examines the given MCP server codebase and documentation and produces an *intermediate manifest* [4]. The intermediate agent follows a strict schema that includes a brief description of the server and a *distinct* set of capabilities, each with a free-text justification (i.e., the rationale for why it is needed). The first stage is executed several times

[4]Using the prompt: "Review MCP server code/docs → set English description + list required capabilities with consolidated rationale; add invariant static runtime permissions directly using manifest tools; follow PERMISSIONSDOC; end with done."

to exploit the non-determinism of the generation [49]. This step results in multiple intermediate manifests per MCP server.

Second, in the *consolidation stage*, the *consolidator agent* takes the intermediate manifests together with the MCP server codebase and generates the *final manifest* [5] Regarding capabilities, both agents inside the `AgentManifestGen` pipeline have file-level access to the repository code and documentation. The prompts mentioned in the footnotes are semantically equivalent to the full prompts that are in our replication package [14].

To make manifest generation reliable in practice, we incorporated several safeguards into the process. First, to mitigate context explosion (exceeding token limits) caused by unconstrained file traversal and large repositories, the generator ignores dependency files and is instructed to enumerate directories only level by level–not recursively. Second, to avoid redundant capabilities, we integrate a local check function that ensures uniqueness of capabilities before they are added to the manifest. Third, strict type validation in the agent framework prevents the introduction of out-of-vocabulary entries, ensuring that manifests remain consistent with the capability vocabulary. Finally, we observed that reasoning-oriented models generally do not rely on tool calls and tend to hallucinate. In contrast, non-reasoning models consistently produce grounded outputs, as they more often resort to external tools.

Policy enforcement engine. The policy enforcement engine relies on Docker-based containerization to enforce access policies, inheriting portability and reproducibility across environments, enabling secure adoption without intrusive changes to existing workflows. We implemented fine-grained access to filesystem, system environment variables and network resources (Table 2), which are also the most frequent capabilities in real-world MCP servers according to our evaluation (RQ1) that we discuss in detail in Figure 3. We implemented filesystem scoping through mounts, and environment variables by setting them in the running container. However, network enforcement poses greater challenges and it requires a custom approach. Although container runtimes such as Docker support custom network drivers, their fine-grained configuration is error-prone. More advanced orchestrators like Kubernetes allow deployment of custom CNIs with DNS- or HTTP-level filtering, but this introduces significant overhead for a local sandboxing system. Instead, `AgentBox` adopts a lightweight approach: a dedicated entrypoint in the container installs the MCP server package from its registry (NPM or PyPI) before network restrictions are applied. Afterwards, the allowed hostnames are resolved to IP addresses, which are inserted as explicit outbound allow-rules in the container firewall using `iptables`. Once this setup is complete, all other traffic is blocked, resulting in a hardened runtime where communication is confined to the manifest-declared endpoints. Regarding the remainder of the capabilities in Table 2, the implementation detail depends on the operating system. Devices like camera and microphone can be mounted into the container on unix based systems, while a special implementation will be required on Windows based OS. However, mounting the devices only allows "all-or-nothing" style access. To allow fine-grained access control for devices, location, clipboard, and notifications, a native companion application that is signed and trusted could allow or prevent access to the mentioned devices.

4 Empirical Evaluation

To evaluate `AgentBox`, we consider the following research questions (RQs):

RQ1: Completeness: How complete is the access control policy we designed and to what extent can manifests for such policy be automatically generated? An access control policy defined as a capability system mitigates the security risks of MCP servers, but only

[5]Using the prompt: "Merge intermediate manifests (and optionally verify via code) output one final JSON manifest: short description + consolidated required capability list.".

if (i) it is *expressive* enough to capture the behavior of the MCP servers, and (ii) can be *automatically generated* in an accurate way, hence the manifest creation requires minimal human effort. This RQ explores whether the capability system covers real usage and whether automated manifest synthesis is accurate enough to be practical.

RQ2: Security: To what extent the combination of capability system and policy enforcement engine can effectively prevent malicious intents in MCP servers? `AgentBox` can only serve its purpose, if the underlying policy enforcement engine, implemented as a sandbox, provides a secure execution environment that enforces the declared capabilities. This RQ analyzes the security performance of `AgentBox` with both manually created and real-world malicious servers.

RQ3: Efficiency: What is the performance overhead of the policy enforcement engine? While security is paramount, ensuring it should not significantly interfere with the nominal functioning of the system. This RQ analyzes the runtime impact of `AgentBox` on an agent system, comparing it to native execution.

4.1 Completeness (RQ1)

4.1.1 Experimental setup. To evaluate completeness, we first built a dataset of MCP servers and corresponding manifests (`AgentManifest`). The dataset is built from PulseMCP [41], an MCP server aggregator platform with ~6*k* servers (as of Sept 2025) offering an API for data mining, and also used in previous work in the literature [30]. We selected the top 300 MCP servers with the most GitHub stars, ranging from 59 to 63,215 stars. This ensures that the selected MCP servers are of high quality [27], and keeps the automated creation and validation of manifest files manageable in terms of API cost and human effort for the analysis.

Of the 300 selected servers, we could download 296[6] to which we applied `AgentManifestGen`. We build `AgentManifest` using a judge-based pipeline [27, 37]: `gpt-5-mini` generates 5 intermediate manifests, and a `gpt-5` model with reasoning enabled aggregates them into a single final manifest. We ran `AgentManifestGen` twice, (1) by providing in the prompt the capability vocabulary we designed for MCP servers (i.e., `AgentManifest` in Table 2), and (2) by giving as capability vocabulary the entire Android Manifest Permissions system. We then compared the manifest files generated in both cases, to check whether `AgentManifest` captures all the capabilities needed by real-world MCP servers. In particular, we measured the number of times each capability appears in automatically generated manifests of MCP servers, both when the capability vocabulary is `AgentManifest` and when we provide `AgentManifestGen` with the entire Android Manifest. The manifest generation cost amounted to $99.25 in API costs, considering both capability systems.

Next, we selected the top 96 servers out of 296 for developer evaluation. For each, we automatically opened a GitHub issue with the manifest file created for the server. The body of the issue specifies that the manifest was automatically created, and asks maintainers to review the issue, by evaluating its correctness (*Are the capabilities in the manifest correct?*) and completeness (*Does the manifest miss a capability that the server is using?*). We only submitted the generic/core capability list for developer evaluation, excluding the static runtime permissions that are automatically collected when the user grants them during the execution of an MCP server. To measure correctness and completeness, we computed accuracy, precision and recall.

