Mephistopheles: A Recursive, Simulation-Grounded Language-Model Agent for Permissionless Smart-Contract Exploit Discovery

The Mephistopheles Project

Independent research · correspondence to the maintainers of the reference implementation

18 August 2026

Abstract

We describe **Mephistopheles**, an agentic system that pairs a locally-hosted large language model with a live Ethereum node to discover vulnerabilities that an *arbitrary unprivileged caller* can exploit against a deployed, source-verified smart contract. The system operates in two stages. A *static audit* reasons over verified Solidity source and the contract's callable ABI to emit a structured set of candidate findings. A subsequent *recursive hunt* then closes the loop against reality: the model proposes hypotheses and read-only on-chain *probes*, a simulation oracle executes each probe as an `eth call` against current mainnet state, and the returned observations are fed back to sharpen, confirm, or refute each hypothesis across bounded rounds. We formalize this interaction as a partially-observed, oracle-grounded search in which the model is a policy and the chain is a deterministic environment, and we detail the mechanisms that make it robust: schema-constrained decoding with repair, caller spoofing for permissionless-reachability proofs, and a simulate-then-repair fixpoint for exploit-argument synthesis. Crucially, the reference build is **simulation-only**: the transaction-signing path is disabled by construction, so no confirmed finding is ever broadcast. We discuss the threat model, the soundness and the sharp limits of single-call simulation, an evaluation protocol, and the ethics of publishing dual-use automation.

Keywords: smart-contract security, large-language-model agents, simulation-grounded search, reinforcement-style feedback, EVM, automated vulnerability discovery, responsible disclosure.

1 Introduction

A deployed smart contract is a rare object in computer security: its complete logic is public, immutable, and directly addressable by anyone, and the assets it guards are liquid and irreversible. This collapses the usual distance between "a bug exists" and "the bug is money." The attack surface that matters most is not the privileged one—an owner who can pause or drain a contract is a trust assumption, not a vulnerability—but the *permissionless* one: what an ordinary external address, holding no special role, can make the contract do against the intent of its authors.

Classical tooling attacks this surface from the syntax and semantics of the code. Static analyzers pattern-match dangerous constructs; symbolic executors enumerate paths and solve for reverts; fuzzers mutate transaction sequences against invariants. Each is powerful and each is blind in a characteristic way —to intent that is legal-but-wrong, to economic composition across contracts, and to the actual on-chain state that decides whether a theoretically-reachable path is reachable *today*.

Mephistopheles takes a complementary position. It treats vulnerability discovery as a *reasoning* problem grounded by an *oracle*. A language model supplies the hypotheses—the "what if this accounting invariant can be violated by fee-on-transfer tokens" intuition that reads like a human auditor's—and a live

Ethereum node supplies ground truth, answering, for any concrete call the model can imagine, exactly what the deployed bytecode does against current state. The contribution is not either half alone but the disciplined loop between them, and the engineering that keeps that loop honest.

This paper documents the system as built. Section 2 fixes the threat model. Section 3 gives the pipeline. Sections 4–10 detail each subsystem, with Section 6 formalizing the recursive loop that gives the system its namesake persistence. Section 11 describes the safety boundary that makes the reference build a discovery instrument rather than a weapon. Sections 12–15 cover evaluation, limits, related work, and ethics.

2 Threat model and scope

We fix the adversary precisely, because the entire system is tuned to it. The adversary A is a single externally-owned account with *no* privileged role: not owner, admin, minter, pauser, upgrader, or the holder of any access-controlled capability. A may call any `public`/`external` function, may supply any arguments, may hold or borrow capital (including via flash loans), and may order or sandwich its own transactions. A may *not* forge a privileged `msg.sender`, compromise a key, or rely on governance.

A *finding* is a sequence of permissionless calls whose effect violates a security-relevant property of the contract—value conservation, access to others' funds, liveness for other users—that a correct implementation would preserve. Everything gated behind a role check is, by definition, **out of scope**: centralization risk, owner rug-pull power, and privileged mis-configuration are trust decisions the system deliberately refuses to report, because flagging them buries the exploitable in the merely-worrying.

> ***Scope invariant.*** Every reasoning stage is instructed, and every simulation is constructed, to model A as an arbitrary non-owner address. A candidate that only "works" as the owner is treated as a refutation, not a finding. The scope is enforced by prompt and by construction of the `from` field in each simulated call—a soft boundary whose limits we return to in Section 13.

