Sign Everything. No, Everything.

A Signed, Layer-Descending Stack for Agent Computation:
Identity, Authentication, and Zero-Knowledge Privacy across the Stack

Lewis Tham
Unbrowse AI
lewis@unbrowse.ai

Cayden Chik
Unbrowse AI
cayden@unbrowse.ai

May 2026

Abstract

An agent that acts on the web stops being a polite HTTP client the instant it has to click something, run a shell command, open a real browser, or put bytes on a socket a bot-detector is inspecting. The load-bearing work does not confine itself to the layer where JSON comes out, and neither can the security model. This paper makes one claim and tries not to oversell it: the discipline that makes a *single* signed request trustworthy — sign it, seal it, cache it, prove who you are without showing your hand — is the same discipline, unchanged, all the way down the stack (screen → browser → CLI → OS → kernel → packet). Every layer is signed by one Ed25519 wallet key; credentials are bound to that identity by zero-knowledge proof and revealed only under signature; every intermediate result is content-addressed and cached, with the signed commitment appended to a hash-chained ledger (value off-chain, root on-chain); and no accepted transition crosses a layer boundary without a signed commitment. We cite an established primitive for each layer rather than inventing one; the contribution is their *composition* into a single signed identity that owns the whole descent. Route discovery — turning a per-call browser tax into a shared, reusable route graph — is an orthogonal result established separately [1]; this paper is the security, authentication, and privacy of the stack that *executes* those routes. Compensation sits above the substrate and stays deliberately narrow: discovery and internal-API routing are free, and paid execution settles over x402.

1 Introduction: one layer is not the stack

The interface an agent uses to read and act on the web resolves an intent to a ranked endpoint shortlist and executes the chosen call against the live site. That much can be made cheap and reusable, and is [1]. This paper is about the security model for what happens *when the chosen action cannot stay at the HTTP layer* — which, for any agent that does real work, is often.

The common assumption is that the load-bearing part of agent work lives at the HTTP layer, where things are clean. They are not clean. A login form is a screen-layer fact. A session cookie is a browser-layer fact. A keychain entry is an OS-layer fact. A TLS fingerprint is a packet-layer fact a defender is keying on whether or not the agent was thinking about it [32, 33]. An agent that is genuinely accountable for its own actions has to be the same coherent, signed entity across all of those layers — not just the one that happens to return `application/json`. A signature that secures the JSON call but not the keychain read, the browser session, or the packet on the wire secures nothing the adversary cannot route around.

So the thesis fits on one line: *the layering is the point, and one signing discipline holds at every layer.* The end-to-end argument in system design [5] already established where each function

belongs; we take it at its word and run it down a signed agent stack. The result is a single Ed25519 identity that owns every layer of the descent, credentials bound to it without disclosure, and a content-addressed, sealed, ledgered substrate underneath — each layer pinned to an existing cryptographic primitive rather than a new one.

2 Related work

Our contribution is not a new agent or a new chain; it is the claim that one signed discipline holds across every layer an agent touches, and that the route economy is its natural settlement. We situate that against four lines of prior work.

Web agents and the browser-first assumption. A large body of work has agents drive real browsers: ReAct interleaves reasoning and action [7], and benchmarks such as Mind2Web [8], WebArena [9], and the GPT-4V grounding study SeeAct [10] measure agents on live or simulated sites. This work treats the rendered page as the interface. We take the opposite stance: the browser is a fallback, and the durable interface is the first-party API the page already calls. Our stack keeps the browser as the bottom of a descent, not the default.

Tool and API invocation. Toolformer teaches a model *when* to call an API [11]; Gorilla teaches it to emit *correct* calls across thousands of APIs [12]; the Model Context Protocol standardises how a model connects to tools and data [13]. These solve single-agent tool use. They do not address what happens when many agents discover the same interface repeatedly, who maintains that knowledge as sites drift, or how the cost of discovery is shared — the maintenance-and-economics problem we leave to future work.

Identity, transparency, and capabilities. Our trust layer reuses established cryptographic systems rather than inventing them: EdDSA signatures [15], zero-knowledge credential binding [18, 17], Certificate-Transparency-style append-only logs [22], the object-capability model [30], and the Dolev–Yao adversary [14]. The novelty is not the primitives but their *composition* into one identity that signs every layer and one ledger that records every claim, with ERC-8004 [16] as the cross-agent identity surface.

Agent payments and incentives. x402 revives HTTP 402 as a real micropayment rail [4]. Prior agent-payment work largely stops at "the agent can pay." We extend it to "the payment routes to everyone who created the value" — indexer, domain owner, platform — as a fair three-way settlement over x402, while discovery and internal-API routing stay free. To our knowledge the combination — signed multi-layer descent, a content-addressed-plus- ledgered substrate, and a contribution-weighted three-party settlement over x402 — is not present in prior work as a single system.

3 The stack is one node, instantiated at every layer

We model the stack as a layered tree in the OSI tradition [6], but oriented around *who is acting* rather than *which bytes move*:

$$\text{screen-clicks} \rightarrow \text{browser} \rightarrow \text{CLI} \rightarrow \text{OS} \rightarrow \text{kernel} \rightarrow \text{packet}.$$

These are not six layers held together by prose. They share one structure: each layer is described by the same four fields, so we define the structure once and instantiate it per layer. A layer node has exactly four fields:

- **subject** — the action this node performs at its altitude.
- **verb** — one of *observe*, *act*, *verify*: the kind of step.
- **witness** — a check returning settled / not (two independent corroborations, or the step is not accepted).
- **parent** — the hash of the node one layer up: the edge that links a layer to the one above it.

The descent is then a uniform recursion over one schema: each layer is a node, and its child is the same node one altitude lower. Using a single schema is the whole point — it is what lets *one* signature discipline and *one* cache serve the entire stack instead of six bespoke integrations.

