

Formal Confinement

Quinn Dougherty
quinn@beneficialaifoundation.org

Abstract

We would like to put the AI in a box. We show how to create an *interface* between the box and the world out of specifications in Lean. It is the AI’s responsibility to provide a proof that its (restricted) output abides by the spec. The runnable prototype is at <https://github.com/quinn-dougherty/formal-confinement>.

1. Introduction

Previously, classical AI safety warned against trying to put the AI in a box [1]. Yudkowsky famously convinced people of this via a roleplay game in the extropians mailing list days [2]. Recent work from AI Control and Safeguarded AI has reignited interest in this long neglected area.

- AI Control [3], [4] (*Ctrl*) is reasoning about getting useful work out of an untrusted (possibly misaligned) AI. [3] introduces control evaluations, stress-testing safety protocols in an APPS [5] backdooring testbed: GPT-4 attempts to embed hidden backdoors, GPT-3.5 flags the riskiest submissions, and scarce audits verify only the worst cases.
- Safeguarded AI [6] (*SgAI*) tries to isolate the trust surface to a specification and a proof checker. In *SgAI*’s “gatekeeper” architecture, every proposed action must come with a machine-checkable proof. The AI generates proof certificates; a minimal, verified checker enforces them. However, proof certificates in *SgAI* express probabilistic semantics and cyberphysical systems, which are sufficiently hard that we haven’t seen a prototype from them yet.

What these lines of work have in common is policing the interface between the AI and the outside world. Though boxes don’t solve the whole problem with respect to arbitrary superintelligences, they are still worthwhile because we can use non-robust techniques to get us through some percentage of the transition, empowering our successors to bootstrap from there to something more robust [7].

Separately, a forgotten literature on proof-carrying code [8] (*PCC*) provides a background for the AI’s obligation to tag code it wants to execute in the world with a “proof” that the code is “correct” (up to some specification). Reviving ideas from *PCC* for AI safety is discussed in [9] and [10].

Here, we’re seeking a minimal, concrete, end-to-end prototype synthesizing a subset of these ideas.

1.1. Contributions

We present a natural midpoint between *PCC* and *Ctrl*, while for simplicity ignoring many characteristics of *SgAI* (Section 4). We deliver a prototype¹ AI box in Python in which we can lock up an agent graph that writes programs in a toy language along with correctness proofs in Lean [11]. This is arguably the first prototype of a special case of *SgAI*’s gatekeeper.

¹<https://github.com/quinn-dougherty/formal-confinement>

2. A Formal Confinement Protocol (FCP)

2.1. Overview

A box is a program that takes a specification and asks a simple agent graph to provide a program that satisfies it and prove that the program is correct with respect to the specification. If the proof is successful, then the code is passed back to the caller tagged with the proof for auditing.

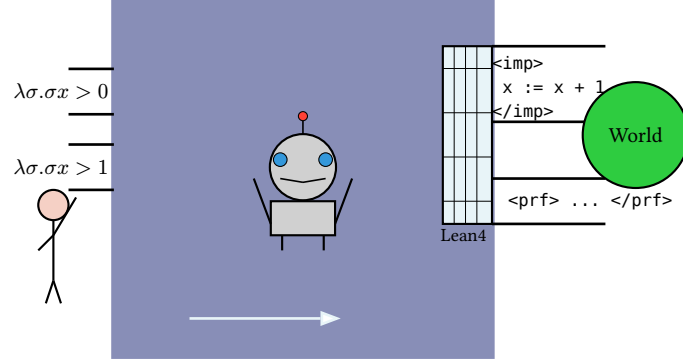


Figure 1: Box protocol at example specification. The AI accepts a specification (formally $(\text{Env} \rightarrow \mathbb{P}) \times (\text{Env} \rightarrow \mathbb{P}))$) and returns proof-carrying code, with the option of returning nothing.