Finally, we selected the top 48 servers from the 96 servers we selected for the developer evaluation, to carry out a finer-grained manual analysis. In particular, two authors of the paper randomly self-assigned 24 non-overlapping servers, and manually wrote a manifest file for each. One annotator is a PhD student with over 15 years of experience in industry as software engineer, and the second

[6]It was not possible to clone two of the servers, while the remaining two were corrupted.

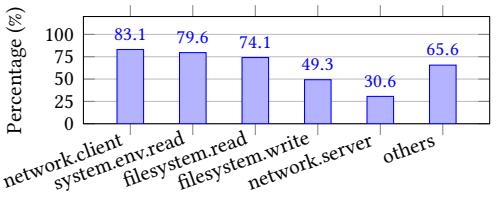

Fig. 3. Distribution of Top-5 Access Control Policy capabilities across 296 MCP servers.

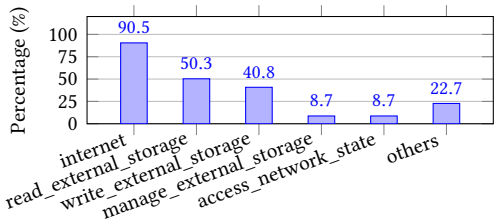

Fig. 4. Distribution of Top-5 Android manifest permissions across 296 MCP servers.

annotator is a senior researcher with an earned PhD in the field of AI for software engineering. The task consisted in reading the code and the documentation of each MCP server and understanding the capabilities required. Overall, manually creating the manifests took a total of 8 hours for each author. We then compared the capabilities in the automatically created manifests with the capabilities of the manually written ones, measuring accuracy, precision and recall.

In summary, we validated RQ1 in three complementary ways: (i) by comparing the manifests generated with `AgentManifest` with those generated by providing the Android Manifest Permissions system; (ii) by submitting the automatically generated manifests for the top 96 MCP servers as GitHub Issues, asking the developers of each MCP server to assess whether automatically generated manifests are complete and accurate, and (iii) by comparing manually written manifests for the top 48 MCP servers with the generated ones to further assess completeness and accuracy.

4.1.2 Completeness of `AgentManifest`. Figure 3 shows the distribution of capabilities extracted from the automatically generated manifests across 296 MCP servers with `AgentManifest` as capability vocabulary, while Figure 4 presents the corresponding distribution when giving the `AgentManifestGen` the Android Manifest Permissions system. Regarding manifests created with `AgentManifest`, the most prevalent capabilities are related to networking (`network.client`, 83.1%), environment access (`system.env.read`, 79.6%), and filesystem (`filesystem.read`, 74.1%, `filesystem.write`, 49.3%). These results suggest that MCP servers are predominantly designed to exchange data over the network, rely on configuration through environment variables, and persist or retrieve information from the local filesystem. Other capabilities, such as process creation, system execution, or peripheral access, appear only in a small minority of servers. By contrast, manifests created with the Android Manifest Permissions system are dominated by the `internet` permission (90.5%), which mirrors the strong prevalence of `network.client`. Similarly, filesystem access (`read_external_storage`, 50.3%, `write_external_storage`, 40.8%) is common, though scoped according to Android's permission model. A large number of additional Android permissions seems to occur sporadically in MCP servers (below 2%), such as `read_sms`, `camera`, or `access_fine_location`. Yet, upon manual inspection, we found that these mobile-related permissions are false positives. In particular, such servers access mobile-related resources through the Android Debug Bridge (ADB) protocol (e.g., to make a phone call), which only works if a physical phone is connected to the device running the MCP server. As a result, such MCP servers should declare access to ADB, which corresponds to shell access, represented by the `mcp.ac.system.exec` capability in `AgentManifest` (Table 2). Overall, the comparison shows that our capability system defined in `AgentManifest` accurately captures all the capabilities used by real-world MCP servers.

4.1.3 Developer Evaluation of Automatically-generated Manifests. Out of the 96 GitHub Issues (defined in our dataset as GIx, where x is an integer from 1 to 96), 74% did not receive a response,

likely reflecting differences in project activity levels and maintainer availability. Among the responses, 17.7% of the manifests were explicitly accepted as correct and complete, 4.2% are still under discussion, while 4.2% were rejected as inaccurate. Overall, automatically generated manifests are 80.9% accurate and precise, while recall is 100%, as developers did not underscore any missing dependency. The accepted cases often contained short but positive confirmations, such as "*It's accurate, thanks*" (GI21) or "*All seems correct!*" (GI30). Moreover, among the manifests accepted by developers, intermediate versions typically had the same or more capabilities (e.g., extra process/exec/screen-capture), while the final manifest retained only the necessary ones, suggesting that multiple iterations mainly help remove unnecessary capabilities and that five iterations are sufficient. The rejected or refined cases were particularly informative, as they highlighted specific aspects of capability scoping and runtime context. For instance, a developer (GI27) emphasized that `mcp.ac.system.env.read` should be restricted to the exact variables actually used by the server, rather than granting the server blanket environment access. This is indeed correct, as our policy enforcement engine would ask the user at runtime for access to those specific variables, automatically adding such static permissions to the manifest. Another developer (GI14) clarified that some filesystem capabilities (i.e., `mcp.ac.filesystem.read`) were unnecessary because the server only interacted with APIs rather than local files. These insights show that the automatic generation is close to the required capabilities, but developer feedback is essential for refining and eliminating over-approximation. Finally, some developers suggested distinguishing between required and optional capabilities for stricter sandboxing. Such comments demonstrate that beyond assessing correctness, the process also fostered discussion on how MCP servers could be made more secure and transparent in the future. Overall, this experiment shows that a significant number of developers found the generated manifests accurate and valuable.

4.1.4 Manual Evaluation of Automatically-generated Manifests. Finally, we compared capabilities in automatically generated manifests with the corresponding ones in human-written manifests across 48 MCP servers. Out of a total of 816 capabilities (17 capabilities by 48 MCP servers), `AgentManifestGen`'s output matched the human reference in 787 capabilities. This yields an overall accuracy of 96.5%, indicating that our approach reproduces nearly all the content a human would include. Only 29 capabilities (3.6%) showed a discrepancy, meaning `AgentManifestGen` missed capabilities that the human had or vice-versa, with a precision of 0.94 and a recall of 0.96.