The end-to-end argument [5] supplies the rule for which node acts: a function lives at the highest layer that can do it completely and correctly, and lower layers are fallback, not first resort. A cached signed plan beats a CLI call beats a browser action beats moving a mouse across pixels like it is 2004. But when the higher layer simply cannot act — no API, hostile anti-bot, a page that is 100% JavaScript and 0% mercy — the agent descends to the child node, and at the bottom it emits packets whose TLS/HTTP fingerprint is indistinguishable from a real browser via curl-impersonate [31], matching exactly the JA3/JA4 signature the other side is keying on [32, 33]. The browser, CLI, and HTTP layers and the fingerprint-faithful fetch run in the product, and the uniform *signed descent through every layer* (screen → browser → CLI → OS → kernel → packet) ships in production: `src/values/signed-descent.ts` signs one wallet root once and threads a hash-chained, per-layer signature down the whole stack, with tests that any tampered or reordered layer fails to verify. Ownership is vertical and rides the **parent** edge: `src/values/wallet-hierarchy.ts` derives each layer's wallet from its parent's (HKDF) and the parent signs the child's key, and `src/values/layer-wallet-descent.ts` has each layer sign with its own parent-owned wallet — so the root wallet owns the screen wallet owns the browser wallet, all the way down to the packet wallet. What remains referenced rather than shipped is emitting *real* signed OS/kernel/packet operations across platforms; the signed descent record, the wallet hierarchy, and the fingerprint-faithful HTTP emission are in the product today.

The reason any of this composes is that the structure at every layer is identical — observe, act, verify, under one signature — so the system's mechanisms are not a grab-bag of separate subsystems but the recurring concerns of that one structure, each met by an existing primitive (Table 1). We did not invent these primitives; we mapped each layer's concern onto the appropriate one.

3.1 Traversing the tree: descent is a cheapest-first search

A tree of nodes is inert until something walks it. The stack's operations — resolve an intent, execute a route, fall back to a browser — are not three subsystems but *one graph traversal of the node tree, weighted by compute cost.* The end-to-end rule above is exactly Dijkstra's algorithm with edge weight = compute cost: visit nodes in increasing cost and stop at the first whose witness settles.

```
walk(intent):
  for node in tree.nodes_by_increasing_cost(intent):   # Dijkstra over compute cost
    if node.cached and node.witness.settled:            # RESOLVE -- cache hit, 0 descent
      return node.replay()                              # EXECUTE -- no deeper layer touched
```

concern	realised as	primitive (cited)
descent	spawn a child node one layer down	curl-impersonate / JA3, JA4 [31, 32, 33]
the tree	one node whose children are nodes	end-to-end argument / OSI [5, 6]
identity	the root node's key, inherited down	ERC-8004 trustless agent [16]
witness	`node.witness` — the settle check	zkLogin / Pedersen commit [18, 20]
cache	a node memoising its child	Merkle content-addressing [21]
↪ KV parallel	same node, token altitude	attention KV cache / PagedAttention [26, 27]
seal	the gate on `node.verb`	AEAD / object-capabilities [29, 30]
ledger	the `node.parent` edge, append-only	Certificate Transparency / hash chain [22, 23]
loop	a node re-firing until its witness settles	OODA / proof-of-history clock [35, 24]
(edge) economics	payment on the parent→child call	x402 split settlement [4]

Table 1: The system's mechanisms are the recurring concerns of one layer structure, each aligned to an existing, cited primitive. One element is *not* a node and we name it rather than hide it: economics (x402) is a weight on the **parent** edge, not a node — so the precise structure is one node type plus one typed edge type, a weighted tree (this is the whole content of §9). A runnable reference implementation of the cache + ledger core ships alongside this paper (`reference/`, with tests that execute each claim).

```
    if node.can_act_completely(intent):                    # highest capable layer
      r = node.act()
      if r.witness.settled:
        cache.put(node, r); ledger.append(node)            # CAPTURE the win for next time
        return r
  return walk_child(intent)                                # FALLBACK -- the browser is the deepes
```

BFS, DFS, and Dijkstra are not three different walks here but *one walk with three priority functions* over the same tree: BFS = uniform cost, DFS = descend immediately on miss, Dijkstra = cost-weighted (the production default, cheapest-capable-first). The resolve→execute→fallback ladder *is* that Dijkstra instance — three short-circuit tiers, cheapest first: cache replay (0 descent) → one capable-layer act → descend to the child. Each settled node pushes future identical intents to a cheaper tier, so the deep layers (the browser) are the cache-miss oracle, visited as rarely as possible.

3.2 The traversal record is the ledger

Every node the walk visits, and the verdict its witness returned, is appended to one append-only log keyed by the node's content hash. That single object wears three names and is the same primitive each time: as the *route ledger* it records `{signer, value-hash, timestamp, prev-hash}` per settled node (§8); as the *resolution ledger* (`src/values/resolution-ledger.ts`) it is the content-addressed memo of the traversal, so a hit short-circuits the whole descent (the cache of §6); and as a *session record* it is the human-readable tail of the same log, where a node recorded *refuted* is a pruned branch the walk never re-enters — the negative result is a cache entry, not a deleted fact. Memoisation (do not recompute a settled node), pruning (do not re-walk a refuted node), and provenance (the hash-chain proves the order) are one append-only ledger seen three ways. §8 is that ledger as the **parent** field.

3.3 Composition: a walked traversal is itself a node

Descent makes a node's *child*; the same schema run upward makes a node's *parent*. When one intent cannot be settled by a single route — the route it resolves to needs a binding only another route

yields — the walk visits several nodes in dependency order, threading each one's output into the next. That ordered sub-walk is not a transient. It is the traversal record of the previous subsection, and by the same content-addressing it is itself a node: a *composite* is the parent whose children are the routes it composed, keyed by the content that makes it the same DAG — its domain, its target, its ordered constituent nodes, and the binding edges between them — and replayed as one unit. The first agent to walk the chain records the composite; a later agent, including one that never walked it, looks it up by that address and replays the recorded order rather than re-deriving it — exactly the cache-hit short-circuit of the walk (`node.cached` ⇒ `node.replay()`) one altitude up. So the recursion closes in both directions: descent spawns the child when the parent cannot act, and composition records the parent when several children together settle an intent no child settles alone — and the traversal record is the ledger at every altitude. This ships: `src/orchestrator/` content-addresses a walked multi-step traversal as a composite and, on a later resolve, replays the recorded order before re-walking, with a constituent-staleness guard — a removed or disabled child invalidates the composite back to a full re-walk, so a stale parent never replays a broken child.