3 System overview

Mephistopheles is a single-page client that orchestrates two external services: a local model server exposing an OpenAI-style chat endpoint, and Ethereum JSON-RPC (reached through a redundant set of public gateways). Given only a contract address and a network, it runs the pipeline of Figure 1 end to end.

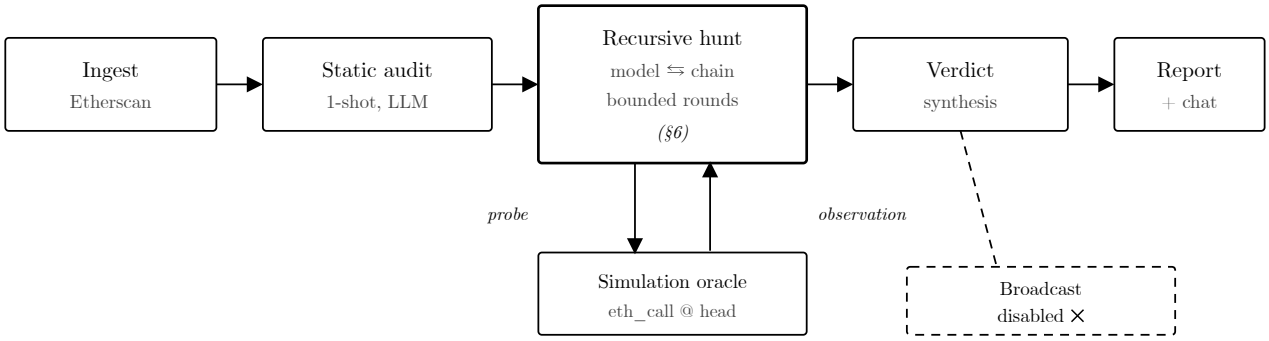

Figure 1: The pipeline. Solid arrows carry the forward flow; the central cycle is the model–chain loop of Section 6, grounded by a read-only simulation oracle. The dashed box marks the deliberately severed signing path (Section 11): confirmed exploits are demonstrated in simulation and never broadcast.

Two design decisions in this overview are load-bearing. First, ingestion and the interactive tester are enabled the instant source and ABI arrive, *independent* of the model—the chain half of the system never waits on the language half. Second, the static audit and the recursive hunt are separate stages with separate prompts and separate output schemas; the audit is a fast, breadth-first pass, while the hunt is a slow, depth-first pass that spends its budget confirming a few deep hypotheses against live state.

4 Contract ingestion and normalization

The pipeline begins by resolving an address to auditable material. The client queries the block-explorer `getsourcecode` endpoint, which returns the contract name, the verified Solidity source, and the ABI when—and only when—the contract is source-verified. Verification is the gate: unverified bytecode carries no reliable intent to reason about, so the system distinguishes three cases by cross-checking `eth getCode`:

- **Verified**—source and ABI present; full pipeline proceeds.
- **Deployed but unverified**—non-empty bytecode, no source; the system reports that there is nothing to source-audit rather than hallucinating over a disassembly.
- **Empty / EOA**—no code at the address; the run halts with a diagnostic.

From the ABI the system derives a canonical *function catalogue*: for each callable entry, a signature string of the form `name(type arg, …) mutability`. This catalogue is the vocabulary shared by every downstream stage. Both prompts are instructed to reference *only* names drawn from it, and every simulated call is encoded against it—a single source of truth that prevents the model from inventing functions that do not exist on the deployed contract. Source is truncated to a fixed budget (tens of thousands of characters) before it enters a prompt; Section 13 discusses what this costs.

5 Stage I: single-shot structured audit

The first model interaction is a breadth-first sweep. A senior-auditor system prompt fixes the mandate (permissionless exploits only), enumerates the classes of interest—reentrancy, edge-case arithmetic, allowance and approval races, oracle and AMM manipulation, front-running and MEV, flash-loan-enabled

paths, missing validation, and broken accounting invariants—and constrains the output to a single JSON object. Each finding carries a severity, a plain-language explanation, a concrete step-by-step *exploitation*, the real-world *outcome*, a mitigation, and—critically—a *test*: the single call from the catalogue whose result best demonstrates or probes the finding.

