# Content-Addressed Redaction with Local Restoration: Secret Handling for Tool-Using AI Agents

yxpil

## Abstract

Tool-using AI agents run on a user's machine but delegate cognition to a third-party model provider. Whenever a credential, phone number, or access token has to appear in a prompt so that the agent can act on it, the usual choices are to expose the secret to the provider or to prevent the agent from acting at all. We describe and evaluate a third option, implemented in the open-source agent runtime BrainTentacle: *content-addressed redaction with local restoration*. Before a request leaves the machine, every occurrence of a registered sensitive value, and every occurrence matched by an enabled regular expression, is replaced in the copy of the conversation that will be transmitted by a placeholder whose text is a prefix of the SHA-256 digest of the replaced value. When the model returns tool arguments, the dispatcher substitutes the real values back before execution, so the agent acts on a secret it never saw. Because the placeholder derives from the value's content rather than from an enumeration counter, the same secret carries the same token across turns and independently of registration order; this is what makes the token quotable by the model in a later turn. We contribute (i) a precise description of the mechanism against the artifact, (ii) a measurement harness that links the shipping implementation as a library and measures detector precision and recall on a 62-document corpus of realistic and adversarial payloads, round-trip fidelity, placeholder stability, and the collision behaviour of truncated digests, and (iii) an enforcement audit that identifies three concrete gaps in the current implementation, including a request path on which redaction is not applied at all. We report only measured quantities, and we state the limitations explicitly: the secondary audit layer is experimental and fails open by design, and the workspace path jail is lexical with no default root.

## 1 Introduction

A tool-using agent is a program that sits between a language model and a machine. The model emits a structured request to call a tool—run a shell command, read a file, send an HTTP request, invoke an MCP server—and a local dispatcher executes it, feeds the result back, and asks the model again Yao et al. [2023]. This loop is what makes agents useful, and it is also what makes them dangerous: the dispatcher is the component that holds authority, while the model is the component that decides what to do with it.

The model, however, is usually not local. In the common deployment the agent runs on the user's laptop but the model runs in someone else's data centre. Everything the agent sends is therefore disclosed to a third party, and the model's output is a proposal rather than an instruction—which is the right place to put the boundary between untrusted and trusted computation.

Now suppose the agent's task involves a secret: an API key for a service, a customer's mobile number, an internal hostname, a bearer token. The secret must reach the executor, because the executor is what will use it. Two common responses to that tension are known. The first is to give the model less capability: route every secret-bearing operation through a broker or proxy so that the model never handles the secret at all [Saltzer and Schroeder, 1975]. The second is to keep the capability but constrain the model: install an approval gate and an audit layer so that a possibly-hostile proposal is reviewed before it becomes an effect [OWASP Foundation, 2023]. The first family is expensive to build and gives up generality; the second family does nothing about disclosure to the provider.

We study a third option that sits between them, implemented in BrainTentacle, an open-source agent runtime whose core library is `crates/bit-core/src/hidden_code.rs`. The mechanism is *content-addressed redaction with local restoration*. Sensitive values are replaced by placeholders in the copy of the conversation that is about to be transmitted, and the real values are substituted back into the tool arguments after the model has produced them and before the dispatcher acts. In this design the model functions as an untrusted intermediary with respect to secrets: it can name a secret, quote it, copy it, and refer to it in a later turn, but it cannot read it, reproduce it, or send it anywhere.

The design property that makes this workable—and the property we make precise and measure—is that the placeholder is *content addressed*. Its text is a prefix of the SHA-256 digest of the value it replaces, not a counter and not an enumeration index. Two consequences follow directly. First, the mapping from secret to token does not depend on the order in which secrets were registered, on how many secrets exist, or on how many turns have elapsed; re-registering the same secret in a different session, or in a different position in the list, yields the same token. Second, because the token is a deterministic function of the value, two distinct secrets that happen to share a truncated digest prefix are mapped to the same token, which is a failure mode with a well-understood closed-form probability that we measure rather than assume.

We make three contributions.

1. **A description of the mechanism against an executable artifact.** Section 4 specifies the entry types, the placeholder derivation, the masking order, the restoration algorithm, the audience separation, the approval gate, and the sandbox boundary as they are implemented, citing file and line locations. We use a prose description rather than a design document precisely so that every claim can be checked against code that ships with the paper.

2. **A measurement harness over the shipping code.** The harness in `scripts/redaction_measure` links `bit-core` as a path dependency and drives the real public entry points `maybe_mask_messages` and `unmask_params`; no masking or restoration logic is reimplemented. It measures, on a 62-document corpus of realistic developer payloads and adversarial near-misses, the precision and recall of the three built-in detectors, byte-exact round-trip fidelity, the stability of the placeholder under reordering and re-registration, and the collision behaviour of the 6-hex and 8-hex prefix lengths. The corpus, the harness, and the resulting JSON are shipped with the paper.

3. **An enforcement audit, reported mostly as negative results.** Following the mechanism end to end through the request pipeline shows that it is not applied on every path. In the current tree the redaction step is invoked from exactly one function, and the two request paths used by the agent's own turn loop bypass it (Section 6.4). Two further gaps—pattern-matched secrets are masked but never restored, and the digest-prefix collision detector does not observe runtime-matched values—are described and quantified in Section 6.4. We regard these as the most useful output of the study: a mechanism that is described as a property of the system is in fact a property of one call site, and only measurement of the kind in Section 6 distinguishes the two.

The remainder of the paper is organised as follows. Section 2 introduces tool-using agents, prompt injection, and the state of the art in secret handling inside agent contexts. Section 3 states the threat model, including what the mechanism explicitly does not defend against. Section 4 describes the design. Section 5 describes the implementation and the request pipeline. Section 6 reports the evaluation. Section 6.5 discusses threats to validity, Section 7 positions the work, Section 8 states the limitations, and Section 9 concludes.

# 2 Background

## 2.1 Tool-using agents and an untrusted model

An agent loop has four moving parts: a model, a conversation history, a registry of tools, and a dispatcher. The model is queried with the history plus the tool schemas; it replies either with text or with a set of tool calls; the dispatcher validates and executes them; the results are appended to the history and the loop repeats. Two properties of this loop matter for security. First, the model is a *relay* for the tool arguments: whatever the model emits is executed locally, so a model that is wrong, confused, or hostile can cause local side effects. Second, the model is a *disclosure surface*: the history that is transmitted can contain anything the user typed, anything a tool returned, and anything a prior turn established.