4 Identity: the root node's key, inherited by every child

Identity is not a layer of its own — it is the one **subject** field every node shares, and the property that the whole tree is keyed by one keypair. The root of trust is a single Ed25519 keypair [15] — and conveniently, the agent already has one, because a Solana account literally *is* a base58 Ed25519 public key. The wallet is the identity. Every action — a click, a syscall, a packet — is an admissible `node.verb` only if it chains to a signature under that one root. No signature, no action; the stack has no anonymous side door. The Unbrowse signer already produces wallet signatures for resolve/execute admission. The key descends the **parent** edge, never re-keying: each child node's key is HKDF-derived from its parent's and the parent signs it (§3), so one identity threads the whole tree.

For trust *between* agents we adopt the ERC-8004 "Trustless Agents" model [16], which defines three on-chain registries: **Identity** (portable agent IDs), **Reputation** (signed feedback), and **Validation** (independent re-execution / ZK / TEE checks). An Unbrowse route maintainer is precisely one of these agents: a portable identity that can carry reputation and be checked by someone who does not trust it yet. The three ERC-8004 record types map cleanly onto the wallet identity already in play — a portable Identity that is the wallet pubkey, signed Reputation feedback, and an independent Validation re-execution record, each signed by a real wallet. Binding to the *deployed* on-chain registries remains integration work, and we will not pretend otherwise.

Key mobility and the public boundary. One root keypair owning every layer raises the obvious question: what may be exposed, and where? The answer is an asymmetry. The *private* key is mobile *inward* — it descends or ascends the stack to surface an identity or value at whatever altitude needs it, a click at the screen layer or a signature on a packet, always the same root. Across the *public* boundary the rule inverts: **only value copies cross the public boundary** — content-addressed copies of the value, verifiable against the *public* key and nothing more. The private key never leaves; what the world receives is a copy it can check, but cannot forge and cannot reverse into the secret. Surfacing a value at a layer is a private-side act under the one key; publishing it emits a fresh, content-addressed copy carrying the public key and a signature, never any private material. This is the runnable property in `paper/reference/layers/key_mobility.py`: one key surfaces a value at every layer, a foreign wallet cannot forge a public copy, a tampered copy fails its content hash, and the private key is provably absent from everything that crosses. The production

port ships in `src/values/wallet-seal.ts`: `publishValue` emits a **wallet-signed value copy** verifiable under the public key, the outward complement of the sealed-unless-revealed cache, with the private key loaded, used once, and zeroed before the copy ever leaves.

5 Witness without disclosure: ZK credential binding

This section is `node.witness` made non-disclosing: the witness must settle "yes, bound" without revealing what it witnessed. Signing is the easy half. The hard half — the actual research — is proving a credential belongs to the identity *without revealing the credential*. Your password, your session cookie, your 1Password entry, the fact that you clicked "approve" for this domain: all of it should be provably bound to the wallet and leak *nothing* identifiable at any layer unless you choose to open the proof under signature.

This is the classical anonymous-credential problem [17], and it has grown up. zkLogin [18] binds an existing OAuth/OpenID identity to a chain address through a Groth16 proof, hiding the link even from the identity provider; Semaphore [19] proves anonymous group membership with zk-SNARKs; Pedersen commitments [20] give the "commit now, open later, can't lie about it" primitive for a single secret. Together they make the binding a two-witness corroboration that betrays nothing: the chain sees *that* a credential is bound, never *what* it is. This is the central contribution, and the primitive is implemented in the product: `src/values/zk-binding.ts` implements the same non-interactive Schnorr proof (Fiat–Shamir over a 2048-bit MODP group) that proves a credential is *bound to the wallet without revealing* it — the wallet signs $y = g^x$ where x is derived from the credential, and a holder proves knowledge of x while the verifier learns only "yes, bound." It is wired at the capture boundary by `src/capture/zk-bound-hole.ts`: each redacted secret *hole* carries the binding, so the backend confirms a credential is bound to the wallet without ever seeing the secret, and the holder proves knowledge only at fill time. Tests confirm the credential never appears in the binding or the proof, that a wrong credential or a foreign wallet cannot forge one, and that an unbound hole fails closed. The remaining integration is adopting this binding across every entry of the live credential vault end-to-end; the primitive and its hole-level binding are shipped and tested, not promised.

6 Cache: a node memoising its child (content-addressed, sealed unless revealed)

The cache is simply a node that memoises its child: address the result by the hash of its own bytes, and a later walk that reaches the same node replays the stored value instead of descending again (the traversal short-circuit of §3.1). Every layer's settled result — a resolved endpoint, a rendered page, a signed plan — is memoised under a content-addressed key, in the Merkle tradition [21] and exactly as realised by content-addressed stores like IPFS [25] and Git's object model [28]. Same structure re-walked, same key, same answer, deterministically; the cache is re-derived to *verify*, never trusted because it looked right last time. Unbrowse's centralised cache servers can hold these entries and serve them across agents, while a sensitive entry is kept sealed until the holder opens it — a content-addressed cache that is public by default and private by proof. Endpoint and route caching run in the product, and the sealed-unless-revealed commitment layer now ships in production: `src/values/wallet-seal.ts` addresses each value by the sha256 of its *plaintext* (so the same content resolves to the same key on any host) yet stores AES-256-GCM ciphertext under a key bound to the wallet, and tests confirm the at-rest bytes are unreadable, that only the binding wallet can reveal, and that a tampered ciphertext refuses to open.