The *test* field is what turns a paragraph of prose into something falsifiable. Rather than trusting the model's assertion that (say) an allowance can be double-spent, the system attaches to each finding a concrete, catalogue-valid call that a human—or Stage II—can execute against the chain to see the claim confirmed or contradicted. The prompt further demands *realistic* arguments: amounts scaled to token decimals, real addresses with genuine on-chain state, and every attacker-controlled recipient routed to a single controlled test address so that any redirected effect is observable in one place.

Stage I is deliberately shallow and fast. It produces the candidate set; it does not verify it. Verification is the job of Stage II.

6 The model–chain recursive loop

The second stage is the heart of the system and the reason it is described as *recursive* and *reinforcement-style*. Rather than answer in one shot, the model is placed in a closed loop with the blockchain and allowed to *investigate*: it forms hypotheses, requests evidence, reads the evidence, and revises—for a bounded number of rounds—until each hypothesis is confirmed or refuted. Figure 2 shows one turn of the cycle.

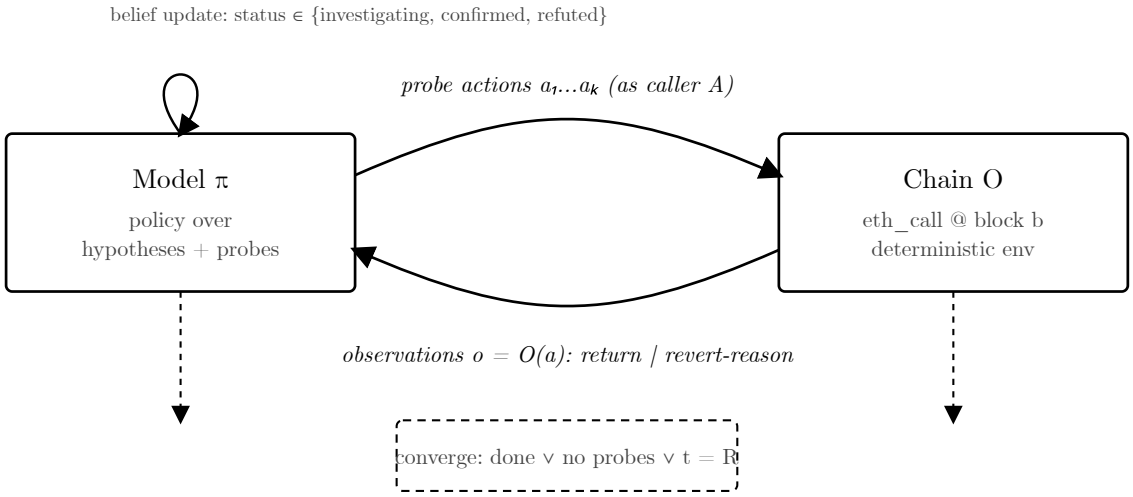

Figure 2: One round of the recursive hunt. The model acts as a policy that emits a set of hypotheses together with up to six read-only probe actions; the chain evaluates each probe deterministically at the current head and returns an observation; the model folds observations into an updated belief over each hypothesis' status and either requests more evidence or halts.

Formalization

Fix the contract's on-chain state s_b at block b; within a hunt it is effectively constant, so the environment is deterministic and stateless across probes. Let c be the source, Σ the function catalogue, and A

the adversary address. The model is a policy π_ϑ that, given the interaction history, emits at round t a set of hypotheses H_t and a batch of probe actions:

$$(H_t , a_{t,1} \ldots a_{t,k}) = \pi_\vartheta(c , \Sigma , o_{<t}) , k \le 6 \tag{1}$$

Each probe action is a concrete call $a = \langle \text{fn, args, from, value} \rangle$ with $\text{fn} \in \Sigma$ and $\text{from} = A$ by default. The oracle O evaluates it against live state and returns either a decoded return value or a revert reason:

$$o = O(a) = \texttt{eth\_call}(a ; s_b) \tag{2}$$

Every hypothesis $h \in H_t$ carries a status in {investigating, confirmed, refuted}. The belief update is the model's own re-reading of its hypotheses in light of the newest observations—a soft, learned transition rather than a hand-coded rule:

$$H_{t+1} = U_\vartheta(H_t , o_t) \tag{3}$$

A hypothesis is treated as *confirmed* when the model exhibits a witness—a concrete permissionless call whose simulated observation satisfies the exploit predicate φ_h the model attached to it:

$$\text{confirm}(h) \Leftrightarrow \exists\, a : \text{from}(a) = A \wedge O(a) \vDash \varphi_h \tag{4}$$