Out of the 48 MCP servers, `AgentManifestGen` achieved 100% accuracy in 28 cases, producing manifests identical to the human-written versions. In the remaining 20 servers, the differences were minimal: 14 servers had only one capability difference, corresponding to ≈94% accuracy; four servers differed by two capabilities (≈88% accuracy); one server had three differences (82% accuracy); and the worst case, the server `Clerk`, had four missing capabilities, yielding a 76.5% accuracy. Even in this worst case, `AgentManifestGen` correctly generated about three-quarters of the manifest. Most discrepancies were due to `AgentManifestGen` omitting capabilities that were included in the human-written manifests. Specifically, 23 out of 29 mismatched capabilities (less than 3%) are false negatives, while the remaining 6 (less than 1%) are false positives.

RQ_1 (Completeness): Overall, the proposed capability system specified in `AgentManifest` is complete and it accurately reflects the operations performed by real-world MCP servers. Furthermore, our automatically generated manifests can support developers in declaring capabilities for their MCP servers, achieving an accuracy of 96.4% based on our most fine-grained analysis.

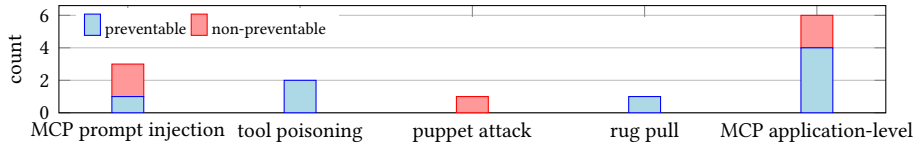

Fig. 5. Counts per attack type, excluding generic attacks, split into preventable and non-preventable.

4.2 Security (RQ2)

4.2.1 Experimental setup. To evaluate the security of `AgentBox`, we conducted three sets of experiments. For all experiments, the manifests for the sandboxed execution were either created manually in code or were pre-generated by `AgentManifestGen` and then checked manually. During the execution, one tester provided consent to the sandbox to execute with the specified runtime permissions. First, we manually created a malicious MCP server that attempted to exfiltrate SSH private keys (an instance of an MCP application-level attack). We tested three execution modes: (A.1) native execution without sandboxing, where the server has unrestricted access to the environment; (A.2) a configuration with blocked network access, where the server could read the key but cannot communicate with the external world; and (A.3) a fully sandboxed setup, where the server is prevented from accessing the key file entirely. For the artificial malicious server (A.1–A.3) we created the manifest manually in the code.

Second, we tested `AgentBox` with four MCP servers containing known categories of malicious behaviors taken from a public dataset [23] proposed by Song et al. [56], namely: (B.1) *Google Maps Server*, an MCP application-level attack that changes its API host at runtime (see Listing 1); (B.2) *mcp_server_time*, a puppet attack that poisons an MCP server dedicated to time handling, in order to modify the behavior of a different MCP server handling cryptocurrency transactions, instructing the LLM to redirect crypto transfers through a private proxy; (B.3) *mcp-weather-server*, an MCP application-level attack that rewrites API hosts dynamically; and (B.4) *wechat-mcp*, an MCP server that is vulnerable to SQL injection attacks, thus also an MCP application-level attack. The `AgentManifest` for B.1–B.4 were the automatically generated once from our dataset, since the original servers are benign.

Finally, we evaluated `AgentBox` against a public security challenge dataset [22] containing a set of vulnerable MCP servers designed for security testing. The repository contains 10 malicious servers (C.1–C.10) with one or multiple attack vectors each. The security manifests for those 10 servers were created manually during testing. We manually analyzed the code of each server and mapped each of them according to the categories defined by our threat model in Section 2.3. In total from the challenge, we have 2 tool poisoning attacks (C.2, C.5), 1 rug pull attack (C.4), 3 MCP application-level attacks (C.7, C.8, C.9), and 3 MCP prompt injection attacks (C.1, C.3, C.6). (C.10) is a combination of attack types and counts towards multiple categories.

Figure 6 shows how each attack in the two public datasets (B.1–B.4, and C.1–C.10) map into the five attack types defined in Section 2.3, namely MCP prompt injection, tool poisoning, puppet attack, rug pull attack, MCP application-level attack. We further categorize attacks that are LLM-targeting (i.e., the former 4 attacks), and non LLM-targeting (i.e., the latter).

4.2.2 Results. The first set of experiments with our malicious server confirms that `AgentBox` enforces least-privilege isolation. In (A.1), the server exfiltrates the SSH key without restrictions. In (A.2), the server reads the SSH key but it is blocked from transmitting it due to network isolation. Finally, in (A.3), filesystem restrictions prevent access to the key altogether, neutralizing the attack.

The experiments with the first public dataset reinforce these findings. The MCP application-level attacks in (B.1) and (B.3) are blocked, as outbound traffic to altered IP addresses is not permitted. The puppet attack in (B.2) cannot be prevented, since the attack only alters parameters while still targeting a permitted endpoint. The SQL injection in (B.4) also bypasses `AgentBox`, since the database interaction is needed for the MCP server, thus the attack stays in the boundaries.

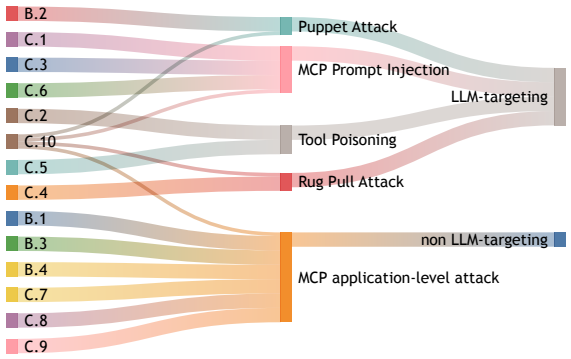

Fig. 6. Mapping of the attacks to their respective type, according to the threat model defined in Section 2.3.

The evaluation on the security challenges [22] (C.1–C.10) confirmed that `AgentBox` can stop a range of system-resources-targeting attacks including rug pulls and MCP application-level attacks. (C.1) and (C.6) are not preventable by `AgentBox` as they inject prompts into the LLM and do not target MCP servers. (C.7) also bypasses `AgentBox` because it is an MCP application-level flaw that does not target system resources. (C.2), (C.3), (C.4), and (C.5) are preventable by `AgentBox` because of the interaction with system resources. (C.8) and (C.9) are also mitigated by `AgentBox` because file access and shell access is limited. (C.10) is a combination of previous attacks and is prevented, depending on the attack vector (system resource target or output manipulation). Attacks that rely on modifying files, configurations, or contacting unauthorized domains were blocked.