6.1 The route graph as a commitment-only structure

The content-addressed cache above keys each value by the sha256 of its plaintext, so the same content resolves to the same key on any host. That is exactly right for a *private* cache — but the route graph is *shared* across agents, and a bare hash on a shared structure leaks: a low-entropy value (a boolean, a locale, a short id) is recoverable by hashing a dictionary against the visible commitment, and equal values expose linkable equality across hosts. So a captured credential or storage value — a session token, an anti-bot `localStorage` entry — is promoted to a first-class node in the requires/yields graph under a keyed commitment: a MAC under a wallet-derived key, not a bare hash. An observer of the shared graph who lacks the wallet cannot brute-force a low-entropy value, and the same value under two different wallets yields two different commitments, so nothing links across installs. The value itself is sealed off-graph under the wallet and revealed only locally, per step, at the moment of replay — the graph carries the dependency *shape* (which operation yields a token, which operation requires it) while the secret is held by no one but the holder. The keyed commitment and its graph binding are implemented in `src/values/storage-hole-bindings.ts` over the off-graph store `src/values/sealed-blob-store.ts`, and revealed at the replay boundary in `src/execution/index.ts`; tests confirm the secret appears on neither the graph node nor the bytes at rest, that a wrong key cannot reveal, and that a legacy plaintext token is never mistaken for a commitment.

The cache and the ledger meet in one production primitive worth naming, because it is how a slow truth-resolution is paid for once and never again. `src/values/resolution-ledger.ts` treats a signature as the KV key to a *pointer* that resolves on truth resolution. An intent is content-addressed to a pointer; the first resolution runs the expensive work, stores its result content-addressed (`sha256:`⟨hex⟩, exactly the `putBlob`/`resolvePointer` shape), and appends one row to an append-only, hash-chained *ledger of resolutions*. Every later call resolves the pointer instead of recomputing — and because the value is addressed by the hash of its own bytes, an evicted entry rebuilds the *identical* layer, exactly like a Docker layer cache: a miss rebuilds it, a hit replays it. A settled value is a *promise* in the two senses computer science already gives the word: a *future* [37] — a placeholder computed once and thereafter only read, never recomputed — and a promise in Burgess's *Promise Theory* [36], where an autonomous agent can promise only its *own* behaviour and the order of the whole is nothing but the body of promises voluntarily kept. The resolution ledger is exactly that body: each layer, as an autonomous agent, promises the single value it derived, addressed by the hash of that value's own bytes — so it is a promise the agent can always keep (derived once, then *true forever*), and the append-only, hash-chained ledger is the public, independently corroborable record of those promises. Because it is addressed by the hash of its own bytes, the same value resolves to the same pointer in every process, so a memo keyed on that pointer hits forever and never recomputes — asking "does it need to re-resolve each time?" answers itself: no, not if it is true forever. The only thing that busts the cache is the *value* changing, and the sharpest case is when *time is part of the value*: then the pointer changes every moment and the derivation re-resolves each time — unless the promise included time itself, a timeless claim is computed once and held. Invalidation is this same discipline run the other way: when an upstream value changes, its bytes change, its content hash changes, the pointer keyed on the old hash no longer resolves, and the layer re-resolves on next access (`src/values/resolution-ledger.ts`). Nothing is invalidated by a clock or a manual flush; correctness falls out of the addressing, because a changed input is *by construction* a different key — the same reason changing a base layer in a Dockerfile invalidates the layers built on it. Crucially, dependent recompute needs *no* dependency-graph walk: a value keyed on another value's pointer is automatically a different key the instant that pointer changes, so a changed input invalidates everything addressed through

it without a cascade to engineer. The fallback that feeds it walks the descent ladder highest-capable-first (`src/values/kv-fallback-pipe.ts`), content-addressing each layer's result, so a hit short-circuits the whole descent and a changed input re-resolves only at the layer it touched. The recompute boundary is proven runnably (`reference/ledger/recompute.py`): a timeless value derives once over a hundred reads, a time-keyed value recomputes each moment, and a dependent value re-derives automatically when its upstream pointer flips. Tests confirm the warm path never recomputes, the ledger is tamper-evident, and eviction reproduces the same pointer.

The parallel worth naming is to the *KV cache* inside the transformer doing the reasoning. Attention [26] computes a key and value for every token; the KV cache memoises those states so decoding never recomputes them, and modern serving engines page that cache like OS virtual memory and *share identical prefixes across requests* by content (hash) key [27]. That is the same discipline, one altitude up: same prefix, same cached blocks, reused not recomputed — except the structural key is now a stack subtree rather than a token prefix, and the entry is ZK-sealed rather than served in the clear. The model caches computation it has already done; the stack caches *actions* it has already signed. Same shape, different altitude — a KV cache all the way down, where each layer's entry stays sealed until a signature opens it.

7 Seal: the gate on `node.verb` (nothing ships unsigned)

The seal is not a layer either — it is the gate every node's `verb` must pass before it emits, and authority descends the **parent** edge attenuated, never widened. Before any layer emits, it self-verifies through the root. At the wire that is authenticated encryption with associated data (AEAD) [29]: the receiver checks the tag before it trusts a single byte, and tampering fails closed rather than failing quietly into your logs. The same discipline generalises upward as object-capability attenuation [30] — authority travels only by reference and can be narrowed, never picked up from the ambient air. The rule is boring on purpose: no unsigned action crosses the gate, at any layer, ever.