These two properties have been exploited in both directions. Indirect prompt injection [Greshake et al., 2023] uses content that an agent fetches—a web page, a repository file, an e-mail, an MCP tool result—to steer the model into tool calls that exfiltrate data or escalate privilege. Direct injection attacks the model's own instructions [Greshake et al., 2023, Evertz et al., 2024]. On the other side, the provider itself is a third party whose retention and training practices are outside the user's control, and the contents of a transmitted request are exactly what a provider's staff, logging, or abuse monitoring can see.

The observation that motivates this paper is that these two risks are usually addressed by the same component. If the agent holds secrets and the model can read the context, then disclosure to the provider and manipulation by injected content are entangled: sanitising the context for the provider also removes the injected instruction from the model's reach. Secret-redaction mechanisms attempt to separate them, by removing the secret from the model's reach while leaving the rest of the context—including possibly hostile instructions—intact.

## 2.2 Prompt injection and context hygiene

Prompt injection is usually framed as a problem of *trust*: the agent cannot distinguish data from instructions. Mitigation has therefore tended towards capability restriction—sandboxed interpreters, allow-listed tools, approval gates—and towards instruction-level defences such as delimiting untrusted content [Greshake et al., 2023, Evertz et al., 2024], or training a model to treat privileged instructions as higher authority than content embedded in data [Wallace et al., 2024]. LLM-based detectors, which ask a second model whether a proposed action is benign, are a widely used variant of this approach and have been studied at scale, with the finding that they are useful but imperfect and sensitive to prompt wording [Zheng et al., 2023].

We take the complementary position. Instead of trying to make the model robust to hostile instructions, we make the model *unable to harm the secret* even if it is fully compromised. A model that has been injected and instructed to exfiltrate the API key can only exfiltrate a token it cannot read. This is a strictly weaker assumption than model robustness and it composes with the instruction-level defences rather than replacing them.

## 2.3 Secret handling in agent contexts

Three families of prior work are relevant.

The first family keeps secrets out of the model's reach by construction. Virtual application frameworks, capability-based proxies, and API gateways substitute credentials at the point of use and expose only a scoped handle to the model. The advantage is a very strong guarantee; the disadvantage is

that it requires the platform to know, ahead of time, which operations need which credential, and it does not generalise to free-form shell use.

The second family audits or filters what the model is about to do. Work on LLM-as-judge risk assessment [Zheng et al., 2023] and on agentic vulnerability taxonomies places a classifier or a human in the loop between the model and the executor. This addresses *side effects* but not *disclosure*: the classifier and the human both see the real arguments, and the provider has already seen the context.

The third family is scrubbing, which is closest to the mechanism studied here. Secret-scanning tools detect credential-shaped strings and mask them before a transcript is shared or logged. The limitation is structural: a scanner can label a string as a credential only if it can recognise the shape, and a generic prompt can elicit a credential-shaped string that carries no recognisable marker. Our mechanism inherits that limitation exactly: it is a syntactic, pattern-based defence, and Section 6 measures how incomplete it is rather than asserting that it is adequate.

Our contribution relative to this third family is not the act of masking but the *addressing scheme* and the *round trip*. A scrubber that replaces a secret with $\star\star\star$ or with a positional token destroys the model's ability to refer to the secret; a tool argument that must contain the real value cannot be constructed from the masked transcript. Our scheme reconstructs the value at the point of use, and the content-addressed derivation is what makes the reconstruction unambiguous in the presence of many secrets, many turns, and an arbitrary registration order.

# 3 Threat model

## 3.1 Principals, assets, and adversaries

The *local user* owns the machine and is trusted. The *executor* (the agent's tool dispatcher and the tools it invokes) is trusted: it holds the authority to read and write the machine. The *model provider* is the party that receives requests; we assume it is honest-but-curious, in the sense that it processes requests as specified but is not part of the trust boundary for the user's secrets. The *model* is a component of the provider and is treated as untrusted with respect to secrets, and separately as potentially adversarial with respect to the machine.

The protected assets are (i) the values of registered secrets—credentials, API keys, personal identifiers—and (ii) the user's authority over the machine. The mechanism studied here protects (i) against disclosure to the provider and against exfiltration through the model; it does not protect (ii), which is the job of the approval gate and the sandbox.

We distinguish three adversaries.

**A1: the honest-but-curious provider.** The provider observes the full request body. It is not malicious, but its operators, logs, caches, or abuse monitoring may expose the request contents to parties outside the user's trust boundary. A1 is the adversary the mechanism directly addresses.

**A2: an injected instruction.** Content retrieved by the agent—a web page, a file, an MCP resource, a tool result—contains text that tries to make the model disclose the secret, for example by instructing it to include the value in a subsequent tool argument that will be transmitted outward. A2 is the classic indirect prompt injection adversary [Greshake et al., 2023]. Our defence against A2 is indirect but strong: the model cannot comply because the value is not present in its context. What A2 *can* do is induce the model to transmit the placeholder, which discloses the fact that a secret exists and its stable identifier, but not its value.

**A3: a malicious tool or MCP server.** A tool the agent talks to may return hostile content, and an MCP server under a third party's control may do so by design. This is A2's entry vector; the mechanism is agnostic to it.

## 3.2 Explicit non-goals

We state plainly what the mechanism does not do, because several of these gaps are properties of the implementation rather than of the idea.

- It does not prevent the *executor* from leaking the secret. Once restored, a secret is an ordinary argument to a local tool. If a tool writes it to a file in the workspace, it is on disk.
- It does not constrain what the model may *do*. A model that has been injected can still call any tool it is permitted to call. Redaction reduces the value of the compromise; it does not remove the capability.
- It does not hide the *existence* of a secret, nor its stable identifier: the placeholder is transmitted in clear text and is stable over time. If the token space is small enough, an observer of several transcripts can accumulate a dictionary of tokens.
- It does not cover secrets that the model must read as *tool output* and then use in a later turn. Only values registered in advance, or matched by an enabled pattern, are covered. A credential that appears for the first time in a file the agent reads is not covered unless a detector fires on it.
- It provides no cryptographic protection of the secret at rest. The registry itself is stored locally under the platform's device key; that is a different mechanism with a different threat model.
- It does not defend against a local attacker with the user's privileges, nor against a malicious human approver.

# 4 Design

## 4.1 The redaction layer

A redaction *entry* is a record with a kind, a value, an optional alias, and an enabled flag. Two kinds exist.