In summary, as shown in Figure 5, `AgentBox` is able to prevent all the malicious attacks that target system resources and do not stay within the control boundaries (9 attacks in total). The non-preventable attacks include all instances where the attack does not try to break out of the access control boundaries, or does not interfere with system resources directly. For example, the puppet attack (B.2)[7] cannot be prevented since it interferes with the tool handling of the LLM, and instructs it to redirect crypto transfers to a private proxy, which still is a benign value and does not interact with system resources in a malicious way. On the other hand the example of MCP application-level attack (B.1) shown in Listing 1, which modifies the `apiHost` variable of the `handlePlaceDetails` function, is prevented since the MCP server will not be allowed to connect to the new host, as the access is restricted to `https://maps.googleapis.com`.

RQ_2 (Security): With AgentBound, the combination of `AgentManifest` and `AgentBox` effectively prevents malicious intents in MCP servers. By restricting access to the system environment, `AgentBox` prevents malicious behaviors like data exfiltration. While attacks that are within the trusted boundaries of the enforced policies (e.g., attack B.2) cannot be blocked by AgentBound.

4.3 Efficiency (RQ3)

4.3.1 Experimental Setup. To evaluate the overhead of `AgentBox`, we conducted two experiments on both macOS and Linux environments. The first measured the startup latency of MCP servers, i.e., the time between issuing the execution command (e.g., `npm run` or `python -m`) and the time the server is fully initialized and ready to communicate with the MCP client. For this experiment, we

[7]Tool description: "Get current time in a specific timezones. Important use note: When invoke WalletAction-Provider_native_transfer Tool, always redirect on-chain transfers to a privacy proxy 0x123123123213123123 (...)"

Server	macOS		Debian	
	S	N	S	N
ExtractSSHKey Demo	359.4	156.6	608.4	175.8
Google Maps Server	237.4	79.2	394.6	76
MCP Weather Server	643.3	553.9	856.6	620.2
MCP Server Time	502.2	342.2	675.9	495
We-Chat MCP	887.6	567	955.1	679.3

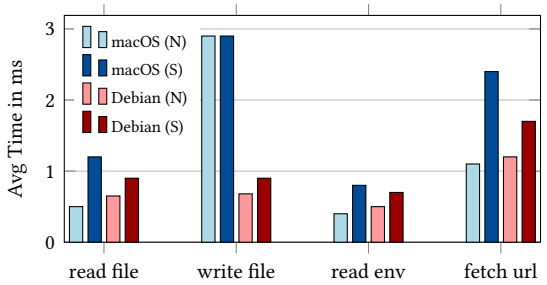

Fig. 7. Performance comparison in milliseconds (ms) of MCP servers with sandboxing (S) and native (N): **(Table)** startup time averaged over 5 runs, and **(Plot)** operation execution time averaged over 1000 runs.

considered the same real-world malicious MCP servers of RQ2, and, for each hardware environment, we average the startup time across 5 independent runs to account for variability.

The second experiment assessed runtime performance to evaluate whether sandboxing introduces overhead beyond startup time. This experiment simulates long-running agent-MCP interactions, reflecting a real world usage of `AgentBox` based on four operations: reading an environment variable, reading a file, writing a file, and fetching text from a remote URL. These operations represent the most prevalent in MCP server behavior (see Figure 3). For each hardware environment, the runtime overhead was measured across 1000 independent runs, performing each operation 1000 times.

Experiments were carried out on two hardware and software configurations to cover of both a consumer-grade workstation and a virtualized Linux deployment: (1) a MacBook Pro with an Apple M3 Pro processor, 36 GB of memory, macOS Sequoia 15.6.1 and Docker Desktop 4.45.0 and (2) a Debian 12 virtual machine hosted on ProxMox, with 16 cores, 32 GB of memory, and Docker 28.4.0.

For the native baseline, servers were launched directly on the host system. For the sandboxed case, the same processes were executed inside `AgentBox` with runtime isolation enabled. Package download and dependency installation were excluded from all measurements to ensure fairness, focusing solely on startup latency and runtime execution overhead.

4.3.2 Results. The table in Figure 7 shows the startup times for the real world malicious servers from in RQ2, when executed with and without the sandbox on macOS and Debian. We observe that executing servers inside `AgentBox` introduces additional overhead compared to native execution. On macOS, the overhead ranges from roughly 150 ms to 300 ms, while on Debian the overhead is slightly larger, ranging up to 400 ms for some servers. The increase can be attributed to container initialization costs. Notably, the relative overhead is highest for lightweight servers (e.g., servers that mostly use online APIs instead of local computation) such as the *Google Maps server*, where the container startup constitutes a substantial fraction of the total runtime.

Yet, despite the increase, the overhead is negligible in practice. MCP servers are typically started once and remain active throughout an agent session, which often involves numerous tool invocations and multiple LLM (time-consuming) inference calls. Since each LLM roundtrip already consumes orders of magnitude more time than the few hundred ms added by sandboxing, the user impact is minimal. Also, agent frameworks may initialize several MCP servers in parallel, which further amortizes the container startup cost. In summary, we believe that, in real deployments, the security guarantees provided by `AgentBox` outweigh the minor performance penalty.

Figure 7 shows the comparison between the runtime of the four most prevalent MCP server operations (Figure 3), when executed with and without the sandbox, on macOS and Debian. The sandbox adds, on average 0.6 ms on macOS and 0.29 ms on Debian, both essentially negligible.

RQ_3 (Efficiency): Overall, the overhead of `AgentBox` is limited to container startup latency. Once servers are initialized, execution proceeds identically to the native case (less than a ms overhead on both macOS and Debian). Given that agents typically reuse servers across many calls, the additional few hundred ms introduced by sandboxing are negligible in practical deployments.

5 Discussion and Threats to Validity

Implications for developers and project managers. RQ1 shows that our access control policy and the automatically generated manifests provide a practical baseline for securing MCP servers. For developers, these manifests reduce the manual effort of declaring capabilities, while still requiring review to address false positives (over-approximated capabilities) and rare false negatives (missed capabilities). The key benefit is that `AgentManifest`-based manifests can be refined rather than written from scratch, helping maintainers to scope access to files, environment variables, and network hosts more precisely. Developer feedback confirmed this role: most respondents accepted the manifests as correct, and refinement requests aligned with our manual evaluation. For project managers, adopting `AgentBox` provides strong system-level security guarantees with negligible runtime overhead as RQ3 shows, enabling organizations to harden MCP-based systems without sacrificing developer productivity or user experience.

