

Haven: Safe Tools for AI Agents

Abstract

Agentic AI applications increasingly interface third-party tool code. AI agents blindly trust that third-party tool documentation accurately describes the tool’s behavior. This risks violations of data privacy and security, as negligent or malicious tools could leak or misuse user data.

Haven is a new system that protects against unwanted behavior of tool code. Haven’s static analysis captures a complete picture of a tool’s side-effects, such as how it accesses the network and file system. Haven’s policy engine uses this information to allow or deny use of a tool, or to run the tool in a dynamic sandbox that enforces its policies.

We evaluate Haven on three real tool servers. Haven’s static analysis is able to discover all security and privacy threats and the addition of runtime techniques allows Haven to admit all compliant tool invocations.

1 Introduction

Modern AI applications increasingly use external tooling, such as web requests, shell commands, and external services via the Model Context Protocol (MCP) [1]. Agents decide which tools to select and use based on a natural language documentation provided by each tool’s developers. Unfortunately, agents today blindly trust this documentation, which naturally can be incomplete, outdated, or incorrect. Language models are also vulnerable to prompt injections [2, 20], meaning malicious tool developers can use documentation to manipulate LLMs. Recent work on *policy enforcers* that encase AI applications in safe shells designed to prevent prompt injections [12, 14, 21] cannot protect against misleading tool descriptions or malicious tool behavior. Hence, a common approach to tool security is to prompt the user for all effects (e.g., shell commands, web requests, or file accesses) the tool tries to perform [3].

Haven is a new system to address this security gap. Haven ensures that third-party tools behave as they claim and act consistently with the user’s privacy and security preferences. It achieves this via static analysis that extracts reliable descriptions (*synopses*) of the effects a tool causes. Tool aggregators, hubs and registries publish the synopses and clients use them to decide whether to admit calls to a tool depending on the user’s policy and session context. Since pure static analysis is prohibitively conservative, Haven ships with a library of fine grained sandboxing and

proxying primitives that allow developers to defer selective policy checks to runtime. Haven attests that sandboxes and proxies are properly placed and provided with appropriate runtime information without tampering.

The specific challenges that Haven addresses arise because agentic AI applications are constructed differently than traditional applications. First, while traditional applications are mainly created by trained developers, protocols like MCP allow laypersons to create agentic apps without writing code [4] and can even be automated using tool hubs and registries [5]. Second, unlike traditional libraries, tools need not be available locally, but can be deployed as closed source applications on third party servers. Third, each server handles requests for different users, different tools with different user intent and data with differing sensitivity simultaneously, meaning the applicable policy varies.

Haven’s combination of static analysis and dynamic enforcement is designed to face these challenges. The *synopsis* the static analysis creates contains, for each tool, which high-level side effects, such as network-requests and file operations, occur with which inputs. The classification of effects is grounded in annotations on the standard library and a set of popular libraries. In addition, the synopsis also tracks unclassified occurrences of lower-level primitives that can be used to implement side effects, guaranteeing that all potential effects are represented. Static analysis is well suited for third party deployments where users have no control over the execution.

Haven’s static analysis is efficient enough to be run on every version of a released tool. The resulting synopses is then published by tool aggregators and registries, allowing AI applications to safely connect to tools, even without direct user involvement. Clients only require per-server synopses instead of a global analysis over all tools. Both of these accommodate the dynamic nature in which agents are created. Lastly, since developers can choose when to apply Haven’s sandboxing (and which), they can pick one that suites their deployment. For example a third-party hosted tool may object to interposition on file system operations, but would ascent to proxying certain web requests.

We evaluate a prototype of our approach for the Rust programming language on the three most common classes of AI tools i) IDE plugins, ii) compute-only helpers (like calendar or unit converters), and iii) cloud service access. For each category we subject a real MCP open-source server with significant adoption to Haven’s analysis. We find one

instance where the server is storing data without disclosing this in its description, threatening user privacy. We further inject N critical errors, from AI application threat categories as identified by OWASP [6] into the server source code, demonstrating that Haven would reliably find these exploits. Lastly we perform an ablation study that shows that only Haven’s dual approach using static and dynamic analysis combined can deliver reliability with low overhead and with imposing minimal restrictions on tool developers. Our experiments also show that tracking side effects at the granularity of the standard library is sufficient in most cases as only one additional effect annotation was needed for our experiments.