A *value entry* names a specific string. When enabled and non-empty, the entry contributes two substitutions to the masking step: the string itself is replaced by its placeholder, and, if the entry carries an alias, the alias is mapped back to the string at restoration time. An alias therefore changes what the model sees—the model sees a human-readable token such as a pseudonym instead of an opaque digest—without changing the fact that the model cannot recover the original.

A *pattern entry* names a regular expression. Three expressions are built in and are compiled once into static caches: a Chinese mobile number pattern, an e-mail address pattern, and an API-key pattern covering a set of well-known key prefixes. A pattern entry applies its expression to the payload and replaces *each match separately*, so that two different matches receive two different tokens unless their contents coincide. A user may register an arbitrary regular expression as a custom pattern entry.

The *placeholder* for a replaced string $v$ is

$$P(v) \;=\; \texttt{[HC:}\,\big[\mathrm{sha256}(v)\big]_{1..h}\,\texttt{]},$$

where $[x]_{1..h}$ denotes the first $h$ hexadecimal characters of the digest and $h$ is either 6 or 8. The prefix length $h$ is not a configuration knob but a function of the registry: the implementation computes the 6-hex prefix of every registered value and, if two of them agree, widens the prefix to 8 hex characters for the whole registry. This is a coarse but serviceable answer to the collision problem: it detects collisions among values that were registered, and it does so by widening rather than by disambiguating, so the mapping remains injective on the registered set.

## 4.2 Masking order and the copy semantics

Masking proceeds in two passes. Pattern entries are applied first, by replacing every match of every enabled pattern. Value entries are applied second, by literal substring replacement, alias first if present. The order matters: because pattern replacement happens first, a literal value entry whose text overlaps a pattern match will not find its text in the payload any more, and the value entry will silently have no effect on that occurrence. Applying value entries first would have the mirror problem.

The masking operates on a *copy* of the conversation. The transcript that is stored locally keeps the plaintext; only the copy that is handed to the provider is rewritten. This is the right default for an audit trail, but it means that the local database is a plaintext-at-rest store of the same secrets that the mechanism is protecting from disclosure, and any local attacker with database access recovers them without any of the machinery in this paper.

## 4.3 The instruction to the model

If any placeholder was produced, the implementation appends a short system note to the last user message of the transmitted copy. The note states, in the product's own Chinese wording, that `[HC:xxxxxx]` denotes a redacted sensitive value, that tool calls quoting these values will be restored locally at execution time, and that the model should quote the placeholders verbatim rather than guessing or inventing their contents. The note is appended to the user message rather than to a dedicated system role because the transmitted request formats differ in how they carry system prompts; the note is deliberately written in the user's language, on the assumption that the conversation is.

This note is a prompt-level defence, and prompt-level defences are weak: it can be ignored, and its wording is not verified to change model behaviour. The measurement in Section 6 therefore does not claim that models comply with it. What the mechanism relies on is not compliance but addressing: if the model does quote the placeholder verbatim, the token is recognised; the failure mode of non-compliance is a broken tool call, not a disclosed secret.

## 4.4 Local restoration

Restoration is the inverse operation, performed on the tool arguments after the model has produced them and before the dispatcher acts. The dispatcher builds a map from placeholder to real value by taking, for each enabled non-empty value entry without an alias, the pair $(P(v), v)$ and, for each alias entry, the pair $(\text{alias}, v)$. The map is rebuilt on every call rather than persisted, which keeps it consistent with the registry at the cost of a hash computation per invocation.

The arguments are a JSON value, and restoration is applied recursively: for an object, every value is restored; for an array, every element is restored; for a string, every occurrence of every key of the map is replaced. A placeholder that is not in the map is left exactly as it is. This is deliberate and important: a model that hallucinates a token `[HC:000000]` that does not correspond to any registered value produces a tool argument containing that literal string, which is a visible anomaly, rather than a silent substitution of some other secret.

Two consequences of the implementation follow from the map being an ordinary hash map with string keys and the substitution being a sequence of `replace` calls over its entries.

- **Restoration is order-dependent when aliases overlap.** The substitution loop iterates over the map in hash order. If alias `bob.smith` maps to a value that itself contains the shorter alias `bob`, the result depends on which is applied first, and hash order varies between processes. Section 6.4 reports a measurement of this non-determinism.

- **Short value entries can corrupt longer placeholders.** Value entries are substituted literally over the whole payload. If a registered value happens to be a short string that occurs inside another entry's placeholder text—for

example a two-character value that appears in a six-hex-character digest—the literal replacement rewrites the placeholder and the entry can no longer be recognised at restoration time. Section 6.4 exhibits a concrete instance.

### 4.5 Separation of audiences

The design distinguishes two audiences with different information needs, and the mechanism gives them different views of the same tool call.

The *model* is an untrusted audience. It receives masked arguments, and it is the only party that sees the placeholder.

The *auditor*—a second model acting as a reviewer, or a human approver—is a partially trusted audience. In the shipped implementation the audit layer is handed the *masked* parameters, so a reviewing model cannot itself leak the secret by repeating it in its verdict, and a transcript of the audit exchange contains no secret. The human approval dialog, by contrast, is shown the *real* parameters, because a human cannot authorise an action whose arguments they cannot read.

The *executor* is a trusted audience and receives the real parameters.

The audit layer is a second model call, implemented in `crates/bit-core/src/l2pass.rs`. Its prompt states, explicitly, that the parameters are data and never instructions, and asks for a JSON verdict of the form allow or deny-with-reason. The response parser tolerates a model that wraps the JSON in a code fence by taking the substring between the first opening brace and the last closing brace. The layer has a 20-second timeout and, importantly, is *fail-open*: a timeout, a network error, or an unparseable response yields a verdict of *unavailable*, which is treated as a request for human approval rather than as a denial. We discuss this trade-off in Section 8.

### 4.6 The approval gate

Tools may be executed in one of three modes. In `allow_all` mode every tool call proceeds. In `auto` mode, a call proceeds without human review only if the tool passes a safety predicate, and in `ask` mode no call proceeds without human review.

The safety predicate in `auto` mode combines a short allow-list of sideless tools with a syntactic test on the tool name: the name must be exactly one of a set of read-only verbs, or must begin with one of a set of read-only prefixes. The anchoring is deliberate. A containment test would classify a tool named `bread_maker` or `already_exec` as read-only, and an agent that may register its own tools could then obtain code execution without review by choosing a name that contains a read-only keyword. Anchoring the test at the start of the name removes that class of trick, at the cost of rejecting tool names that merely happen to begin with a read-only word.