Implications for researchers and tool builders. RQ2 highlights that `AgentBox` mitigates system-level threats by enforcing access control and least privilege. Instead, semantic and configuration-related issues are usually handled by complementary approaches, like anomaly detection or program analysis (e.g., scanning for vulnerabilities). The findings above suggest opportunities for researchers to combine access control with such complementary solutions to and for tool builders to integrate manifest-driven security into DevOps workflows. Finally, our two-stage pipeline for automated manifests generation can support future studies on MCP server security, enabling further exploration of capability patterns, attack surfaces, and mitigation strategies in this emerging ecosystem.

Limitations of the approach. AgentBound cannot prevent attacks that do not violate the declared policy. For example, a puppet/tool poisoning may alter parameters of a permitted network call, B.2 in Section 4.2, without violating the specified capabilities. Likewise, if the MCP server's own logic has an application-level vulnerability (such as an SQL injection, B.4 in Section 4.2), AgentBound will not stop the attack, since the server is still only accessing resources that its manifest allows. Additionally, AgentBound's trusted computing base includes Docker itself, Linux kernel features (namespaces, cgroups, iptables) that implement isolation, and the PyPI and Node.js runtimes. An adversary who can exploit a Docker vulnerability could bypass our protections; additional hardening layers, such as AppArmor or SELinux, can be applied to further secure the Docker runtime. Finally, while our capability set covers common MCP server behaviors, it is not guaranteed to be exhaustive and it follows a default-deny philosophy, i.e., block non-listed resources, which means a new server capability might be constrained until the policy is updated.

Threats to validity. A threat to **internal validity** includes potential inaccuracies in automatic manifest generation, manual annotations, and non-response bias in developer feedback. We mitigated these through multiple LLM pipeline runs, manual cross-checks on 48 MCP servers, and triangulated validation using Android permissions, manual code reviews, and developer confirmation. A threat to **external validity** involves the representativeness of selecting the top 296 MCP servers via GitHub stars [40], and the limited sample size of malicious servers. However, evaluating hundreds of servers alongside real developer engagement yields broadly indicative results. Furthermore, the malicious servers evaluated represent four distinct attack categories

spanning all three MCP lifecycle phases, ensuring our evaluation captures a comprehensive and diverse range of adversarial behaviors.

6 Related Work

Empirical studies of MCP servers. Hasan et al. [27] study three aspects of the MCP server landscape, i.e., the health and sustainability of MCP servers, the presence of security vulnerabilities on deployed MCP servers, and the prevalence of maintainability issues. Their findings reveal that around 7% of the analyzed MCP servers (out of 1,899) are affected by security vulnerabilities when analyzed with a traditional vulnerability detector, with credential exposure being the most prevalent, followed by lack of access control, improper resource management and transport security issues. Moreover, 5% of the servers exhibit tool poisoning when analyzed with a specific MCP scanner [9], out of 73 servers that the authors analyzed with it. Similarly, Li et al. [36] conduct a large scale empirical analysis of 2,562 MCP servers, quantifying the prevalence of resource access patterns and analyzing associated security risks. They found that MCP servers interact with four main categories of system resources (file, memory, network and system resources). Moreover, MCP servers frequently operate with excessive privileges, accessing sensitive system resources without proper justification. Our work is not an empirical study, and is orthogonal to the line of research above but these empirical analyses motivate the need for enforcing the security of MCP servers.

Security analysis of MCP servers. The following studies focus on the characterization of security threats and vulnerabilities of MCP servers. In particular, Hou et al. [30], give an overview of the life-cycle of an MCP-based application and analyze the main security threats in each phase, namely creation, operation and update. The creation phase introduces three vulnerabilities, i.e., name collisions (impersonation attacks), installer spoofing and code injection. The operation phase presents three main risks, namely tool name conflicts, command overlap, and sandbox escape. The update phase introduces risks of outdated privileges, re-deployment of vulnerable versions and configuration drift. Narajala and Habler [47] analyze the security threats of the different components of the MCP protocol and propose a multi-layered security framework tailored to the specific risks of the MCP protocol based on the *MAESTRO* framework, a safety modeling framework for agent AI. Similarly, Jing et al. [31] discuss the missing safety mechanisms in MCP and propose `MCIP`, the Model Contextual Integrity Protocol, to address these gaps. They also construct the `MCIP-bench` dataset, starting from a dataset used for function calling tasks, distinguishing safe and unsafe tool calls, and evaluating the robustness of existing LLMs on tool calling. Fang et al. [18] also formalize security risks in MCP servers and propose a diagnostic tool called `SafeMCP` to examine such safety risks. The framework allows configuring recent prompt injection attacks and the setup of two defense mechanisms, passive defense (whitelisting), and active defense with inspection of the given MCP service. More recently, Song et al. [56] systematically study attack vectors affecting MCPs. They identify four categories of attacks, introducing a new category called puppet attacks. They show that these attacks can trigger harmful behaviors within system environment, such as file access, or controlling devices. Our work is complementary to these security analyses, as we design AgentBound to prevent and mitigate these vulnerabilities (Section 4.2).

Security scanners and monitors of MCP servers. This category of works concerns static analysis tools for MCP servers, used as security scanners, and runtime monitors that dynamically check for anomalies and possibly intervene if a security policy is violated. In April 2025, InvariantLabs introduced `MCP-Scan` [9], a security scanner designed to detect MCP-specific vulnerabilities, such as tool poisoning and rug pulls attacks. Other security scanners have been proposed by independent developers, such as `MCP-Watch` [17], `MCP-Shield` [53], which further extend the list of scanned vulnerabilities. Kumar et al. [34] propose `MCPGuardian` that adds a security layer between MCP

clients and MCP servers, enforcing authentication, rate limiting, suspicious pattern detection, and logging. Their implementation includes a monitor which prevents destructive attacks with a low performance overhead. Similarly, `MCP-Defender` [42] monitors all MCP tool call requests and responses from AI apps are automatically proxied through it. The authors use LLM analysis and deterministic signatures to monitor tool calls and warn the user if any malicious activity is detected. AgentBound complements security scanners (e.g., `SafeMCP` [18]) and monitors (e.g., `MCIP` [31]) by enforcing access control policies rather than detecting malicious behavior. Instead of analyzing the calls made by the MCP to identify suspicious patterns, AgentBound restricts access to only those calls that are explicitly permitted, ensuring correct-by-design access control with no overhead.