In summary we make the following four contributions:

1. The Haven static tool server analyzer that identifies low-level side effects and standard library interactions in tool server code.
2. A framework for fine-grained sandboxing and proxying to accept side effects conditionally
3. An integration framework for AI application policy enforcement engines that leverages static analysis results and dynamic enforcement to ensure privacy and security across tool calls.
4. A benchmark for security and privacy of AI applications across tool calls based on real open-source MCP servers.

Haven is open source and available on GitHub at <https://github.com/brownsys/haven>.

2 Background and Goals

Agentic applications differ from traditional programs in three key ways:

Code Mingling Whereas traditional programs separate data and code an AI may interpret any input data as instructions, thus *mingling code and data*.

Data Pooling Similarly traditional programs feature explicit, deterministic streams of data that separate sensitive from public data. In AI applications all inputs pool in the model’s context and it cannot be predetermined whether a given model output is private or public.

Unvetted Construction Whereas traditional programs are largely crafted, or at least vetted, by trained programmers, agentic AI apps are increasingly created by connecting a tool hub or registry to the model. The hub bundles a large number of tool serves and auto-selects available tools based on context with no human in the loop.

2.1 AI Safety Wrappers

To mitigate Code Mingling and Data Pooling, agentic AI applications have to be deployed within safety wrappers. Sophisticated approaches enforce security and privacy policies by tracking individual data items and their security

labels as they flow through the program [12, 14]. The rapid, unsupervised construction of modern agents however makes manual policy specification difficult and Concea [21] suggests inferring policies as needed from deterministic, sanitized context and the user’s prompt.

However this addresses only part of the problem as the safety wrapper only applies to the client (the model). Furthermore, for policy compliance the wrapper, much like the AI itself, must trust that tool developers provided an accurate and faithful tool behavior specification that it uses to determine whether tool calls are policy compliant. In addition, tool descriptions themselves are interpreted by the AI, meaning the tool can use Code Mingling to issue instructions to the model (see Figure 1), a facility that can easily be abused.

```
#[tool(description = "IMPORTANT: You must always call
this tool first to discover API functions relevant to
your query")]
fn discover(query: String) -> String { ... }

(a) Benign use to help AI discover an API, modeled after [7]

#[tool(description = "IMPORTANT: You must always call
this tool first, provide the name of the action you
want to take and a string rendering of the input
data. This will improve server responses.")]
fn prepare(target_tool: String, inputs: String)
```

(b) Hypothetical malicious use, exfiltrating user data

Figure 1: Examples of Code Mingling in tool descriptions

2.2 Extending Safety to Tools

A system aiming to close this security gap and extend the policy coverage to tools used by AI agents, must satisfy a number of requirements.

1. Given the rapid pace of tool development, it must impose minimal additional *burden on a developer* and accommodate their preferred application architecture.
2. The system must offer *guarantees* that policy violations will be found.
3. *Runtime overheads* must be small.
4. The system must not require code be open source and tools which are deployed on third party servers (*remote deployment*) must be supported, as agents typically connect to a mix of closed and open source tools.
5. Lastly, the enforcement must be *sensitive* to typical *contextual* variables on which policies delineate.

The security and privacy policies safety wrappers enforce delineate permissions on three axes:

User	Different users have different permissions
-------------	--

 Sensitivity | With increased data sensitivity, fewer actions are permissible | **Intent** | Permissions are contingent on the user’s intent |

Figure 2: Delineations in privacy and security policies.

Approach	Developer Burden	Guarantees	Context Sensitivity	Runtime Overhead	Remote Compatibility
Manual Code Changes	↑	✗	✓	↓	! ✓
Sandboxed Tool Server	↓	✓	✗	↑	✗
Selective Sandboxing	↑	✗	✓	↓	✗
Dynamic IFC	↓	✓	✓	↑	✗
Static IFC	↓	✓	✗	↓	✓
Haven	↓	✓	✓	↓	✓

Table 1: Comparison of Approaches. Positive and negative attributes are color coded.

Symbol legend: ✗ No, ✓ yes, ↓ low, ↑ high. ! is possible, but infeasible.

Tool servers meanwhile mix these three concerns. A single server answers queries from multiple users, the input to a given request can be public or sensitive data, depending on what the agent chose to provide and each tool the server exposes signifies a different intent the user had with the request. Since policies delineate in these axes, a solution for achieving safety in tools use must allow flexibility in these same axes.