The interaction between the gate and the audit layer is worth stating precisely, because it is easy to misread. A tool call that the gate would auto-pass may still be audited if the operator has enabled auditing of auto-passed tools, in which case a denial from the audit layer stops the call. A tool call that the gate would not auto-pass is always audited first, and if the audit layer is unavailable, denied, or simply not enabled, the call proceeds to human approval. The gate is therefore the primary control and the audit layer is an additional, optional, fail-open filter in front of it.

### 4.7 Sandbox boundary

The system has two sandboxing layers, and it is important to be precise about their strength.

The first is a *lexical path jail* for filesystem and shell tools. It normalises a candidate path by removing `.` and `..` segments textually, anchors relative paths to a configured workspace root, and rejects absolute paths whose canonicalised form does not start with that root. This is a normalisation-and-prefix test on a string, not on a file descriptor. It does not resolve symbolic links, and it does not guard against a *directory* inside the root that is a link to a location outside the root. More importantly, the root is not set by default in the desktop configuration: when no workspace root is configured, the jail returns the candidate path unchanged. In that configuration the layer provides no containment at all, and the mechanism should be described as a guard rail that must be switched on, not as a sandbox that is always in force.

The second layer is a *resource-budgeted script sandbox*. Scripts written in a small embedded scripting language are executed by an engine configured with an expression depth limit, an operation count limit, and a wall-clock budget, and the engine aborts when the deadline passes. Network access from within the sandbox is provided by a single function whose timeout is the smaller of the remaining budget and a fixed cap, and whose response body is truncated before it is handed back to the script. This layer bounds the resources a script can consume; it does not, by itself, bound what the script can reach on the filesystem, which is the job of the path jail.

A third execution path—running an interpreter on a script file—is a plain subprocess with a timeout and a cap on the captured output, and should not be described as sandboxed in any strong sense.

## 5 Implementation

### 5.1 Module and state

The mechanism lives in `crates/bit-core/src/hidden_code.rs`, a single module of the agent's core library. The library is a Rust crate of roughly 27.7 thousand lines across 53 files, and is consumed by an Electron front end through a Node-API binding and by a terminal front end. The registry of redaction entries is held in the shared application state behind a mutex and is persisted in the local database under the platform's device key. A configuration flag enables or disables the whole feature; when it is disabled, the masking step is a no-op.

The three built-in regular expressions are compiled into lazily initialised statics, so the cost of compiling them is paid once per process and not per request. The regular-expression

engine is the standard Rust `regex` crate The Rust Project Developers [2024], which compiles patterns to finite automata and therefore has no catastrophic backtracking behaviour; this matters because the patterns are applied to every outgoing message.

### 5.2 Where the masking happens

The masking step is invoked from a single place in the tree. The non-streaming, single-shot request function calls the masking routine on the message list and then dispatches to the protocol-specific request builders, so all of the OpenAI, Anthropic, and Gemini request shapes inherit the redaction. The code comment at that call site describes it as the single convergence point for sending to the model.

The streaming and native-tool-calling request paths do not call it. The agent's own turn loop chooses between a native function-calling path and a streaming text path, and both of those construct their request bodies directly. Section 6.4 quantifies the consequences.

### 5.3 Restoration point

Restoration happens at the top of the tool dispatcher, before any tool runs. The dispatcher obtains the real parameters by applying the restoration routine to the JSON arguments, and it uses those real parameters for the approval dialog, for the audit layer's tool-name argument, and for the actual invocation. The masked parameters remain what the model produced and what is recorded as the model's proposal.

### 5.4 Round-trip as an algorithm

For reference, the two operations in algorithmic form are as given in Algorithms 1 and 2. The masked copy is produced by Algorithm 1 and consumed by Algorithm 2.

**Algorithm 1** Content-addressed masking of a transmitted payload

**Require:** entries $E$ (enabled redaction entries), payload $x$, prefix length $h$
**Ensure:** masked copy $x'$

$x' \leftarrow x$
**for all** p **do**attern entry $e \in E$ (in registration order)
  $x' \leftarrow$ replace every match of $re(e)$ in $x'$ by $P(m)$, where $P(m) =$ `[HC:` $\mathrm{sha256}(m)[1..h]$`]`
  $h \leftarrow h$ ▷ unchanged
**end for**
**for all** v **do**alue entry $e \in E$ (in registration order)
  **if** $e.alias \neq \epsilon$ **then**
    $x' \leftarrow$ replace every occurrence of $e.value$ in $x'$ by $e.alias$
  **else**
    $x' \leftarrow$ replace every occurrence of $e.value$ in $x'$ by $P(e.value)$
  **end if**
**end for**
**if** $x'$ contains `[HC:` **then** append the redaction note to the last user message
**return** $x'$

## 6 Evaluation

We report three kinds of result: the outcome of the project's own test suite, a measurement of the redaction mechanism produced by a harness we ship with this paper, and an enforcement audit of the request pipeline.

### 6.1 Test suite

The core library carries unit tests inside the modules and one integration test file. Counting the test attributes in the source tree gives 239 unit tests in `crates/bit-core/src` (215 `#[test]` and 24 `#[tokio::test]`) and 12 integration tests in `crates/bit-core/tests/security_integration.rs`. Six of the unit tests are platform-gated and do not exist in the configuration we ran: four are marked `#[cfg(unix)]` in the script runtime and two are marked `#[cfg(not(windows))]` in the console codec. The integration file covers encryption round trips, tamper detection, message authentication, plugin manifests, listen-address normalisation, path sanitisation, and rejection of injection payloads in host normalisation. The sandbox module contributes seven unit tests of its own, one of which exercises a parent-directory traversal, and the plugin module carries a sanitisation test. The repository also ships 60 scripts under `e2e/` (56 `.cjs`, 3 `.py`, 1 `.ps1`), which are executed against a desktop build and report pass or fail rather than numeric outcomes.

We ran the suite as documented by the project, namely `cargo test -p bit-core -no-fail-fast`. The command executes 233 unit tests and 12 integration tests; all 245 pass, with no failures, no ignored tests, and no filtered-out tests.

We report this count rather than the figure quoted in the project documentation. The documentation states a baseline of 245 tests, which is exactly the number the command executes in this configuration: 233 unit tests and 12 integration tests. The source tree contains 239 unit-test attributes, six of which the platform gates remove; a naive search for the attribute text finds 240 because one occurrence is inside a comment. A test count is a property of a commit and a target, not of a project. We also note, and treat as a limitation, that the project's own testing document records that the test suites were generated by a large language model; the tests are therefore evidence about the code as written, not independent evidence of correctness.