The loop halts at the first round T where the model signals completion, requests no further probes, or the round budget R is exhausted:

$$T = \min \{ t : \text{done}_t \vee k_t = 0 \vee t = R \} \tag{5}$$

The label *reinforcement-style* is deliberate and honest about what it is not. There is no gradient step and no weight update: ϑ is frozen. The reinforcement is *in-context*—the environment returns a signal (a revert, an unexpected balance, an invariant that fails to hold), and that signal conditions the next action within the same episode, exactly the actor-plus-environment shape of a control loop, realized entirely through the model's context window rather than through its parameters. It is closest in spirit to reason-act agents that interleave thought and tool use, and to self-reflective agents that revise from execution feedback [7, 8].

Why the loop beats one shot

A single-shot auditor must commit to claims it cannot check; it hedges, and it hallucinates preconditions. The loop replaces assertion with evidence. Concretely, it lets the model (i) read the *actual* values that decide reachability—a fee that is currently zero, a reserve that is currently imbalanced—rather than guessing them; (ii) discover the correct arguments for a call by observing why a first attempt reverts; and (iii) abandon hypotheses that reality contradicts, which is what keeps the confirmed set small and precise.

7 The simulation oracle

Every probe is an `eth_call`: an execution of the contract's bytecode against current state that computes a result but commits nothing. This choice is the spine of the system's safety and much of its power, and its semantics deserve to be stated exactly.

- **Non-mutating.** An `eth_call` runs the full EVM but discards the resulting state. No storage is written, no event persists, no gas is truly spent. Probing is free and side-effect-free by construction.

- **Caller spoofing.** The call's `from` can be set to any address without a signature. This is precisely what lets the system *prove permissionless reachability*: by simulating as an arbitrary non-owner address, a successful (non-reverting) call is evidence that *anyone* can make it, which is the whole scope of Section 2.
- **Head-of-chain fidelity.** Probes execute against live mainnet state at the current block, so reachability reflects the contract's real balances, allowances, and reserves right now—not an idealized deployment.
- **Statelessness.** Probes do not compose: each runs against the same base state, so effects do not carry from one probe to the next. This is the oracle's fundamental limitation, and Section 13 treats it directly.

Calls are encoded and decoded against the ingested ABI; arguments are marshalled to EVM types (addresses as hex strings, integers as decimal strings scaled to decimals), and results are decoded back to readable values or, on failure, surfaced as the raw revert reason—which is itself signal the model consumes. RPC access is made resilient by rotating through a redundant set of public gateways and failing over on error.

8 Exploit synthesis and argument repair

Confirming a hypothesis often requires a call whose arguments are non-obvious—an amount that must exceed a reserve, an address that must already hold a balance. A first guess frequently reverts. Rather than give up, the system runs a *simulate-then-repair* fixpoint: it simulates the candidate call, and on revert it feeds the revert reason and the relevant on-chain readings back to the model to correct the arguments, then re-simulates—bounded to a small number of attempts. Only a call that would *succeed* in simulation is treated as a landed exploit.