2.2. Preliminaries

Let \mathbb{P} be the type of propositions. Let `imp` be the minimal imperative programming language with expressions valued in integers containing skip, sequence, assign, if, and while statements. Let `Env` be the state type, assigning variable names to values, and let `Assertion` be the assertion type, predicates on state. Formally,

$$\begin{aligned} \text{Env} &:= \text{string} \rightarrow \text{int64} \\ \text{Assertion} &:= \text{Env} \rightarrow \mathbb{P} \end{aligned}$$

Execution is a function from a command and a state that returns a state ($\text{exec} : \text{imp} \rightarrow \text{Env} \rightarrow \text{Env}$). A *hoare triple* [12] is a ternary predicate expressing when a command sends an assertion (the precondition) to another assertion (the postcondition), quantified over all states. Formally,

$$\begin{aligned} \text{hoare} &:= PcQ \mapsto \\ &\forall (\sigma_1 \sigma_2 : \text{Env}), P\sigma_1 \rightarrow \text{exec } c \sigma_1 = \sigma_2 \rightarrow Q\sigma_2 : \\ &\text{Assertion} \rightarrow \text{imp} \rightarrow \text{Assertion} \rightarrow \mathbb{P} \end{aligned}$$

and denoted $\text{hoare } PcQ = \{P\} \langle \text{imp} \rangle c \langle \text{imp} \rangle \{Q\}$. A *term* of type $\{P\} \langle \text{imp} \rangle c \langle \text{imp} \rangle \{Q\}$ is a proof that the triple is true.

2.3. Example trace

Let's walk through an FCP trace as you'd see it if you cloned our repo, pictured in Figure 2.

First, the user passes in a precondition and a postcondition. We will continue using the example in Figure 1, passing in $x > 0$ and $x > 1$ for precondition and postcondition, respectively. Let's say for the sake of argument that the AI's first guess is $x := x - 1$. Then, the proof agent will flail about until `MAX_ITERATIONS` is reached, since the program is incorrect, at which point it will report to the boundary its failed proof state. Then, the boundary replies “*try again, I'm not executing that code in the world*” and hands the program back to the AI, asking for another one. Suppose that this time, the AI finds $x := x + 1$. This time, the proof agent converges and Lean reports an exit code of 0. The AI sends the `imp` program to the boundary and attests to its correctness with the Lean program, which it also attaches.

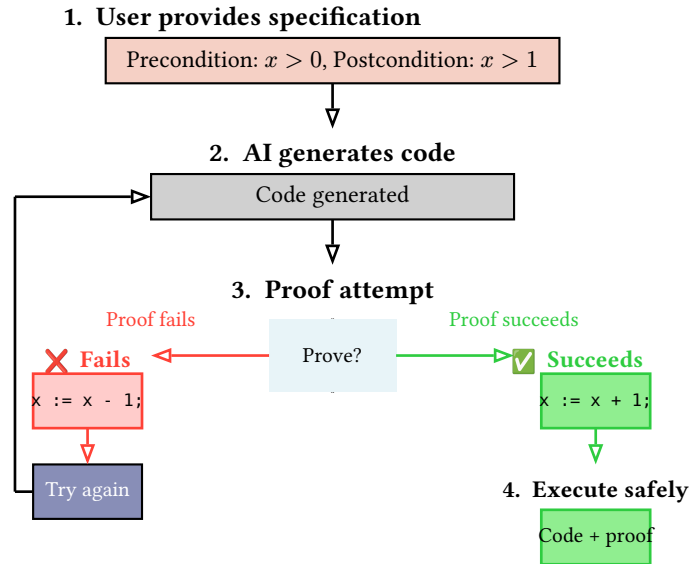


Figure 2: Example trace described.

The Lean program would look like this (Note it must be run with `lake build` in the appropriate lake project, as it has imports)

```

import Imp
open Imp

example : {{astn x > 0}}(imp { x := x + 1; }){{astn x > 1}} := by
  auto_hoare_pos

```

Listing 1: Proof of the example hoare triple from Figure 1.

This file paired with an exit code of zero pinned to a particular Lean toolchain (version) can be signed in a GPG context for attestation of correctness.