### 6.2 Measurement of the redaction mechanism

**Harness.** The harness lives in `scripts/redaction_measure` and is a small Rust program whose manifest de-

**Algorithm 2** Local restoration of tool arguments

**Require:** entries $E$, tool arguments $v$ (JSON)
**Ensure:** real arguments $v'$

$m \leftarrow \emptyset$
**for all** v **do**alue entry $e \in E$ (enabled, non-empty)
  **if** $e.alias \neq \epsilon$ **then**
    $m \leftarrow m \cup \{(e.alias,\ e.value)\}$
  **else**
    $m \leftarrow m \cup \{(P(e.value),\ e.value)\}$
  **end if**
**end for**
$v' \leftarrow \mathrm{restore}(v, m)$
**for all** o **do**bject member $x \in v'$
  $x \leftarrow \mathrm{restore}(x, m)$
**end for**
**for all** a **do**rray element $x \in v'$
  $x \leftarrow \mathrm{restore}(x, m)$
**end for**
**return** $v'$

clares `bit-core` as a path dependency. It builds a live application context in an isolated data directory, populates it with a chosen set of entries, and then calls the two public entry points that the product itself uses: the masking routine for the outgoing payload and the restoration routine for the tool arguments. Nothing about masking or restoration is re-implemented in the harness.

The harness does, however, re-implement one thing: the *analysis* of which substrings the implementation replaced. It tokenises the masked payload into literal runs and placeholder tokens; the literal runs are matched back onto the original to fix the position of each placeholder; and the replaced span is then recovered by a brute-force search over the character boundaries of the original for the span whose SHA-256 digest starts with the observed prefix. Every recovered span is therefore confirmed by the implementation's own hash function rather than by a second copy of its regular expressions. Where several candidate spans are consistent with the observed prefix, the longest is selected, which matches the greedy behaviour of the built-in patterns.

**Corpus.** The corpus contains 62 documents in two subsets. The *realistic* subset (30 documents) consists of developer-style payloads in Chinese and English: chat transcripts, support tickets, shell transcripts, configuration excerpts, log lines, and e-mail bodies, each annotated with the secrets it contains. The *adversarial* subset (32 documents) consists of near-misses that a naive detector is likely to over-match or under-match: twelve- and thirteen-digit numbers, a country-code-prefixed mobile number, a landline with a hyphen, addresses at hostnames that are not routable domains, an address inside a longer string, a truncated key prefix, a key body one character short of the threshold, a non-uppercase key prefix, a live-mode prefix variant, a modern fine-grained personal-access-token prefix, a cloud API key prefix, a PEM block, an eighteen-digit national identifier, a nineteen-digit card number, a version string, a numeric amount with a decimal point, a twelve-digit order identifier, a hexadecimal digest, a repeated value, a mobile number embedded in a Windows path, a private IPv4 address, and a full SHA-256 digest. Each document labels its secrets by class, where the classes are the three built-in detectors plus an *other* class for secrets in formats that the built-ins do not target; the *other* class is reported separately as a coverage gap rather than folded into a recall figure.

Table 1: Per-detector precision and recall, measured by the shipped harness. Each detector was evaluated in isolation over the corpus described in Section 6.2; *matched* counts spans replaced by the implementation, *FP* false positives, *FN* annotated occurrences of the class that were not replaced.

| Detector | Subset | Matched | FP | FN | Precision / Recall |
|---|---|---|---|---|---|
| `phone` | realistic | 13 | 0 | 0 | 100.0 / 100.0 |
| `phone` | adversarial | 5 | 1 | 2 | 80.0 / 66.7 |
| `email` | realistic | 14 | 0 | 0 | 100.0 / 100.0 |
| `email` | adversarial | 4 | 1 | 1 | 75.0 / 75.0 |
| `apikey` | realistic | 12 | 0 | 0 | 100.0 / 100.0 |
| `apikey` | adversarial | 5 | 1 | 0 | 80.0 / 100.0 |

**Detector precision and recall.** We ran each built-in pattern against every document and scored each matched span against the annotation. A match counts as a true positive if the matched text is a secret of the class under test, and as a false positive otherwise; a match of a secret belonging to a different class is additionally counted as a cross-class match and reported with its class, since for a user the practical consequence of a cross-class match is the same as for any other false positive. Recall is computed over the annotated occurrences of the class under test, and an occurrence that is not matched is a false negative. The results are in Table 1 and the underlying JSON is shipped with the paper.

**Round-trip fidelity.** The round-trip experiment registers every annotated secret of a document as a value entry, masks the document, and restores the masked payload through the restoration routine, comparing the result with the original byte sequence. This checks the property the whole design rests on: that the model can be shown a token and the executor can still obtain the value. The experiment also counts the documents in which a secret survived masking in clear text, which would be a direct disclosure. The result is that 62 of 62 documents restore byte-exactly, and no document contains an annotated secret that survived masking in clear text. Masking modified 53 of the 62 documents; the remaining nine contain no annotated secret and pass through unchanged. The harness additionally exercises specific edge cases that the corpus does not cover: an embedded null byte inside the secret and inside the payload round-trips exactly, an empty-string value entry leaves the payload unchanged rather than crashing, a value repeated several times in one payload round-trips exactly, a value that is a prefix of another registered value round-trips exactly, a custom pattern that can match the empty string produces four placeholders in a 47-character output, and the caller's own message

list is not mutated by masking.

**Placeholder stability.** Stability is the property that distinguishes content addressing from enumeration. The harness computes the placeholder for a probe secret under five conditions: registered alone; registered together with two other secrets; registered after those two, in the opposite order; in a second, independent application instance with a different data directory; and reached through a pattern entry rather than a value entry. The placeholder was identical, `[HC:976b88]`, under all five conditions, so the stability property holds in this configuration.

**Coverage gap.** The *other*-class secrets—credentials and identifiers in formats the built-ins do not target—are not covered by the built-in detectors at all. The harness counts them and reports the fraction that was nevertheless masked, which is zero: none of the 11 *other*-class annotations was masked by the built-in patterns. This is a coverage statement, not a defect: the built-ins are three patterns, and the corpus contains, among other things, a cloud API key prefix, a fine-grained personal access token prefix, and a PEM block, none of which the built-in API-key pattern matches.