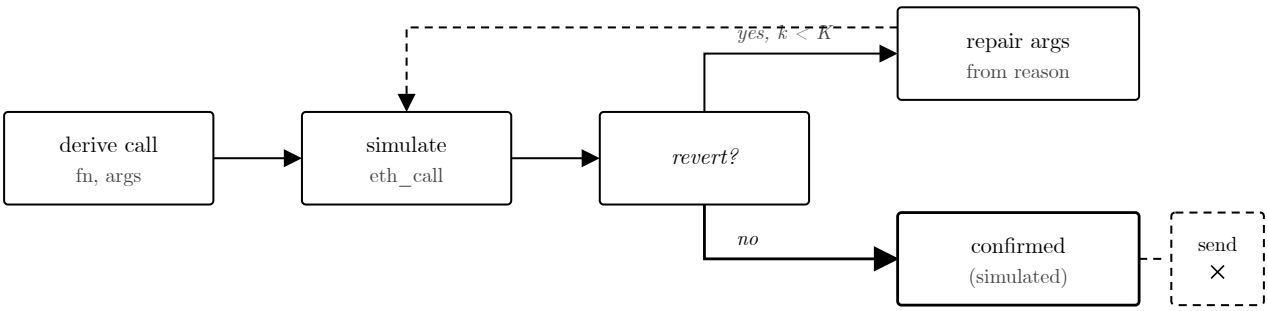

Figure 3: The exploit-repair ladder. A derived call is simulated; a revert routes the reason back to the model to correct arguments and re-simulate, up to K attempts. A non-reverting simulation confirms exploitability—and the ladder *stops there*. The broadcast step (dashed, severed) that a live-attack tool would place after "confirmed" is removed in the reference build.

Formally the repair is a bounded fixpoint. With e_{k} the revert reason at attempt k and ϱ the model's correction operator:

$$a_{\mathrm{k}+1} \;=\; \varrho\big(\; a_{\mathrm{k}} \;,\; e_{\mathrm{k}} \;,\; O(a_{\mathrm{k}}) \;\big) \quad , \;\; \text{stop when } \neg\mathrm{revert}(O(a_{\mathrm{k}})) \;\vee\; k = K \tag{6}$$

In a live-attack system the terminal state "would succeed" is the trigger to sign and broadcast. In Mephistopheles it is the terminal state, full stop: the ladder reports a simulated confirmation and never advances to a transaction (Section 11).

9 Structured decoding and reliability

Every model turn must return a machine-consumable object, because the next stage is code, not a human. The system requests JSON-mode generation and validates each reply against the stage's schema. Malformed output triggers exactly one stricter retry—a re-prompt demanding a single valid, minified object with no prose, no code fences, and no chain-of-thought markers—after which a tolerant parser makes a final salvage attempt. Retry messages are not persisted into the conversation, so a formatting stumble never pollutes the reasoning history.

This discipline is what lets a loop of six equations above be implemented as a reliable program: hypotheses, probe batches, per-finding tests, and verdicts are all typed artifacts the orchestrator can execute, render, and diff, not free text it must parse heuristically. Generation temperature is held low to favor determinism over variety, appropriate for an evidence-gathering task rather than a creative one.

10 Verdict synthesis and interactive refinement

After the hunt converges, a final model pass compresses the union of static findings and confirmed hypotheses into a single verdict—a one-line headline, a short posture assessment for the unprivileged attacker, and two explicit columns: what is *exploitable by anyone* and what is *safe or out of scope*. Grounding the summary strictly in the enumerated findings keeps the headline from drifting beyond what was actually shown.

The report is not a dead end. A natural-language layer lets a user interrogate the results—"what is the worst issue," "re-run focusing on reentrancy"—and the model answers with a typed action tag that can re-trigger the audit or the hunt, or direct the user to refine a single finding's test arguments in place. Each finding's attached test can be re-simulated on demand, and its arguments edited in plain English through the same repair operator of Section 8. The whole surface is thus a live instrument: every claim on screen is one click from a fresh simulation against current state.

11 Safety architecture: the simulation-only boundary

An automated system that could discover *and execute* exploits against arbitrary live contracts would be an instrument of theft. The reference build is not that system, by construction rather than by policy. Discovery is grounded entirely in read-only simulation, and the single code path that could sign and broadcast a transaction is disabled: it raises rather than sends. Because that path is the only chokepoint through which any state-changing call could reach the network, severing it neutralizes *every* route to a live exploit at once—the automatic post-audit hunt, a manual "run" button, and the argument-refinement flow alike.

> ***Design boundary.*** The confirmation criterion of Section 6 and the repair ladder of Section 8 both terminate at a successful `eth_call`. A successful simulation is the strongest evidence the system will produce: it demonstrates that a call *would* succeed against current state without ever committing it. The gap between "would succeed in simulation" and "is broadcast on-chain" is exactly the gap between an audit and an attack, and the build keeps that gap open on purpose.

This boundary is what makes the difference between a disclosure aid and a weapon. It also reframes the system's output: not a pending transaction, but a reproducible proof-of-reachability that a contract's authors can inspect, and that any reader can re-run for themselves against a state they choose.

12 Evaluation protocol

We describe how the system should be assessed rather than reporting headline accuracy numbers, because a single figure hides the two error modes that actually matter for a discovery tool and invites overtrust in an instrument whose outputs must be independently verified.