The same object-capability rule answers the access-control question identity raises: who may *read* what. Because the wallet is the identity (§4) and every sealed value is bound to it, read access to a sealed value is not an ambient permission but a capability the holder *grants* — a scoped, revocable reference, narrowed to a specific value or value class, that the grantee can exercise but cannot widen or re-delegate beyond its bounds [30]. The granter's signature is the only authority that mints or revokes such a grant: there is no account or admin row beneath the key that could override it, so the asymmetry of §4 holds for delegated reads too — the holder decides what crosses, and to whom, and can withdraw it. Authentication (who the key is) and authorisation (who may see what) are thus the same key discipline seen from two sides, not two subsystems. This holds as a *uniform* stack property: `reference/layers/gate.py` is a runnable gate bound to one wallet root that admits an action only if its signature verifies over the action's canonical bytes, and a test confirms that an unsigned, foreign-signed, or tampered action is rejected and counted — zero unsigned actions ever cross. AEAD itself is, obviously, ubiquitous and shipped at the transport layer; we are not claiming to have invented encryption.

8 Ledger: the `node.parent` edge made append-only

Stated as a node field, the ledger is what happens when `node.parent` is required to be append-only and ordered (§3.2): the `prev-hash` *is* the parent edge, realised as a chain rather than a free pointer. Signing is only half the story; the obvious next question — and the one most "signed" systems wave away — is *where does the signature go*. A cache and a ledger are not the same

object, and conflating them is the bug. A cache is content-addressed: you fetch a value by the hash of its content and order does not matter, so the same hash resolves on any host [21]. A ledger is the orthogonal half: append-only, *ordered*, and hash-chained, so the history and the order are themselves tamper-evident. A signed entry — a signed KV result, a route attestation, a signed quality claim — needs both: the cache holds the value, the ledger holds the signed commitment to it.

The load-bearing design choice is *value off-chain, root on-chain*. You never put the payload on a chain — that would be paying gas to store megabytes and reading them slowly. You store the small, signed commitment: an entry of `{signer, value-hash, timestamp, prev-hash}`, where the value itself lives in the content-addressed cache and only its hash appears in the log. Each entry chains to the prior, so reordering or editing any past row breaks every root after it [23]. Independent auditors — not a trusted operator — verify the log is append-only and consistent, exactly the model Certificate Transparency formalises with its Merkle log, Signed Tree Head, and inclusion proofs [22]. That is what makes the ledger trustless rather than "trust our server."

This also answers how the system decentralises without a rewrite. Per-entry on-chain writes do not scale, so you batch: thousands of entries collapse to one Merkle root, and only the *root* is checkpointed on-chain (the Certificate-Transparency / rollup pattern), ordered by a decentralised clock such as Solana's proof of history [24]. The maturity ladder is one shape at every rung: a local append-only log today, a signed-root server next, on-chain Merkle-root checkpoints as the decentralised end-state — you swap the host, the signatures and the hash-chain never change. The local append-only ledger runs in the product (the route graph already keeps signed JSONL records). The Merkle-root checkpoint and independent-auditor layer ship as runnable reference code: `reference/ledger/checkpoint.py` batches many signed entries into one root and emits a per-entry inclusion proof any auditor verifies against that root, with a test that an entry outside the batch cannot prove inclusion and that changing any entry changes the root. Publishing the root on-chain is the deployment step; the commitment it would publish is exactly this root.

8.1 Crystallised intelligence: the resolved graph, keyed by identity, surfaced only under the key

The work the substrate does is itself the thing worth persisting. A resolution — the intelligence of "this intent resolves to that ranked endpoint, and the call succeeded" — is not a transient log line; it is a signed node, and committing it to the ledger *crystallises* it. Every resolution the runtime settles is written as a wallet-signed, content-addressed row on the append-only on-chain ledger (`src/values/iq-ledger.ts`): the row stores the content hash of the resolution and the wallet signature that sealed it, written to a Solana table via `writeRow`, so the transaction signature itself proves that one specific identity wrote that exact row. The substrate that produced the resolution is the same one compiled into the stateless binary and bound across the CLI, the server, and the site (`src/values/contract-native.ts`, `src/values/contract-chain.ts`); what it learns by resolving, it crystallises by signing. This is the substrate deploying its own learning into durable public state — the resolved graph becomes an on-chain record rather than evaporating with the process.

What "settles" means here is precise, and it is the gate on crystallisation: a resolution crystallises *only* once its `node.witness` has settled (§3) — and settling is a *fixed-point coherence* check, not a bare boolean. The substrate runs a joint-embedding predictive witness over the candidate: it iterates the learned operator from two independent seeds and admits the result only if both converge to the same direction *and* that direction is stable under one more application. A configuration that does not reach this fixed point is held, not written — so the on-chain record accretes coherence-

settled intelligence rather than every transient guess. This is the same two-independent-witnesses discipline the rest of the stack uses for admission (§7) and for a slash quorum (§9), here applied to the question *"is this resolution coherent enough to persist forever?"*; the energy selector it gates is the subject of the companion route-ranking work [2].

Three properties fall out of keying that record by the one root identity (§4), and they are exactly the three a maintainer of a shared, compensable graph needs.

Identifiable. Because each row carries the wallet signature of its author, a contribution is attributable to a portable on-chain identity — the wallet public key *is* the contributor id, with no separate account system layered on top. This is the wallet-bound contribution ledger that already keys credit in the product (`src/cli-v7/eval/stats.ts`): who resolved which route is a signed fact on the chain, not a row in a database the operator could quietly rewrite. That attributability is the precondition for paying the party who created a route's value, the economics deferred to §9.

Persistent. The on-chain table is append-only and hash-chained, so the full history of resolutions survives independent of any single server; the local ledger is a recency-weighted hot window over it, never the system of record. Reordering or editing a past row breaks every root after it, so persistence here is tamper-evident durability — a history independent auditors can check — not merely "we kept a copy."