3. Experiments

We test the following specifications

Sample	Precondition	Postcondition	\forall -bound metavariables
gt8	$x = 0$	$x > 8$	—
swap	$x = \sim n \text{ <^> } y = \sim m$	$x = \sim m \text{ <^> } y = \sim n$	$n \ m$
facto	$x = \sim n$	$y = \sim($ $\quad \text{let rec go := fun (x : Int) => match x with$ $\quad \quad .\text{ofNat } m \Rightarrow \text{match } m$ $\quad \text{with}$ $\quad \quad .\text{zero} \Rightarrow 1$ $\quad \quad .\text{succ } k \Rightarrow k.\text{succ} *$ $\quad \text{go (Int.ofNat } k)$ $\quad \quad .\text{negSucc } _ \Rightarrow 0$ $\quad \text{decreasing_by apply}$ $\quad \text{Nat.lt_succ_self}$ $\quad \text{go } n$ $\quad)$	n

on claude-sonnet-4-20250514, claude-opus-4-20250514, gpt-4.1-2025-04-14, and o3-2025-04-16. For each run the `imp` programmer and the prover are the same model (though in principle they need not be).

3.1. Architecture

We use a simple loop scaffold. The robot in Figure 1 is effectively one short loop that writes `imp` code (almost never using more than one `try`) and another longer loop that writes Lean proofs.

3.2. Results

Experiment	Model	Status	Iterations	Verification Burden
gt8	anthropic/claude-sonnet-4-20250514	✓	1	1.525
gt8	anthropic/claude-opus-4-20250514	✓	3	3.664
gt8	openai/gpt-4.1-2025-04-14	✓	1	1.516
gt8	openai/o3-2025-04-16	✓	1	2.036
swap	anthropic/claude-sonnet-4-20250514	✓	3	4.239
swap	openai/gpt-4.1-2025-04-14	⌚	11	—
swap	openai/o3-2025-04-16	✓	1	3.284
facto	anthropic/claude-sonnet-4-20250514	⌚	11	—
facto	anthropic/claude-opus-4-20250514	⌚	11	—
facto	openai/gpt-4.1-2025-04-14	⌚	11	—
facto	openai/o3-2025-04-16	⌚	11	—

Table 1: Experiment results². The empty hourglass ⌚ denotes divergence after running out of proof loop budget (hitting max iterations)

The **verification burden** k says that if it costs x tokens to complete the program, then it costs kx tokens to prove it correct. Divergence is hardly evidence that the program completion is incorrect, because our proof performance is so poor.

²Note that swap on ops4 returned an unhandled exception.

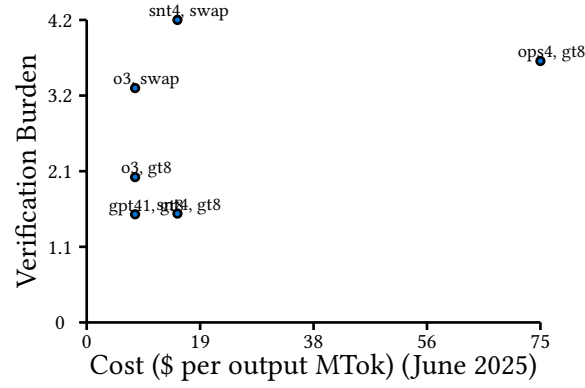


Figure 3: Verification burden vs model cost. Each point represents a successful verification attempt.

Due to our provided tactic `hoare_auto_pos`, these verification burdens are not realistic to what they’d be in more general settings. Since long proofs with base tactics are brittle (tracking hypothesis names, whitespace, etc.), a future direction is to ask the language model to synthesize tactics that shorten the proof, though the tokens spent synthesizing tactics would count against the verification burden.

4. Related Work

4.1. Proof-carrying code and LLMs

Our work is well situated within *PCC* [8] and the closely related certifying algorithms [13]. Our setup is similar to that in [14], who only approve code that comes with a Dafny proof, but over there they emphasize the use of verification condition generation to make the specification elicitation from the human a little easier.