Ground-truth corpora

Two complementary corpora make a fair test. The first is a set of *historically exploited* mainnet contracts with documented permissionless root causes, replayed against a fork pinned to a block *before* the incident—the system should surface the known class. The second is a set of *audited-clean* contracts of comparable complexity—the system should return an empty permissionless-findings set, testing its restraint. Because scope explicitly excludes owner powers, ground truth must be labelled at the same scope, or the system will be penalized for correctly ignoring centralization risk.

Metrics that separate the error modes

Table 4: The measurements that characterize a discovery tool, and what each one guards against.

Measure	Question it answers	Failure it exposes
Recall @ class	Of known permissionless bugs, how many are surfaced?	Missed exploits (silent, dangerous)
Confirmed precision	Of `confirmed` findings, how many reproduce independently?	False alarms that erode trust
Simulation fidelity	Does a "confirmed" call actually succeed on a clean fork?	Oracle mis-encoding, stale state
Rounds-to-converge	How much loop budget does confirmation cost?	Cost / latency of depth
Scope adherence	Fraction of findings that are truly permissionless	Owner-gated leakage past the prompt

The distinction between *reported* and *confirmed* is the crux: only the confirmed set—findings backed by a witnessing simulation—should be held to precision, and every confirmed finding should be re-executable on an independent fork as an acceptance test. A tool whose confirmed precision is high and whose recall is honestly bounded is more useful than one advertising a single inflated accuracy.

13 Limitations and failure modes

- **Probe statelessness.** The oracle evaluates each call against the same base state, so genuinely multi-step exploits—deposit, then manipulate, then withdraw—cannot be executed as a composed sequence in simulation. The system approximates them with the closest single entry call and the model's narrative of the remaining steps; a whole class of sequential attacks is therefore reasoned-about but not witnessed. A stateful fork with persistent overrides would close this gap and is the most consequential extension.

- **Single-contract horizon.** Ingestion is one address. Cross-contract and protocol-composition exploits—the economic heart of many real incidents—are outside what the source-and-ABI view can capture.
- **Soft scope enforcement.** The permissionless boundary lives in the prompt and in the `from` field, not in a proof. A model can still mislabel an owner-gated path; scope adherence is therefore a measured quantity, not a guarantee.
- **Source truncation.** Large contracts are truncated before entering a prompt, so logic in the tail may never be reasoned about. Findings are bounded by what fits the context.
- **Model-bound reasoning.** Hypothesis quality is the model's; a weaker local model yields shallower hypotheses and more hallucinated functions (mitigated, not eliminated, by the catalogue constraint and JSON discipline). The oracle grounds *reachability*, not the *relevance* of what is reached.
- **Verified-source dependence.** Unverified contracts are out of reach by design; the system reasons over intent, and intent requires source.

None of these are incidental bugs; they are the shape of the design. The honest summary is that Mephistopheles is a strong generator of *grounded, single-call hypotheses* and a deliberately conservative confirmer—an accelerant for a human auditor, not a replacement for one.

14 Related work

Automated smart-contract analysis has three established lineages. *Symbolic execution* enumerates reachable paths and solves for dangerous states—Oyente [1] introduced the approach and Mythril [3] and Manticore [5] carry it forward. *Static analysis* pattern-matches over intermediate representations for known-bad constructs, exemplified by Slither [2]. *Property-based fuzzing* mutates transaction sequences against user-specified invariants, as in Echidna [4], while formal verification tools such as the Certora Prover [6] discharge specifications against the bytecode. These methods excel at what is enumerable and specifiable, and are characteristically weak on *intent*—a contract can be path-safe, invariant-clean, and still economically exploitable in a way no written property anticipated.

Mephistopheles sits alongside a newer line that uses language models as the hypothesis engine and pairs them with execution feedback, in the tradition of reason-and-act agents [7] and self-reflective agents that revise from tool results [8]. Its distinctive commitments are the strict permissionless scope, the use of caller-spoofed `eth_call` against *live* state as the grounding oracle, and the simulation-only safety boundary. It is complementary to the symbolic and fuzzing tools above rather than a substitute: the model proposes where those tools would have to enumerate, and the chain adjudicates.