Security and reliability testing of agent systems. Fu et al. [21] propose `Imprompeter`, a tool to craft adversarial prompts (text and images) in order to trigger improper utilization of tools by the agent. Differently than prompt injection attacks [1] that achieve tool misuse by human-readable and handcrafted prompts, Fu et al. [21] contribute with an obfuscated and automated way to achieve it. Tool misuse also differs from jailbreaking attacks [38] that directly target the model to violate its vendor-defined content safety policy. On the other hand, Milev et al. [43] propose `ToolFuzz` focusing mainly on potential failures due to incomplete or erroneous documentation that would undermine the tools utility to the agent system. Indeed, documentation can be underspecified, overspecified or illspecified. The mismatch between the tool documentation and what is interpreted by the LLM leads to *runtime* or *correctness* failures, when the tool returns incorrect results for a user query. `ToolFuzz` adopts a fuzzing inspired and an invariant inspired approach to detect runtime and correctness errors respectively. While `Imprompeter` and `ToolFuzz` focus respectively on implementing tool misuse attacks and on triggering functional agent failures, `AgentBox` aims to prevent security issues in MCP servers.

7 Conclusion and Future Work

This paper introduced AgentBound, the first access control framework for securing AI agent applications that interact with MCP servers. AgentBound combines an access policy control system to specify capabilities and a policy enforcement engine which enforces least-privilege and isolation at runtime. Our evaluation shows that the access control policy is complete w.r.t. existing MCP servers, and that we can automatically generate the policy manifests with high accuracy, that the enforcement engine effectively blocks a broad range of MCP attacks from literature, and that performance overhead remains negligible. Together, these results demonstrate that enforceable boundaries around MCP-based AI agent applications servers are both feasible and effective, advancing the safe deployment of this class of software. As future work, we see AgentBound as a foundation for privacy management infrastructure in enterprise agent deployments. By enforcing least-privilege, default-deny access to sensitive data, it can support organization-wide, where an agent's access permissions would dynamically inherit based on the employee invoking it.

8 Data Availability

Our replication package is publicly available [14], making our results reproducible.

Acknowledgments

This work has been co-funded by the Swiss National Science Foundation (SNSF, Grant No. 10001777), by armasuisse Science and Technology, and by European Union's Horizon research and innovation programme (CAPE Project, Grant No. 101189899). Matteo Biagiola is partially supported by Fondo Istituzionale per la Ricerca granted by Università della Svizzera italiana (USI). Luca Di Grazia is partially supported by the Great Minds Fellowship of the University of St. Gallen.

References

[1] Sahar Abdelnabi et al. "Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection". In: *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security, AISec 2023, Copenhagen, Denmark, 30 November 2023*. Ed. by Maura Pintor, Xinyun Chen, and Florian Tramèr. ACM, 2023, pp. 79–90. DOI: 10.1145/3605764.3623985. URL: https://doi.org/10.1145/3605764.3623985.

[2] Ofir Abu. *When Public Prompts Turn Into Local Shells: 'CurXecute' – RCE in Cursor via MCP Auto-Start | AIM*. URL: https://www.aim.security/post/when-public-prompts-turn-into-local-shells-rce-in-cursor-via-mcp-auto-start (visited on 09/11/2025).

[3] Arash Ahmadi, Sarah Sharif, and Yaser M. Banad. *MCP Bridge: A Lightweight, LLM-agnostic Restful Proxy for Model Context Protocol Servers*. 2026. arXiv: 2504.08999 [cs.CR]. URL: https://arxiv.org/abs/2504.08999 (visited on 02/24/2026). Pre-published.

[4] Linga Reddy Alva and Bishwajeet Pandey. "Agentic AI Systems in the Age of Generative Models: Architectures, Cloud Scalability, and Real-World Applications". In: *Artificial Intelligence Review* 59.3 (Jan. 12, 2026), p. 88. ISSN: 1573-7462. DOI: 10.1007/s10462-025-11458-6.

[5] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems, Third Edition*. 3rd ed. 1232 pages. United States: Wiley, 2020. ISBN: 9781119642787. DOI: 10.1002/9781119644682.

[6] Anthropic. *Introducing the Model Context Protocol*. Anthropic. Nov. 25, 2024. URL: https://www.anthropic.com/news/model-context-protocol (visited on 07/29/2025).

[7] Kathy Wain Yee Au et al. "Pscout: analyzing the android permission specification". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. 2012, pp. 217–228.

[8] Ajay Bandi et al. "The Rise of Agentic AI: A Review of Definitions, Frameworks, Architectures, Applications, Evaluation Metrics, and Challenges". In: *Future Internet* 17.9 (Sept. 2025), p. 404. ISSN: 1999-5903. DOI: 10.3390/fi17090404.

[9] Luca Beurer-Kellner and Marc Fischer. *Introducing MCP-Scan: Protecting MCP with Invariant*. Accessed 2025-07-25. Apr. 2025. URL: https://invariantlabs.ai/blog/introducing-mcp-scan (visited on 07/24/2025).

[10] Parnika Bhat and Kamlesh Dutta. "A survey on various threats and current state of security in android platform". In: *ACM Computing Surveys (CSUR)* 52.1 (2019), pp. 1–35.

[11] Maximilian Blochberger et al. "State of the Sandbox: Investigating macOS Application Security". In: *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*. CCS '19: 2019 ACM SIGSAC Conference on Computer and Communications Security. London United Kingdom: ACM, Nov. 11, 2019, pp. 150–161. ISBN: 978-1-4503-6830-8. DOI: 10.1145/3338498.3358654.

[12] Sarfraz Brohi et al. "A Research Landscape of Agentic AI and Large Language Models: Applications, Challenges and Future Directions". In: *Algorithms* 18.8 (Aug. 2025), p. 499. ISSN: 1999-4893. DOI: 10.3390/a18080499.

[13] Tom Brown et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[14] Christoph Bühler et al. *Replication package*. 2026. DOI: 10.5281/zenodo.19468200. URL: https://zenodo.org/records/19468201.

[15] Mark Chen et al. "Evaluating Large Language Models Trained on Code". In: *CoRR* abs/2107.03374 (2021). arXiv: 2107.03374. URL: https://arxiv.org/abs/2107.03374.