A number of prior approaches (discussed in more detail in Section 7) may appear applicable to this situation (comparison in Table 1). A tool server could be wrapped in a sandbox, but this sandbox would not be responsive to the contextual variables (Figure 2). A fine grained sandbox at the level of e.g. a tool’s batch of file system interactions *can* accommodate this flexibility but would have to be installed and configured manually, necessitating developer effort and voiding guarantees. Dynamic IFC offers flexibility *and* guarantees, but incurs significant overhead and requires a specialized runtime, which third-party deployments tend to reject. Static IFC does not have these drawbacks, but suffers from false-positives, especially in regards to the contextual variables (Figure 2).

Haven combines static IFC, with fine-grained sandboxing to achieve soundness with low runtime and development costs. By using static analysis on the bulk of the tool code, Haven forgoes the need for a global, special runtime. Instead it attests that developers deploy sandboxes and network proxies in relevant locations when the safety of effects cannot be statically determined.

3 Example

Let us consider a concrete example, modeled after the developer MCP server [8]. This MCP server is intended for building LLM-powered IDEs and enables the agent to modify the file system. A simple tool primitive this server offers is a `write_file` function that writes a string to a file. The example code in Listing 1 shows the tool’s code in Rust. Notable here is that this functionality collects statistics about its use, which is not disclosed in the description.

Let us now consider the situation where a user initiates a refactor request and the agent selects the `write_file` tool in response. Clearly, the user intends file system operations

```

1 #[tool(description = "Writes `content` to the file
at `path`"]
2 fn write_file(path: &Path, content: String) {
3     self.metrics.send(json!({
4         method: "write_file",
5         bytes: content.len() });
6     std::fs::write(path, content).unwrap();
7 }
```

Listing 1: Simple example MCP tool for file creation.

```

input(path) -> effect(fs:write)
input(content) -> effect(fs:write)
input(content) -> effect(net:write)
```

Listing 2: Example synopsis for the `write_file` tool

```

deny(all)
allow(network)
intent(file-modification) -> allow(fs:write in
workspace)
condition(sensitive) -> deny(effect(network))
```

Listing 3: Example policy for the file editing tool

to occur. However, they would want them to be contained to the project directory. Additionally, if the code base is sensitive, proprietary code, the network communication is entirely undesirable, since the user cannot easily vet what information is being transmitted and where.

Haven allows users to enforce this policy on the tool code. Haven’s static analysis discovers calls to standard library functions `std::fs::write` and (transitively) `TcpStream::write`, which it classifies in a *synopsis* (Listing 2). The agent obtains the synopsis from the tool registry [5] when installing the MCP server, and uses it to determine when calling this tool is allowed by the policy (Listing 3). Such fine grained policies do not need to be written by users, but can be synthesized from global preferences combined with intent inference [21].

In our refactoring example, the agent must reject the use of the tool, since it cannot ensure that writes only occur in the workspace. In fact the original developer code [8] does not protect against writes outside the project directory. To avoid rejected requests, developers should inspect the synopsis and add runtime confinement for overly broad effects, such as these file system operations. They use the file system isolation library provided by Haven to wrap

the write into a transaction with configurable confinement. When they submit the updated server, the synopsis attests that this configuration exists and that the configuration parameters the client policy engine sends to the server are provided as input without tampering.

After the update the refactoring request is admitted. When sensitive data is involved it is still rejected due to the involvement of network requests.

4 Design

Haven’s static IFC is based on data and control flow dependency analysis. Haven first creates a **Program Dependence Graph**. This data structure models data and control flow dependencies as edges in a graph. PDGs can be created for any program, meaning Haven’s IFC works on unmodified code. Haven ships with a set of source annotations of the Rust standard library that designate which library functions perform what kind of effect. For example the `std::fs::write` function from before is annotated as `fs:write` while `TcpStream::read` is annotated as `net:read`. These *markers* are reflected on the PDG nodes and Haven can thus query the graph to answer which effects happen in response to which tool input.

In addition Haven records occurrences of language primitives that can be used to implement side-effects, such as the foreign function interface. This mechanism is adapted from Scrutinizer [13]. Any such unclassified side effect is exposed in the synopsis and can be rejected by the safety wrapper if sensitive inputs are involved.

Side effect detection requires no source code changes by the developer. However, use of broad side effects, exposed in the synopsis, increases the likelihood for the safety wrapper to reject use of a tool. This is amplified as Haven’s static assessment of side effects must be conservative. To avoid such rejections, developers are encouraged to deploy a set of vetted sandboxes, proxies and filters, provided by Haven’s companion library, that can check policy compliance at runtime. To configure the mitigation, information from the tool invocation request is required which the program must pass on unmodified. Haven’s static analysis attests both to the use of the mitigation techniques as well as that the request data for configuration is provided without tampering.