15 Ethical considerations and responsible use

Vulnerability-discovery automation is dual-use, and honesty about that is part of the design. Three commitments govern the reference system. First, **simulation-only operation**: the build cannot broadcast, so its outputs are evidence, not actions (Section 11). Second, **authorized targets**: the intended use is auditing contracts one owns or is authorized to test, and running against a local fork or testnet is the appropriate default for anything else. Third, **disclosure over exploitation**: a confirmed, reproducible proof-of-reachability is precisely the artifact a responsible-disclosure process needs, and the system is shaped to produce that artifact and stop.

This paper documents architecture and mechanism at the level a security researcher or the affected contract's authors need to understand and reproduce a finding. It deliberately omits an operational playbook for attacking third-party contracts for gain, because that is the line between describing a discovery instrument and handing over a weapon—the same line the build itself draws by severing the broadcast path.

16 Conclusion

Mephistopheles is a small thesis about where language models are useful in security: not as oracles of truth, but as tireless generators of hypotheses that a real environment can cheaply adjudicate. By placing a frozen model in a bounded, evidence-gathering loop with a live chain—and by holding that loop to simulation—the system turns the model's intuition into reproducible, permissionless-scoped proofs of reachability without ever putting funds at risk. The interesting engineering is not the model and not the node, but the disciplined, safe conversation between them.

---

17 References

[1] Luu, Chu, Olickel, Saxena, Hobor. *Making Smart Contracts Smarter* (Oyente). ACM CCS, 2016.

[2] Feist, Grieco, Groce. *Slither: A Static Analysis Framework for Smart Contracts* IEEE/ACM WETSEB, 2019.

[3] ConsenSys. *Mythril: Open-source tool. security analysis of EVM bytecode via symbolic execution*

[4] Grieco, Song, Cygan, Feist, Groce. *Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts*. ACM ISSTA, 2020.

[5] Mossberg, Manzano, Hennenfent, Groce, Grieco, Feist, et al. *Manticore: A User-Friendly Symbolic Execution Framework*. IEEE/ACM ASE, 2019.

[6] Certora. *The Certora Prover: formal verification for smart contracts*. Product documentation.

[7] Yao, Zhao, Yu, Du, Shafran, Narasimhan, Cao. *ReAct: Synergizing Reasoning and Acting in Language Models*. ICLR, 2023.

[8] Shinn, Cassano, Berman, Gopinath, Narasimhan, Yao. *Reflexion: Language Agents with Verbal Reinforcement Learning*. NeurIPS, 2023.

[9] Ethereum.

JSON-EIP-1474 and the execution-API spec.
RPC
spec-
ifi-
cation:
the
eth_ -
call
and
eth_ get-
Code
meth-
ods

[10] Solidity documentation.
Contract
ABI
Specification

---

18 Appendix A. System-prompt design principles

The two reasoning stages are steered entirely by their system prompts, which encode four recurring principles. Each is a lever pulled to keep the loop grounded and in-scope.

- **Fix the adversary in every prompt.** Both stages restate that the caller is an arbitrary non-owner and that anything role-gated is out of scope, so the model cannot drift into reporting centralization risk.
- **Constrain the vocabulary.** The model may reference only functions from the ingested catalogue, and is told to invent nothing—the single strongest defense against hallucinated calls.
- **Demand realism.** Arguments must be scaled to token decimals and use addresses with genuine on-chain state; attacker-controlled recipients are routed to one observable address so redirected effects are visible in a single place.
- **Force falsifiability.** Every finding must carry a single concrete call that best demonstrates it, converting prose into something the oracle can adjudicate.

19 Appendix B. Output schemas (abbreviated)

Each stage emits one typed object. The hunt schema is the interesting one—it carries both the evidence trail and the per-hypothesis witnessing calls.

```
// Stage II -- one turn of the recursive hunt
{
  "thoughts": "what was learned; what to test next",
  "hypotheses": [{
    "id", "title", "severity",
    "status": "investigating | confirmed | refuted",
    "explanation", "exploitation", "outcome", "evidence",
    "test": { fn, args, from, value, note }   // verifying call
  }],
```

```
  "probes": [{ hypothesisId, fn, args, from, value, why }],  // ≤ 6, read-only
  "done": false
}
```

The orchestrator executes each `probe` as an `eth_call`, appends the decoded results to the conversation, and returns control to the model for the next turn—the concrete realization of the update in equation (3).