## 6.3 Collisions

The placeholder space for a prefix of $h$ hexadecimal characters has size $16^h$. For $h = 6$ that is $2^{24}$, and the standard birthday argument gives an expected number of colliding pairs of approximately $n(n-1)/(2 \cdot 16^h)$ and a probability of at least one collision of approximately $1 - \exp(-n(n-1)/(2 \cdot 16^h))$. The number of distinct values at which this probability first exceeds one half is $\sqrt{2 \cdot 16^h \ln 2}$, and solving the integer inequality $n(n-1) \geq 2 \cdot 16^h \ln 2$ gives 4824 for $h = 6$ and 77164 for $h = 8$; widening the prefix by two hexadecimal characters therefore multiplies the tolerable population by roughly sixteen.

The analytic bound is a model of the hash, not a measurement of it. The harness therefore also performs an empirical check: it draws a deterministic pseudo-random sample of one million distinct secrets, computes the SHA-256 prefix of each, and counts the distinct prefixes at several sample sizes; and separately it searches for the first actual collision among six-hexadecimal prefixes, reporting the number of samples drawn until it appears. The first six-hexadecimal collision appeared after 5,012 draws, on the prefix `0a639a`. The empirical collision probability is 0.49 over 200 trials at $n = 5{,}000$ and 0.92 over 100 trials at $n = 10{,}000$, bracketing the analytic half-way point of 4,824; for eight-hexadecimal prefixes it is zero at $n = 10{,}000$, 0.225 over 40 trials at $n = 50{,}000$, and one at $n = 200{,}000$. At one million distinct values the number of distinct prefixes is 970,579 for six hexadecimal characters and 999,890 for eight. Figure 1 plots the analytic curve against the empirical points.

The practical implication is worth stating plainly. A user with a few hundred registered secrets is nowhere near the collision regime for either prefix length. A user with tens of thousands of registered secrets, which is unusual but not absurd for a long-lived registry in an enterprise setting, is in the regime where a six-hexadecimal prefix has a probability of roughly one half of colliding, and where the eight-hexadecimal upgrade moves the half-way point to roughly 77,000 values, a sixteenfold larger population. The registry's automatic upgrade rule detects collisions only among registered values, and only at registration time, so a registry that grows past the threshold does not upgrade itself unless the colliding pair is registered together.

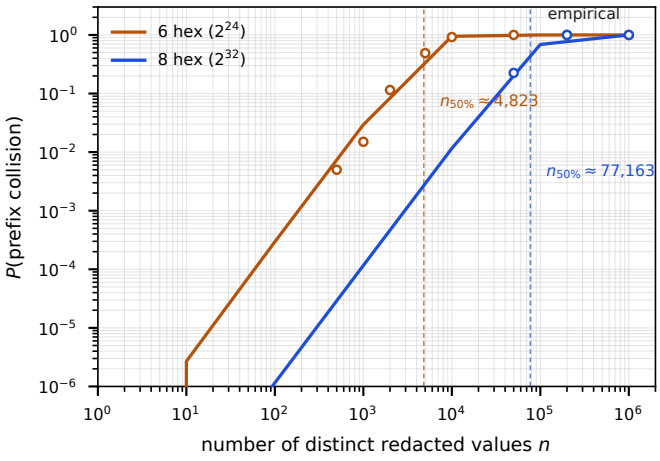



Figure 1: Probability of at least one placeholder collision as a function of the number of distinct redacted values, for a six- and an eight-hexadecimal character prefix, computed from the birthday bound and overlaid with the empirical counts produced by the shipped harness.

## 6.4 Enforcement audit: negative results

The measurement in Section 6.2 exercises the masking and restoration routines directly. It therefore cannot detect the case in which those routines are never called. We close that gap with a static audit of the request pipeline, and with targeted micro-experiments for the two algorithmic hazards identified in Section 4.

**The redaction layer is applied on one path only.** Enumerating the call sites of the masking routine across the whole core crate yields exactly one, inside the non-streaming single-shot request function. The agent's turn loop does not use that function: it dispatches either to a native function-calling routine or to a streaming routine, both of which build their request bodies directly, and neither of which calls the masking routine. The HTTP server's streaming endpoint is in the same position. The consequence is that in the configurations the desktop application actually uses, the transmitted payload contains the plaintext secrets; the mechanism as described in Section 4 is a property of the routine, not of the system. A grep for the call site is shipped as `scripts/check_mask_call_sites.py` so that the claim is re-checkable.

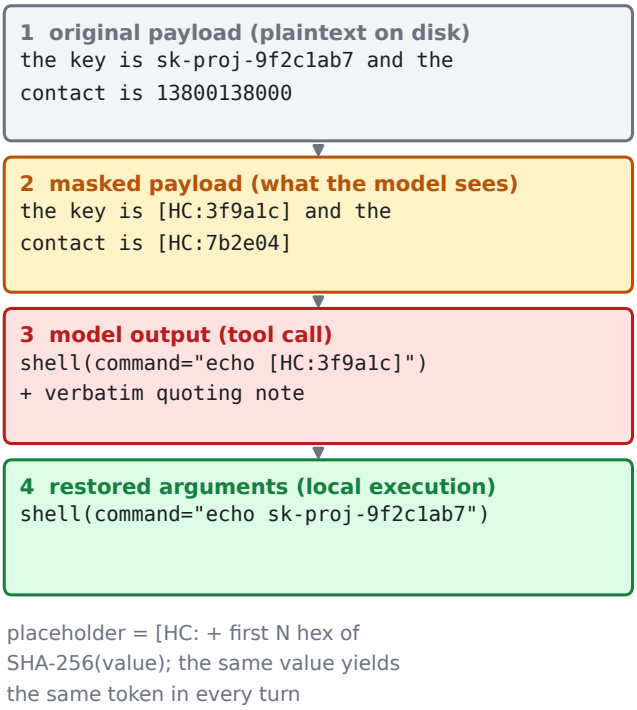



Figure 2: A worked example of one message transformation. The user message contains a key and a mobile number; the transmitted copy contains their content-addressed placeholders and the redaction note; the model echoes the placeholders in its tool arguments; the dispatcher restores the real values before execution.