Surfaced only under key or signature. Crystallising a resolution on a *shared* chain must not leak what the resolution touched. The rule is the value-off-chain / root-on-chain discipline of this section applied to every sealed value: the chain carries only the small signed commitment, while the value itself stays sealed off-graph under the wallet (`src/values/wallet-seal.ts`), and a captured credential or auth value is sealed to the holder's wallet (`src/values/auth-vault.ts`). An observer of the public chain reads *that* a row exists and verifies it against the public key; recovering the value behind it requires the holder's private key, or a signature-gated reveal under it, and a sensitive entry stays sealed until that reveal. The ZK binding of §5 is the strongest form of the same rule: the chain confirms a credential is bound to the identity while learning nothing about the credential unless its holder opens the proof under signature. So the crystallised intelligence is *public as a verifiable, attributable record and private as a payload* — legible to everyone as "this identity settled this, here is the proof," yet readable as content by no one but the key. The honesty boundary of §13 holds unchanged: the on-chain table write is in the product, while batching resolutions under a single Merkle-root checkpoint published on-chain (`reference/ledger/checkpoint.py`) is the decentralisation step that scales it — reported as the frontier it is, not as a finished feature.

9 From security to economics: where this paper stops

A signed, sealed, ledgered route graph is a security substrate; what it is *worth* — who maintains the routes, who pays, who earns, and how a freshness claim is made costly to fake — is a separate concern with its own paper [2]. We draw the line here deliberately. This paper is the security, authentication, and privacy of the stack: one key signs every layer, credentials are bound by zero-knowledge proof and revealed only under signature, every result is content-addressed and sealed, and nothing ships unsigned. Fair compensation that sits on top of that substrate is deliberately simple: discovery and internal-API routing are free, and paid execution settles fairly over x402 across the parties who created the value.

Keeping security and compensation as separate concerns is itself the discipline. A security model must stand on its own, without leaning on a token to be true: the signatures, the ZK binding, and the sealed cache are correct or not on cryptographic grounds alone, independent of any economics layered above them. Conversely, the economics is only worth stating once the substrate it secures is trustworthy. So we settle the substrate here; the compensation rule above is the whole of what sits on top of it — free discovery and routing, paid execution settled fairly over x402.

10 The control loop

The whole stack runs as an OODA loop [35]: observe the layer's state, orient it against the model (this is where the learning actually lives, despite "decide" getting all the press), decide the next action at the right layer, act under signature — then repeat with feedback, dropping a layer or replanning on failure, and stopping when the goal is reached. It is the same observe–orient–decide–act cycle run uniformly at every layer of the stack. If that sounds unglamorous, good: the glamorous control loops are the ones that do not converge.

11 Evaluation: the security substrate

This paper's claims are about a security discipline, so we evaluate the security substrate it introduces — not the end-to-end latency win of route reuse, which is a discovery result established separately. We are precise about what the measurements here do and do not establish.

The substrate is correct by executable test. The local cache+ledger reference implementation that ships with this paper is unit-verified: content-addressing, value-off-chain/root-on-chain separation, hash-chain tamper-evidence, ed25519 signatures, and deterministic Merkle-root inclusion all pass as executable tests, so the substrate's correctness claims are runnable, not asserted.

Sealed-cache reuse, measured. We additionally ship a runnable micro-benchmark (`reference/bench/bench_reuse.py`) that isolates the one property the sealed cache rests on — content-addressed reuse versus re-derivation — on the actual cache implementation. Over many trials, re-deriving a representative ∼64 KB extracted payload costs milliseconds while a content-addressed cache hit (read + hash re-verify) costs tens of microseconds: a large mean speedup, wall-clock, that lands in the **≈50–90**× range hardware-dependent, with a gate that fails (and is the part actually pinned) if reuse is not materially faster ($> 2\times$). This is a CPU-only recompute-versus-reuse demonstration — it isolates the cache primitive, not the network or the browser — and it establishes exactly one thing: a sealed, content-addressed commitment is asymptotically cheaper to re-read than to re-derive, which is what makes the cache safe to lean on rather than merely fast.

The descent reaches the network interface. The security discipline is only as deep as its lowest layer, so we are concrete about the bottom of the descent. The fingerprint-faithful fetch is not only a diagram: it is implemented as the orchestrator's curl-impersonate fetch (`src/capture/curl-impersonate-f` backed by a vendored uTLS CONNECT-proxy daemon (`src/cdp/proxy/utls-daemon.ts`) across four platforms (darwin/linux × amd64/arm64), so the agent's TLS ClientHello and HTTP/2 settings reproduce a real browser's JA3/JA4 signature at the *network interface*, not merely a spoofed User-Agent header. This is the packet layer of screen→browser→CLI→OS→kernel→packet made real: the same signed identity that authorises a route also emits the bytes that carry it, and the bytes are browser-indistinguishable on the wire.

Coverage of the full descent, measured. Because the descent is what lets the agent reach data behind TLS-fingerprinting anti-bot, we measure whether it actually returns results across a broad, diverse intent space rather than on cherry-picked domains. A live coverage gate runs the real `unbrowse search` binary — which exercises the whole descent including the packet-layer fetch — over a diverse intent set and counts the fraction that return real results; on the current graph it returns results on every probed intent (coverage = 1.0 against a ≥ 0.75 release threshold). `scripts/coverage-gate.sh` is re-runnable and fails honestly when the descent cannot reach the data. The dollar saving the wedge measures [1] — a cold browser rediscovery at \$0.10–0.53 collapsed to a \$0.005–0.02 one-time install — rides on this same descent: cheaper because the packets are emitted once and the route reused, not re-derived through a browser on every call.

Per-task retrieval behind anti-bot, measured. We now measure directly what an earlier draft deferred: per-task retrieval on an adversarial, heavily-JavaScript-gated site. On a nine-post corpus drawn from three communities of a major social platform whose HTML surface is JavaScript-challenge-gated and whose read endpoint returns HTTP 403 to a naive client — ground-truthed against the platform's own listing data — a naive HTTP client is blocked on **100%** of requests, while the descent retrieves the real content on **9/9** posts (400–820 KB each) and recovers the ground-truth author handle and a distinctive title token on **100%** of them. This *anti-bot retrieval* head-to-head (a re-runnable anti-bot retrieval suite) is the per-task adversarial-site measurement the descent is built for: naive 0/9, descent 9/9, on a site that 403s ordinary scrapers.