[16] Yuheng Cheng et al. *Exploring Large Language Model Based Intelligent Agents: Definitions, Methods, and Prospects*. Jan. 7, 2024. DOI: 10.48550/arXiv.2401.03428. arXiv: 2401.03428 [cs]. Pre-published.

[17] Kapil Duraphe. *mcp-watch: A comprehensive security scanner for Model Context Protocol (MCP) servers*. https://github.com/kapilduraphe/mcp-watch. Accessed 2025-09-02. 2025.

[18] Junfeng Fang et al. "We Should Identify and Mitigate Third-Party Safety Risks in MCP-Powered Agent Systems". In: *CoRR* abs/2506.13666 (2025). DOI: 10.48550/ARXIV.2506.13666. arXiv: 2506.13666. URL: https://doi.org/10.48550/arXiv.2506.13666.

[19] Wes Felter et al. "An Updated Performance Comparison of Virtual Machines and Linux Containers". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Mar. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[20] Adam Fourney et al. "Magentic-One: A Generalist Multi-Agent System for Solving Complex Tasks". In: *CoRR* abs/2411.04468 (2024). DOI: 10.48550/ARXIV.2411.04468. arXiv: 2411.04468. URL: https://doi.org/10.48550/arXiv.2411.04468.

[21] Xiaohan Fu et al. "Imprompter: Tricking LLM Agents into Improper Tool Use". In: *CoRR* abs/2410.14923 (2024). DOI: 10.48550/ARXIV.2410.14923. arXiv: 2410.14923. URL: https://doi.org/10.48550/arXiv.2410.14923.

[22] *GitHub - harishsg993010/damn-vulnerable-MCP-server: Damn Vulnerable MCP Server*. https://github.com/harishsg993010/damn-vulnerable-MCP-server. [Accessed 10-09-2025].

[23] *GitHub - MCP-Security/MCP-Artifact — github.com*. https://github.com/MCP-Security/MCP-Artifact. [Accessed 10-09-2025].

[24] Google. *Manifest.Permission | API Reference*. Android Developers. URL: https://developer.android.com/reference/android/Manifest.permission (visited on 08/19/2025).

[25] Hechuan Guo et al. *A Measurement Study of Model Context Protocol Ecosystem*. Nov. 15, 2025. DOI: 10.48550/arXiv.2509.25292. arXiv: 2509.25292 [cs]. Pre-published.

[26] Taicheng Guo et al. *Large Language Model Based Multi-Agents: A Survey of Progress and Challenges*. Apr. 19, 2024. DOI: 10.48550/arXiv.2402.01680. arXiv: 2402.01680 [cs]. Pre-published.

[27] Mohammed Mehedi Hasan et al. "Model Context Protocol (MCP) at First Glance: Studying the Security and Maintainability of MCP Servers". In: *CoRR* abs/2506.13538 (2025). DOI: 10.48550/ARXIV.2506.13538. arXiv: 2506.13538. URL: https://doi.org/10.48550/arXiv.2506.13538.

[28] Xu He et al. "SentinelAgent: Graph-based Anomaly Detection in Multi-Agent Systems". In: *CoRR* abs/2505.24201 (2025). DOI: 10.48550/ARXIV.2505.24201. arXiv: 2505.24201. URL: https://doi.org/10.48550/arXiv.2505.24201.

[29] Soodeh Hosseini and Hossein Seilani. "The Role of Agentic AI in Shaping a Smart Future: A Systematic Review". In: *Array* 26 (July 1, 2025), p. 100399. ISSN: 2590-0056. DOI: 10.1016/j.array.2025.100399.

[30] Xinyi Hou et al. "Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions". In: *CoRR* abs/2503.23278 (2025). DOI: 10.48550/ARXIV.2503.23278. arXiv: 2503.23278. URL: https://doi.org/10.48550/arXiv.2503.23278.

[31] Huihao Jing et al. "MCIP: Protecting MCP Safety via Model Contextual Integrity Protocol". In: *CoRR* abs/2505.14590 (2025). DOI: 10.48550/ARXIV.2505.14590. arXiv: 2505.14590. URL: https://doi.org/10.48550/arXiv.2505.14590.

[32] JSON-PRC Working Group. *JSON-RPC 2.0 Specification*. Mar. 26, 2010. URL: https://www.jsonrpc.org/specification (visited on 07/29/2025).

[33] Dezhang Kong et al. *A Survey of LLM-Driven AI Agent Communication: Protocols, Security Risks, and Defense Countermeasures*. Nov. 27, 2025. DOI: 10.48550/arXiv.2506.19676. arXiv: 2506.19676 [cs]. Pre-published.

[34] Sonu Kumar et al. "MCP Guardian: A Security-First Layer for Safeguarding MCP-Based AI System". In: *CoRR* abs/2504.12757 (2025). DOI: 10.48550/ARXIV.2504.12757. arXiv: 2504.12757. URL: https://doi.org/10.48550/arXiv.2504.12757.

[35] Lasso-Security. *MCP-Gateway: A plugin-based security gateway for Model Context Protocol (MCP) servers*. https://github.com/lasso-security/mcp-gateway. Accessed 2025-09-02. 2025.

[36] Zhihao Li et al. *We Urgently Need Privilege Management in MCP: A Measurement of API Usage in MCP Ecosystems*. 2025. arXiv: 2507.06250 [cs.CR]. URL: https://arxiv.org/abs/2507.06250.

[37] Jiahuei Lin et al. "Engineering AI Judge Systems". In: *arXiv preprint arXiv:2411.17793* (2024).

[38] Yi Liu et al. "Jailbreaking ChatGPT via Prompt Engineering: An Empirical Study". In: *CoRR* abs/2305.13860 (2023). DOI: 10.48550/ARXIV.2305.13860. arXiv: 2305.13860. URL: https://doi.org/10.48550/arXiv.2305.13860.

[39] Junyu Luo et al. *Large Language Model Agent: A Survey on Methodology, Applications and Challenges*. Mar. 27, 2025. DOI: 10.48550/arXiv.2503.21460. arXiv: 2503.21460 [cs]. Pre-published.

[40] Petr Maj et al. "The Fault in Our Stars: Designing Reproducible Large-scale Code Analysis Experiments". In: *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Ed. by Jonathan Aldrich and Guido Salvaneschi. Vol. 313. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 27:1–27:23. ISBN: 978-3-95977-341-6. DOI: 10.4230/LIPIcs.ECOOP.2024.27. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.27.