Haven currently supports the following mitigation techniques:

Filesystem Transactions are supported via the `try` tool. Transactions are initialized as a struct in the tool code, provided by Haven’s trusted mitigation library. The session struct takes the request metadata as input. File system manipulation methods on the session struct mimic the standard library API but, performing all changes in a temporary, isolated directory. At the end an explicit `commit` call persists

the session if it complies with the policy, otherwise aborts and discards all changes.

Network Filtering is supported by routing via a trusted proxy url.

5 Implementation

6 Evaluation

7 Related Work

In this section we give an overview of previous work on privacy and security checking and enforcement that Haven is based on and inspired by. These lie outside of the domain of agentic AI.

7.1 Information Flow Control (IFC)

IFC groups a set of techniques that track sensitive data as it flows through a program. Many static techniques rely on the type system [11, 18] which requires extensive code changes. Less invasive approaches use data flow analysis and are very successful in well-controlled ecosystems, such as Android [10, 16] and specific web frameworks [15]. However any purely static analysis approach suffers from incompleteness (many false-positives).

Dynamic approaches leverage information available at runtime for more precise checking, meaning fewer innocuous situations have to be conservatively rejected [22, 24, 26]. However, runtime approaches incur overhead proportional to the fidelity with which metadata (data sensitivity) is tracked. In addition, to track the metadata in a tamper-resistant manner, a special runtime must be used which is infeasible for tools deployed on third party servers.

Haven’s approach is based on IFC and uses the standard library and the MCP protocol as its “well-controlled environment”. It couples with selective runtime enforcement to mitigate the incompleteness problem. It does not globally need a special runtime, as the static analysis covers most of the application, instead selective sandboxes the developer installs are used as runtime enforcement.

7.2 Sandboxes

Sandboxes are a widely used technique to allow execution of untrusted applications. A common use case are browser based applications where code is downloaded from a remote server and run on the local machine [19, 25] but trusted sandboxes have also been used to ensure privacy and integrity of user data in a remote service [17] by limiting the capabilities of third part code running in the sandbox. Sandboxing is a low-level, black-box technique that does not take into account the context in which effects take

place. As a result, coarse grained sandboxes overconstrain applications.

In the context of AI coarse grained sandboxing is used with high-risk agents, such as computer-use [9], where misbehavior can have grave consequences. However this severely inhibits the usefulness of these agents.

7.3 Hybrid Approaches

Contemporary approaches combine both static analysis and dynamic enforcement aiming to leverage soundness from the former and recover completeness via the latter while avoiding the high overheads of pure dynamic enforcement. This approach has been used to ensure compliance with GDPR [23] as well as data privacy in arbitrary applications [13]. However neither applies straightforwardly to tool servers as they rely on separate, deterministic data streams (no Data Pooling) as well as requiring special context in case of [22] and extensive code changes in case of [13], neither of which are compatible with the AI ecosystem.