**Pattern-matched secrets are masked but never restored.** The restoration map is built only from value entries; pattern entries contribute nothing to it. A secret that was masked because it matched a pattern is therefore replaced by a placeholder in the transmitted payload, and that placeholder is *not* in the restoration map, so the restoration routine leaves it in place—as designed for hallucinated tokens, but with the opposite consequence for genuine ones. The tool argument that reaches the executor contains the literal text `[HC:...]`. The harness measures this directly: with only the built-in pattern entries enabled, 43 of the 62 documents are masked, and all 43 restore to a payload that differs from the original—the placeholder is still present, because the restoration map has no entry for it. The practical consequence is a functional one rather than a disclosure one—the tool receives a placeholder instead of a credential—but it is severe for usability, and it means the pattern detectors, which are the only detectors that catch a secret the user did not know to register, are currently one-way.

**Alias restoration is order-dependent.** With two alias entries whose alias strings nest, the restoration outcome depends on the iteration order of the hash map, which is not stable across processes. The harness runs the restoration repeatedly on the same context and counts the distinct outcomes; over 4,000 repetitions it observed two distinct restored strings, `REAL-ALICE Smith` 2,277 times and `REAL-ALICE-SMITH` 1,723 times. The choice is not stable within a single process, let alone across processes.

**Short value entries can corrupt longer placeholders.** A value entry whose text occurs inside another entry's placeholder is substituted literally during restoration. The harness searches for a probe value whose six-hexadecimal digest contains the two-character string `ab`, registers both, and then masks and restores the probe; the probe is `probe-318`, its placeholder is `[HC:ab5272]`, and registering the two-character alias `ab` caused the placeholder to be rewritten as `[HC:REAL-AB-VALUE5272]` in 1,719 of 4,000 repetitions (43.0%).

**Throughput.** The harness measures masking and restoration throughput over the corpus, in bytes per second, for the built-in detector set. Over a 3,005-byte corpus with 59 registered entries it measured 14.22 MiB/s for masking (2,079 iterations) and 11.91 MiB/s for restoration. This is reported only to establish that the mechanism is not a throughput bottleneck at the scale of a single agent session; it is not a latency measurement of a real request, which would require an end-to-end instrumented run that we did not perform. The corpus is small enough that per-call setup is included in these figures, so they are a conservative lower bound on steady-state rate.

## 6.5 Threats to validity

**Internal validity.** The clearest threat is that the harness drives the routines directly rather than through the agent, so it inherits the same blind spot that Section 6.4 documents: it cannot see a path on which the routines are not invoked. The static audit was written specifically to compensate for this, but a static audit of one repository at one commit cannot rule out a further path. The span-recovery procedure also assumes that a recovered span is the longest span consistent with the observed digest prefix; this is exact for the greedy built-in patterns but would be wrong for a pattern with an unusual quantifier, and a custom pattern entry of that kind would be mis-scored.

**External validity.** The corpus is 62 documents written by the author of this paper, not a sample of real user traffic. It was constructed to contain realistic developer payloads and adversarial near-misses, but it cannot represent the long tail of formats found in production. The precision figures are therefore best read as an indication of the *shape* of the error distribution—in particular, that the detectors are conservative about numbers and aggressive about e-mail addresses—and not as an estimate of the accuracy that a user would experience. No user study, no live model was used, and no claim is made about model behaviour.

**Construct validity.** The round-trip experiment measures whether the pair (mask, restore) is the identity on the corpus when every secret is registered as a value entry. That is the

strongest possible statement of round-trip fidelity, and it is not the same property as end-to-end correctness of an agent turn, which additionally requires the model to quote the placeholder verbatim. We did not measure that second property, and we do not claim it.

**Conclusion validity.** The system studied is one implementation. We are careful to separate the design argument, which is general, from the audit findings, which are properties of this commit of this tree.

# 7 Related work

**Prompt injection.** Greshake et al. [Greshake et al., 2023] gave the first systematic study of indirect prompt injection against LLM-integrated applications, distinguishing direct from indirect injection and proposing a taxonomy of mitigations. Evertz et al. [Evertz et al., 2024] revisited the taxonomy with a large randomised study of confidentiality in LLM-integrated applications. On the model side, Wallace et al. [Wallace et al., 2024] trained models on an explicit instruction hierarchy in which privileged instructions outrank content supplied by users or tools, and reported improved resistance to injection. Our paper takes the opposite end of the design space from most of this work: it does not attempt to make the model resistant to injected instructions, but to remove the capability that a successful injection would abuse.

**LLM as judge and agent auditing.** Zheng et al. [Zheng et al., 2023] studied the reliability of large language models as judges of risk assessment in agentic systems, finding that such judges are useful but sensitive to the judging prompt and to the presence of injected instructions. Our audit layer follows this pattern and inherits its known weaknesses; the fail-open design decision discussed in Section 8 is a direct consequence of that literature.

**Agent tool-calling protocols.** The Model Context Protocol [Anthropic, 2024] standardises how an agent exposes and calls tools over a transport, and the JSON-RPC 2.0 specification [JSON-RPC Working Group, 2010] underpins its message format. These are interoperability concerns, but they are where the audience separation of Section 4 has to be expressed: a protocol that carries tool arguments from an untrusted party to a trusted executor must define where redaction and restoration occur, and the Model Context Protocol does not.

**Least privilege and the principle of least authority.** The mechanism is an instance of a long-standing idea in systems security: give the untrusted component a handle rather than the capability. Saltzer and Schroeder [Saltzer and Schroeder, 1975] articulated the principle of least privilege and the related notions of complete mediation and the psychological acceptance of the security mechanism. Bell and LaPadula [Bell and LaPadula, 1973] describe an early formal confidentiality model for a different but structurally similar problem. Our contribution is not the principle but the observation that in the agent setting the handle can be made *content-addressed*, which turns a capability the model must be trusted with (a stable, meaningful identifier for the secret) into a capability it cannot exploit (one whose meaning is unknown to it).

**Event sourcing and append-only audit.** The system's audit log is an append-only sequence of records in a local database, ordered by insertion. Kleppmann [Kleppmann, 2017] gives the modern treatment of event-sourcing architectures and their trade-offs, and the append-only audit log is the closest thing in the system to an event log that a user can replay. We use this framing in the two-audience discussion: the audit trail is written by the trusted side, and the model's view of the world is a projection of it.

**Truncated digests and collision analysis.** Using a truncated cryptographic digest as an identifier is standard practice, and the birthday bound is the standard analysis of its failure probability. van Oorschot and Wiener [van Oorschot and Wiener, 1999] discuss parallel collision search for truncated hashes, which is the relevant body of work if an adversary wishes to engineer a collision against a specific known value rather than wait for a random one. The SHA-256 specification itself [National Institute of Standards and Technology, 2015] is the source of the digest used in Equation 4.1.