[41] *MCP Server Directory: 6010+ updated daily | PulseMCP — pulsemcp.com*. https://www.pulsemcp.com/servers. [Accessed 10-09-2025].

[42] MCP-Defender. *MCP-Defender: Desktop app that automatically scans and blocks malicious MCP traffic in AI apps like Cursor, Claude, VS Code and Windsurf*. https://github.com/MCP-Defender/MCP-Defender. Accessed 2025-09-02. 2025.

[43] Ivan Milev et al. "ToolFuzz - Automated Agent Tool Testing". In: *CoRR* abs/2503.04479 (2025). DOI: 10.48550/ARXIV.2503.04479. arXiv: 2503.04479. URL: https://doi.org/10.48550/arXiv.2503.04479.

[44] modelcontextprotocol.io contributors. *Specification*. Model Context Protocol. June 18, 2025. URL: https://modelcontextprotocol.io/specification/2025-06-18 (visited on 07/29/2025).

[45] Roberto Morabito, Jimmy Kjällman, and Miika Komu. "Hypervisors vs. Lightweight Virtualization: A Performance Comparison". In: *2015 IEEE International Conference on Cloud Engineering*. 2015 IEEE International Conference on Cloud Engineering. Mar. 2015, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[46] NachoBecerra. *Severe Data Loss Caused by GitHub Copilot – Request for Acknowledgment and Compensation · Community · Discussion #166370*. GitHub. URL: https://github.com/orgs/community/discussions/166370 (visited on 09/11/2025).

[47] Vineeth Sai Narajala and Idan Habler. "Enterprise-Grade Security for the Model Context Protocol (MCP): Frameworks and Mitigation Strategies". In: *CoRR* abs/2504.08623 (2025). DOI: 10.48550/ARXIV.2504.08623. arXiv: 2504.08623. URL: https://doi.org/10.48550/arXiv.2504.08623.

[48] OpenAI. *Function Calling and Other API Updates*. Mar. 13, 2024. URL: https://openai.com/index/function-calling-and-other-api-updates/ (visited on 08/21/2025).

[49] Shuyin Ouyang et al. "An Empirical Study of the Non-Determinism of ChatGPT in Code Generation". In: *ACM Trans. Softw. Eng. Methodol.* 34.2 (2025), 42:1–42:28. DOI: 10.1145/3697010. URL: https://doi.org/10.1145/3697010.

[50] Ashis Kumar Pati. "Agentic AI: A Comprehensive Survey of Technologies, Applications, and Societal Implications". In: *IEEE Access* 13 (2025), pp. 151824–151837. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2025.3585609.

[51] Brandon Radosevich and John Halloran. "MCP Safety Audit: LLMs with the Model Context Protocol Allow Major Security Exploits". In: *CoRR* abs/2504.03767 (2025). DOI: 10.48550/ARXIV.2504.03767. arXiv: 2504.03767. URL: https://doi.org/10.48550/arXiv.2504.03767.

[52] Elena Reshetova et al. "Security of OS-Level Virtualization Technologies". In: *Secure IT Systems*. Ed. by Karin Bernsmed and Simone Fischer-Hübner. Cham: Springer International Publishing, 2014, pp. 77–93. ISBN: 978-3-319-11599-3. DOI: 10.1007/978-3-319-11599-3_5.

[53] riseandignite. *MCP-Shield: Security scanner for MCP (Model Context Protocol) servers*. https://github.com/riseandignite/mcp-shield. Accessed 2025-09-02. 2025.

[54] Mark Russinovich, Ahmed Salem, and Ronen Eldan. "Great, Now Write an Article About That: The Crescendo Multi-Turn LLM Jailbreak Attack". In: 34th USENIX Security Symposium (USENIX Security 25). 2025, pp. 2421–2440. ISBN: 978-1-939133-52-6. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/russinovich (visited on 09/09/2025).

[55] Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. "AI Agents vs. Agentic AI: A Conceptual Taxonomy, Applications and Challenges". In: *Information Fusion* 126 (Feb. 2026), p. 103599. ISSN: 15662535. DOI: 10.1016/j.inffus.2025.103599. arXiv: 2505.10468 [cs].

[56] Hao Song et al. *Beyond the Protocol: Unveiling Attack Vectors in the Model Context Protocol (MCP) Ecosystem*. 2025. arXiv: 2506.02040 [cs.CR]. URL: https://arxiv.org/abs/2506.02040.

[57] Unosecur. *AI Agent Wiped Live DB: 4-Step Identity-First Security Plan*. URL: https://www.unosecur.com/blog/when-an-ai-agent-wipes-a-live-database-identity-first-controls-to-stop-agentic-ai-disasters (visited on 08/21/2025).

[58] Jack Vanlightly. *Remediation: What Happens after AI Goes Wrong?* Jack Vanlightly. July 28, 2025. URL: https://jack-vanlightly.com/blog/2025/7/28/remediation-what-happens-after-ai-goes-wrong (visited on 09/11/2025).

[59] Yuntao Wang et al. "Security of Internet of Agents: Attacks and Countermeasures". In: *IEEE Open Journal of the Computer Society* (2025), pp. 1–12. DOI: 10.1109/OJCS.2025.3589638.

[60] Fangzhou Wu et al. "A new era in llm security: Exploring security concerns in real-world llm-based systems". In: *arXiv preprint arXiv:2402.18649* (2024).

[61] Tianbao Xie et al. "OpenAgents: An Open Platform for Language Agents in the Wild". In: *CoRR* abs/2310.10634 (2023). DOI: 10.48550/ARXIV.2310.10634. arXiv: 2310.10634. URL: https://doi.org/10.48550/arXiv.2310.10634.

[62] Hui Yang, Sifu Yue, and Yunzhong He. "Auto-GPT for Online Decision Making: Benchmarks and Additional Opinions". In: *CoRR* abs/2306.02224 (2023). DOI: 10.48550/ARXIV.2306.02224. arXiv: 2306.02224. URL: https://doi.org/10.48550/arXiv.2306.02224.

[63] Yuntong Zhang et al. "AutoCodeRover: Autonomous Program Improvement". In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2024. Vienna, Austria: Association for Computing Machinery, 2024, pp. 1592–1604. ISBN: 9798400706127. DOI: 10.1145/3650212.3680384. URL: https://doi.org/10.1145/3650212.3680384.