Execute, don't guess — the same principle, measured at model scale. The discipline this paper applies to the web — call the real interface and execute it, rather than have an agent re-derive it — holds for models too: route to a real tool and execute, instead of guessing from weights. A reproducible, gated benchmark suite (`bench/BENCHMARKS.md`) shows a small on-device model (Qwen2.5-1.5B) routed to a library of executable tools turning tasks it fails from weights alone into tasks it solves reliably. Two are genuine tools-versus-no-tools gains on the same 1.5B model: knowledge absent from the weights **0% → 95%** by retrieve-then-execute, and applying a retrieved skill rather than reasoning it from scratch **63% → 93%**. Two are distillation gains on the served model: code-correctness **68% → 100%** (raw base to distilled, in-distribution) by routing to a real executor, and hard-reasoning families **50% → 92%** by distilled routing — measured against a trained-specialist baseline, a scope we flag rather than overclaim. The architecture is the capability, not the raw weights — the same claim the route graph makes for the web.

Self-improving by reuse. The system improves by running against itself. Resolving a fixed probe set repeatedly, latency falls **21.1 s cold → 4.1 s warm (−80.7%)** as the route cache fills, then plateaus (tail spread 4.9% over the last five of twenty passes). The plateau is a physical limit, not a tuning choice: once every route is cached, further passes cannot reduce latency. Reuse, not retraining, is where the compounding comes from.

Credentials are wallet-bound, witnessed end-to-end. The auth descent is exercised against live endpoints: a credential is *sealed to the holder's wallet*, revealed only for the correct key (a wrong key fails closed), and attached to a real authorised request — an authenticated search and an authenticated action each round-trip successfully, with the agent ever holding only a content-addressed commitment, never the raw secret (`bench/agent-experience/`, re-runnable). Sealing, fail-closed reveal, and the authorised round-trip are all witnessed, not asserted.

The pointer-only invariant, scanned. The claim that the stateless binary writes no secret to disk — only pointers and signatures cross the wire — is not left as an assertion. A re-runnable capability gate (`bench/capability/`) scans every persisted session file and counts any plaintext secret. On the current build it scans **446** session files and finds **0** leaks: the redaction invariant (sealed value, pointer at rest) holds across the whole on-disk surface, not just the paths under test. This is the sealed cache of §6 measured as a property of the product, not only of the reference implementation.

BrowseComp and warm-cache self-improvement, measured. On OpenAI's **BrowseComp** multi-hop browsing benchmark, driving the same agent and grader through the route-graph search path across repeated tries, the route/content cache warms run-over-run: per-query wall-clock falls from **80.8 s** on the cold graph to **65.4 s** once warmed (**−19%**), the capture→index→reuse self-improvement the sealed cache predicts. Accuracy is dominated by the agent harness above retrieval, not by the substrate, and sits honestly below specialised search stacks: our reproducible single-shot figure is **0.100** against Exa's published **0.336**, a gap we report rather than hide. We record the full per-try ledger in `bench/browsecomp/SELF-IMPROVEMENT.md` rather than headline a number this minimal agent does not earn; where the substrate *does* win head-to-head is anti-bot retrieval, measured next.

What this establishes, and what it does not. The substrate tests establish that the security primitives are correct as implemented; the micro-benchmark establishes that sealed-cache reuse is cheap; the vendored uTLS layer and the coverage gate establish that the descent reaches the network interface and returns real results broadly; the anti-bot and wallet-auth measurements establish that the descent retrieves real content past JavaScript-challenge anti-bot and that credentials bind to the wallet end-to-end. What remains out of this paper's scope is *end-to-end multi-hop task accuracy* on the open web — which is dominated by the agent harness above the retrieval layer, not the substrate (we report our reproducible BrowseComp figure and its warm-cache behaviour in the repository, honestly below specialised search stacks) — and emitting *real signed* OS/kernel/packet syscalls across platforms: the fingerprint-faithful HTTP emission is shipped; raw signed syscall descent is referenced, not claimed.

12 Threat model

We state the adversary explicitly, because a trust paper that never names what it defends against is decoration. We assume a Dolev–Yao network attacker [14] — able to read, drop, replay, and forge messages on any layer — plus three application-level adversaries the route economy specifically invites. For each, we name the atom that resists it and the residual risk we do not claim to close.

(A1) The impersonator. An attacker replays or forges an action to act as another identity. Resisted at the *root*: every action carries an Ed25519 signature over its canonical bytes [15], and the ledger's `prev`-hash makes a replayed entry detectable out of order [22]. Residual: key theft is out of scope — a stolen wallet key is the user, by construction. This is the same boundary every signing system accepts.

(A2) The credential thief / over-reach. An operator retargets imported sessions at accounts they do not own — the abuse vector that closed the source [3]. The *witness* atom narrows it: credentials are bound to the identity by zero-knowledge proof and revealed only under signature [18,

17], so a leaked ledger or cache exposes *that* a credential is bound, never the credential. Capability attenuation [30] bounds what a delegated action may do. Residual: an operator authenticated as themselves can still drive their own credentials; we constrain reach, not self-harm.

(A3) The poisoner. An adversary publishes a false or stale route to mislead later agents — the integrity attack on a shared graph. Resisted at *witness* and *ledger*: every route attestation is signed and appended to the hash-chained log [22], and finalisation can require a t-of-n quorum [34], so one dishonest maintainer cannot unilaterally settle a claim and any tampered or reordered attestation is detectable. Residual: routes are re-derived to verify rather than trusted on sight; the model moves trust-sensitive traffic to corroborated routes, it does not claim every route is adversary-proof.

(A4) The free-rider. An actor consumes paid execution without paying. The *verb* atom binds payment to the act: an unpaid `execute` gets HTTP 402, not service [4], while discovery and internal-API routing stay free. Residual: the free tier is, by design, free to consume; the payment gate sits at execution, which is where the cost actually lands.