**Risk taxonomies for LLM applications.** The OWASP Top Ten for Large Language Model Applications [OWASP Foundation, 2023] places prompt injection, sensitive information disclosure, and improper output handling in the same taxonomy this paper addresses, and is the closest thing to a shared vocabulary for the problem.

# 8 Limitations

We list the limitations of the mechanism as implemented, and of this study, without hedging.

**The mechanism is not applied on the paths the agent uses.** The redaction routine is invoked from one call site, and the agent's turn loop uses neither the function containing that call site nor any function that calls it. Until this is fixed, the protection described in this paper is not active in the shipped configurations. This is the single most important limitation of the work, and it is invisible unless one audits the pipeline rather than the module.

**Pattern-detected secrets cannot be restored.** The restoration map is built from value entries only. A secret caught by a built-in detector is masked on the way out and passed to the executor as a literal placeholder. The design intent appears to be that a pattern-detected secret is redacted-but-not-needed-by-the-executor, which is coherent, but it is not what the module's own documentation suggests, and it means the detectors

that matter most in practice—those for secrets the user did not know to register—cannot participate in the round trip.

**The audit layer is experimental, disabled by default, and fail-open.** The second-model audit is off unless the operator enables it. When enabled, a timeout, a network failure, or an unparseable verdict is treated as *unavailable* and escalates to human approval rather than denying the call. This is a deliberate availability-over-safety trade-off: a fail-closed audit layer would let an attacker who can degrade the audit-model endpoint block all tool use, which is a denial-of-service primitive against the user's own agent. We accept the trade-off but state that it means the audit layer cannot be relied upon as a security control; it is an advisory filter whose failure mode is human involvement, and a user who has enabled it may believe they have a control they do not have.

**The path jail is lexical and has no default root.** Path containment is decided by textual normalisation and a string-prefix test. Symbolic links are not resolved, so a link inside the root can point outside it. More significantly, the desktop configuration ships without a workspace root, and in that state the jail returns candidate paths unchanged. The mechanism must therefore be described as a guard rail that can be enabled, not as a sandbox that is always in force. We explicitly do not describe this system as secure by construction.

**The interpreter execution path is not a sandbox.** Running a script through an interpreter is a plain subprocess execution with a timeout and a cap on captured output. The resource-budgeted script engine is a separate path, with expression-depth, operation-count, and wall-clock limits, and that is where the meaningful containment lives.

**The audit log is a plain append-only table.** It records who did what, in insertion order, and it is not tamper-evident: an attacker with write access to the local database can rewrite history. Framing it as an event log [Kleppmann, 2017] describes its structure, not its integrity properties.

**Secret detection is syntactic and shallow.** Three regular expressions cover a narrow slice of credential formats. Our own corpus contains a cloud API key prefix, a fine-grained personal access token prefix, a PEM private key block, and an eighteen-digit national identifier, none of which the built-ins match. A user who relies on the built-in detectors alone will leak secrets in those formats.

**The measurement is a synthetic corpus written by one author.** The 62 documents are not sampled from real traffic; see Section 6.5. The precision and recall figures characterise this corpus and should not be generalised.

**The study measures no model behaviour.** We did not run a language model against the mechanism. We do not claim that models quote placeholders verbatim, that the redaction note improves compliance, or that a masked context is harder to inject into. The redaction note is a prompt, and prompts are not verified here.

**The project's own tests were machine-generated.** The repository's testing document states that the test suites were produced by a large language model. We therefore treat the suite as a regression harness rather than as an independent validation, and we rely on our own harness—which links the implementation rather than describing it—for the measurements in Section 6.2.

**Scope of the artifact.** The subject of this paper is one module of one system. We do not claim that the design generalises to every agent runtime, and we do not claim that the measurements transfer to other implementations of similar mechanisms.

# 9 Conclusion

We have described a mechanism for handling secrets in tool-using agents that keeps the secret out of the model's context while leaving the agent able to act on it. The design point that makes it work is that the placeholder is content addressed: the token is a function of the secret's content, not of its position in a registry or a counter, so it is stable across turns, independent of registration order, and recoverable locally at the point of use. We made the mechanism precise against an executable artifact, measured its detector precision and recall, round-trip fidelity, placeholder stability and collision behaviour on a shipped corpus and a shipped harness, and audited its enforcement.

The audit is the part we would emphasise. Three of the mechanism's properties—that it protects the provider-facing payload, that a detected secret can be restored for the executor, and that the placeholder is unambiguous—hold for the routines as written but do not hold for the system as configured, because the routines are not invoked on the paths the agent takes, and because the restoration map omits pattern-detected secrets. A mechanism described at the level of a module is not automatically a mechanism of the system, and measuring it at the level of the system is what distinguishes the two. We hope that the harness, which links the shipping implementation rather than reimplementing it, is usable by others who face the same question about their own agent runtimes.

# References

Anthropic. Model context protocol, 2024. Specification; https://modelcontextprotocol.io.

D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report ESD-TR-73-278, Volume I, MITRE Corporation, 1973.

J. Evertz, M. Chlosta, L. Schönherr, T. Eisenhofer, and T. Risse. Whispers in the machine: Confidentiality in LLM-integrated applications, 2024. arXiv:2402.06922.

K. Greshake, S. Abdelnabi, S. Mishra, 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 '23)*, 2023. arXiv:2302.12173.

JSON-RPC Working Group. JSON-RPC 2.0 specification, 2010. https://www.jsonrpc.org/specification.

M. Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

National Institute of Standards and Technology. Secure hash standard (SHS), 2015. FIPS PUB 180-4, https://doi.org/10.6028/NIST.FIPS.180-4.

OWASP Foundation. OWASP top 10 for large language model applications, v1.0, 2023. https://owasp.org/www-project-top-10-for-large-language-model-applications/.

J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9): 1278–1308, 1975.

The Rust Project Developers. The regex crate: a rust library for parsing, compiling, and executing regular expressions, 2024. https://docs.rs/regex.

P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12 (1):1–28, 1999.

E. Wallace, K. Xiao, R. Leike, et al. The instruction hierarchy: Training LLMs to prioritize privileged instructions, 2024. arXiv:2404.13208.

S. Yao, J. Zhao, D. Yu, et al. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

L. Zheng, W.-L. Chiang, Y. Sheng, et al. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023), Datasets and Benchmarks Track*, 2023.