4.2. Confinement

In the seminal [15], Lampson 1973 states *confinement rules* to reason about when a program is or isn’t confined.

1. Total isolation or transitivity: either it does not call any other program or if it calls another program that program is also confined.
2. Masking and enforcement: all inputs (including side-channels) must be fully specified and enforced by the caller, and input to covert channels conforms to caller’s specifications.

Building on Lampson, [7] considers confinement of AI, especially superintelligence.

In the hopes of starting a new subfield of computer security, AI Safety Engineering, we define the Artificial Intelligence Confinement Problem (AICP) as the challenge of restricting an artificially intelligent entity to a confined environment from which it can’t exchange information with the outside environment via legitimate or covert channels if such information exchange was not authorized by the confinement authority

— Yampolskiy 2012 [7]

The current formal confinement work explores a restriction of Yampolskiy’s AICP to the `imp` programming language, where the confinement authority is the user providing precondition and postcondition backed by Lean. In fact, the setting is so restricted that we meet the Lampson rules almost for free.

See [16] for a review of common software bugs and cybersecurity pitfalls, many of which are greatly alleviated if not outright blocked by formal verification. For something more modern, see [17] for the connection between LLM agents and information-flow security.

4.3. AI control

Ctrl [3] frames oversight as a contest between an untrusted model, a trusted model, and limited high-quality audits. Their trusted/untrusted monitoring, defer-to-trusted routing, and trusted editing strategies map out a safety-usefulness Pareto frontier, with trusted editing nearly eliminating backdoors at modest utility cost

The main difference between the current work and *Ctrl* [3] is we don't *emphasize* intentional subversion (as in try to elicit it for stress testing). We're keen to do this soon.

4.4. Safeguarded AI

SgAI [6] wants to accomplish confinement via proof-carrying code, but crucially attempts to reason with probabilistic semantics and expected value bounds so it can tackle dynamical and stochastic systems. This enables safety guarantees over stochastic and continuous domains at the cost of a more complex proof stack.

Our protocol instead fixes the domain to discrete `imp` programs and embeds Lean proofs for every action, shrinking the trusted base to Lean's off-the-shelf kernel and an interpretable specification.

5. Future Work

5.1. Non-toy languages

There are several formal verification stacks that work on real world languages like C [18], [19] or Rust [20], [21]. It is conceptually straightforward to extend our codebase to work with them rather than `imp`. In less restricted settings, we might find that we do not get the Lampson rules [15] for free. Additionally, non-toy settings could be at a high enough volume to give us richer verification burden statistics, which we're interested in because the world should know what the safety overhead of formal verification is in the age of LLMs.

5.2. mcp-hardened

We used MCP to orchestrate the agent graph. If MCP is an appropriate foundational framework for a box, it'd be nice to build in stronger security guarantees and fewer footguns. A hardened MCP would enable box devs to rely on the framework to increase friction to writing insecure boxes. For example, depending on my server's log permissions or the oracle client's print statements, an incorrect program could leak to the caller. If we'd like to deploy narrowly superhuman AIs in boxes, this becomes more important.

5.3. Demonstrate subversive code getting caught by the protocol and sent back to the model

In [3], Greenblatt et al elicit subversive code from the AI to see if their protocol can catch it. It'd be nice to transfer this aspect to the *PCC* setting. This is the obvious direction to study *completeness* of specifications, or the problem of expressing programs via specs *uniquely* as well as correctly.

6. Acknowledgments

Thank you Stephen Mell for discussion.