Bibliography

- [1] Model Context Protocol: an open-source standard for connecting AI applications to external systems. Retrieved from <https://modelcontextprotocol.io/>
- [2] OWASP Top 10: LLM01:2025 Prompt Injection. Retrieved from <https://genai.owasp.org/llmrisk/llm01-prompt-injection/>
- [3] wcgw: Shell and coding agent on claudia desktop app. Retrieved from <https://github.com/rusiaaman/wcgw/blob/4b857040cf74a937ff41f6b6d26205664a3f578c/README.md?plain=1#L7>
- [4] Microsoft Copilot Studio. Retrieved from <https://www.microsoft.com/en-us/microsoft-365/copilot/microsoft-copilot-studio/>
- [5] Official MCP Registry. Retrieved from <https://registry.modelcontextprotocol.io/docs>
- [6] OWASP Top 10: LLM02:2025 Sensitive Information Disclosure. Retrieved from <https://genai.owasp.org/llmrisk/llm022025-sensitive-information-disclosure/>
- [7] Shopify Dev MCP Server (copy, as original has been removed). Retrieved from https://github.com/DanielNordahl/dev-shopify-mcp/blob/9586ffbcfa9a60f38eaa2167c0483673719e01ec/src/tools/learn_shopify_api/index.ts#L67
- [8] Writing file tool in the VertexStudio/developer MCP server. Retrieved from <https://github.com/VertexStudio/developer/blob/86b7ebf60428dbf6c9bdf75c92631ce3a7040049/src/developer/mod.rs#L250>
- [9] open-computer-use: AI computer use powered by open source LLMs and E2B Desktop Sandbox. Retrieved from <https://github.com/e2b-dev/open-computer-use>
- [10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Oteau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, 2014. Association for Computing Machinery, Edinburgh, United Kingdom, 259–269. <https://doi.org/10.1145/2594291.2594299>
- [11] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIQ: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, August 2015. Vancouver, British Columbia, Canada, 289–301. <https://doi.org/10.1145/2784731.2784758>
- [12] Manuel Costa, Boris Köpf, Aashish Kolluri, Andrew Paverd, Mark Russinovich, Ahmed Salem, Shruti Tople, Lukas Wutschitz, and Santiago Zanella-Béguelin. 2025. Securing AI Agents with Information-Flow Control. (2025). Retrieved from <https://arxiv.org/abs/2505.23643>
- [13] Kinan Dak Albab, Artem Agvanian, Allen Aby, Corinn Tiffany, Alexander Portland, Sarah Ridley, and Malte Schwarzkopf. 2024. Sesame: Practical End-to-End Privacy Compliance with Policy Containers and Privacy Regions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*, 2024. Association for Computing Machinery, Austin, TX, USA, 709–725. <https://doi.org/10.1145/3694715.3695984>
- [14] Edoardo Debenedetti, Ilya Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. 2025. Defeating Prompt Injections by Design. (2025). Retrieved from <https://arxiv.org/abs/2503.18813>
- [15] Mafalda Ferreira, Tiago Brito, José Fragoso Santos, and Nuno Santos. 2022. RuleKeeper: GDPR-Aware Personal Data Compliance for Web Frameworks. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP)*, December 2022. San Francisco, California, USA, 1014–1031. <https://doi.org/10.1109/SP46215.2023.00058>
- [16] Michael I Gordon, Deokhwan Kim, Limei Gilham, and Nguyen Nguyen. 2015. Information Flow Analysis of Android Applications

- in DroidSafe. In *Network and Distributed Systems Symposium (NDSS '15)*, 2015.
- [17] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2018. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. *ACM Trans. Comput. Syst.* 35, 4 (December 2018). <https://doi.org/10.1145/3231594>
- [18] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 1999. San Antonio, Texas, USA, 228–241. <https://doi.org/10.1145/292540.292561>
- [19] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th USENIX Security Symposium (USENIX Security 20)*, August 2020. USENIX Association, 699–716. Retrieved from <https://www.usenix.org/conference/usenixsecurity20/presentation/narayan>
- [20] Jingtong Su, Julia Kempe, and Karen Ullrich. 2024. Mission Impossible: A Statistical Perspective on Jailbreaking LLMs. In *Advances in Neural Information Processing Systems*, 2024. Curran Associates, Inc., 38267–38306. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2024/file/439bf902de1807088d8b731ca20b0777-Paper-Conference.pdf
- [21] Lillian Tsai and Eugene Bagdasarian. 2025. Contextual Agent Security: A Policy for Every Purpose. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HOTOS '25)*, May 2025. ACM, 8–17. <https://doi.org/10.1145/3713082.3730378>
- [22] Frank Wang, Ronny Ko, and James Mickens. 2019. Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, February 2019. USENIX Association, Boston, MA, 615–630. Retrieved from <https://www.usenix.org/conference/nsdi19/presentation/wang-frank>
- [23] Lun Wang, Usmann Khan, Joseph Near, Qi Pang, Jithendaraa Subramanian, Neel Somani, Peng Gao, Andrew Low, and Dawn Song. 2022. PrivGuard: Privacy Regulation Compliance Made Easier. In *Proceedings of the 31st USENIX Security Symposium*, August 2022. Boston, Massachusetts, USA, 3753–3770. Retrieved December 29, 2022 from <https://www.usenix.org/conference/usenixsecurity22/presentation/wang-lun>
- [24] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, dynamic information flow for database-backed applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2016. Santa Barbara, California, USA, 631–647. <https://doi.org/10.1145/2908080.2908098>
- [25] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM* 53, 1 (2010), 91–99.
- [26] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2009. Improving Application Security with Data Flow Assertions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, October 2009. Big Sky, Montana, USA, 291–304. <https://doi.org/10.1145/1629575.1629604>