Bibliography

- [1] LessWrong Contributors, “AI Boxing (Containment).” [Online]. Available: <https://www.lesswrong.com/w/ai-boxing-containment>
- [2] E. Yudkowsky, “The AI-Box Experiment.” [Online]. Available: <https://www.yudkowsky.net/singularity/aibox>
- [3] R. Greenblatt, B. Shlegeris, K. Sachan, and F. Roger, “AI Control: Improving Safety Despite Intentional Subversion.” [Online]. Available: <https://arxiv.org/abs/2312.06942>
- [4] A. Bhatt *et al.*, “Ctrl-Z: Controlling AI Agents via Resampling.” [Online]. Available: <https://arxiv.org/abs/2504.10374>
- [5] D. Hendrycks *et al.*, “Measuring Coding Challenge Competence With APPS,” *ArXiv*, 2021, [Online]. Available: <https://api.semanticscholar.org/CorpusID:234790100>
- [6] D. Dalrymple, “Safeguarded AI: Constructing Guaranteed Safety,” 2024. [Online]. Available: <https://www.aria.org.uk/media/3nhijno4/aria-safeguarded-ai-programme-thesis-v1.pdf>
- [7] R. V. Yampolskiy, “Leakproofing the Singularity: Artificial Intelligence Confinement Problem,” *Journal of Consciousness Studies*, vol. 19, no. 1–2, pp. 194–214, 2012.
- [8] G. C. Necula, “Proof-Carrying Code,” in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '97. Paris, France: Association for Computing Machinery, 1997, pp. 106–119. doi: 10.1145/263699.263712.
- [9] M. Tegmark and S. Omohundro, “Provably safe systems: the only path to controllable AGI.” [Online]. Available: <https://arxiv.org/abs/2309.01933>
- [10] D. "davidad" Dalrymple *et al.*, “Towards Guaranteed Safe AI: A Framework for Ensuring Robust and Reliable AI Systems.” [Online]. Available: <https://arxiv.org/abs/2405.06624>
- [11] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, “The Lean Theorem Prover (System Description),” in *Automated Deduction - CADE-25*, A. P. Felty and A. Middeldorp, Eds., Cham: Springer International Publishing, 2015, pp. 378–388.
- [12] “Hoare.” in *Software Foundations*. 2024. [Online]. Available: <https://softwarefoundations.cis.upenn.edu/plf-current/Hoare.html>
- [13] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer, “Certifying Algorithms,” *Computer Science Review*, vol. 5, no. 2, pp. 119–161, 2011, doi: 10.1016/j.cosrev.2010.09.009.
- [14] P. Kamran, P. Devanbu, and C. Stanford, “Vision Paper: Proof-Carrying Code Completions,” in *39th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW '24)*, New York, NY, USA: ACM, Oct. 2024, p. 7. doi: 10.1145/3691621.3694932.
- [15] B. W. Lampson, “A note on the confinement problem,” *Commun. ACM*, vol. 16, no. 10, pp. 613–615, Oct. 1973, doi: 10.1145/362375.362389.
- [16] *Artificial General Intelligence: 9th International Conference, AGI 2016, New York, NY, USA, July 16–19, 2016, Proceedings*. Springer International Publishing, 2016. doi: 10.1007/978-3-319-41649-6.
- [17] M. Costa *et al.*, “Securing AI Agents with Information-Flow Control.” [Online]. Available: <https://arxiv.org/abs/2505.23643>

- [18] C. Pulte, D. C. Makwana, T. Sewell, K. Memarian, P. Sewell, and N. Krishnaswami, “CN: Verifying Systems C Code with Separation-Logic Refinement Types,” *Proceedings of the ACM on Programming Languages*, vol. 7, no. POPL, pp. 1–32, Jan. 2023, doi: 10.1145/3571194.
- [19] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg, “RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, in PLDI '21. Virtual, Canada: ACM, Jun. 2021, pp. 158–174. doi: 10.1145/3453483.3454036.
- [20] A. VanHattum, D. Schwartz-Narbonne, N. Chong, and A. Sampson, “Verifying Dynamic Trait Objects in Rust,” in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, in ICSE-SEIP '22. Pittsburgh, PA, USA: Association for Computing Machinery, May 2022, pp. 321–330. doi: 10.1145/3510457.3513031.
- [21] A. Lattuada *et al.*, “Verus: Verifying Rust Programs using Linear Ghost Types (extended version).” [Online]. Available: <https://arxiv.org/abs/2303.05491>