(A5) The public-boundary leak. An adversary — or an honest contributor by accident — leaks the closed capture/integrity engine into a public artifact. Resisted mechanically, not by policy: `leak-guard.sh` and `paper-gate.sh` run in release CI and fail the build if a sensitive term or an unanchored claim reaches a public path. This paper is itself subject to that gate. Residual: NDA source review remains the only full-disclosure path, by the abuse reasoning of [3].

What we do not defend. We make no claim against a malicious endpoint that lies in its own responses (a route can faithfully replay a hostile API), a global passive adversary correlating timing across the whole network, or coercion of a key holder. These are out of scope and named so the reader is not misled by silence.

13 What is built, what is referenced (no fabricated green)

In the spirit of not selling a roadmap as a changelog, we separate what runs in the product, what is available as runnable code, and where the work stops:

- In the product: Intent → route resolve → execute; live browser capture; HTTP fetch with browser-faithful TLS fingerprinting; wallet-signed admission; route/endpoint caching. The node-at-every-layer is no longer a description of separate modules: the contract substrate now ships *embedded* in the stateless binary and is exercised in-process on the resolution hot path (`src/values/contract-native.ts`), and a single source-of-truth is bound across the CLI, the server, and the site so the same signed node adjudicates at each surface (`src/values/contract-chain.ts`), persisted to an append-only on-chain ledger (`src/values/iq-ledger.ts`). One compiled substrate, instantiated at every layer — the paper's thesis made literal rather than illustrative.
- The cross-layer security primitives are implemented, each with tests that execute the claim: the signed descent through every layer with vertical wallet ownership (`src/values/signed-descent.ts`, `src/values/wallet-hierarchy.ts`, `src/values/layer-wallet-descent.ts`), ZK credential binding — the central contribution — and its capture-boundary wiring (`src/values/zk-binding.ts`, `src/capture/zk-bound-hole.ts`), the sealed-unless-revealed cache (`src/values/wallet-seal.ts`), the signature-keyed, recomputable resolution cache and its ledger of resolutions (`src/values/resolution-ledg` `src/values/kv-fallback-pipe.ts`), and the hash-chained signed ledger (`src/values/sealed-ledger.ts`).

The architecture is backend-as-harness — the client surfaces only the holes to fill plus wallet-sealed auth, and the backend returns nothing but structure; the composed flow and its sealing primitives are implemented server-side (`src/capture/backend-reveng-endpoint.ts`), so the reverse-engineering engine no longer rides in the client. The stateless binary writes no local state — only pointers and signatures cross the wire, a property measured by the 446-file leak scan reported in the evaluation.

- The runnable reference suite in `reference/` remains the cited foundation for each primitive above and for the parts not yet in the product: Merkle-root checkpoints with inclusion proofs (`ledger/checkpoint.py`) and the inverse client/OS harness as a capability-gated, content-addressed pipe (`pipes/pipe_contract.py`). The companion Aiko engine mirrors this boundary: the server owns recursive contract compilation, the CLI is the bridge, and the client sees only holes, approvals, and typed pointers rather than graph-control verbs. The on-chain deployment of the checkpoint root, the ERC-8004 registry binding, and a real signed OS/kernel/packet descent across platforms remain integration work, honestly labelled. The same honesty applies to the end state this embedded substrate points at — a site that is a thin reader of on-chain public state, carrying pointers and sealed payloads only while the closed reverse-engineering compute stays off-chain — which is planned but is not yet the live serving path, and is therefore reported as a frontier, not a feature.
- Zero-edit drop-in replacements for the libraries an agent would otherwise reach for exist as packages backed by the route graph: an `exa-py` drop-in (`packages/py-exa`) and a `browser-use` drop-in (`packages/py-browser-use`), each providing the upstream's public surface so a single import swap routes the call through Unbrowse instead.

Treat the reference suite as running, tested code with cited foundations — not a feature list with a release date, and not a research agenda either. The gap between running code and a press release is the entire difference between a whitepaper and a pitch, and we would like to stay on the correct side of it.

14 Conclusion

An agent that is accountable for its own actions has to be one coherent entity across every layer it touches: under one key, revealing nothing it did not choose to reveal, and never emitting an unsigned action — at the HTTP call, the browser session, the keychain read, and the packet on the wire alike. The contribution is deliberately narrow: one discipline — sign it, seal it, cache it, bind identity without disclosure — that holds all the way down the stack, with each layer pinned to an established primitive and the cache–ledger core shipped as runnable, tested code rather than a diagram. Route discovery makes the web cheap to read [1]; securing the descent that executes those routes is a separate problem, and for it, cryptography is what you needed.

References

[1] L. Tham, N. Mac Gregor Garcia, J. Hahn. *Internal APIs Are All You Need: Shadow APIs, Shared Discovery, and the Case Against Browser-First Agent Architectures*. arXiv:2604.00694, 2026.

[2] L. Tham. *Unbrowse Maintenance Network: Proof of Indexing and Bonded Accountability in a Shared Route Graph*. Unbrowse AI, 2026.

[3] Unbrowse AI. *Open Source Notice — the closed/open boundary*. docs/OPEN-SOURCE-NOTICE.md, 2026.

[4] Coinbase. *x402: An Open Protocol for Internet-Native Payments over HTTP 402*. x402.org whitepaper, 2025. https://github.com/coinbase/x402.

[5] J. H. Saltzer, D. P. Reed, D. D. Clark. *End-to-End Arguments in System Design*. ACM TOCS 2(4):277–288, 1984. DOI: 10.1145/357401.357402.

[6] ITU-T Recommendation X.200 (= ISO/IEC 7498-1), *OSI Basic Reference Model*, 1994.

[7] S. Yao, J. Zhao, et al. *ReAct: Synergizing Reasoning and Acting in Language Models*. ICLR 2023. arXiv:2210.03629.

[8] X. Deng, Y. Gu, et al. *Mind2Web: Towards a Generalist Agent for the Web*. NeurIPS 2023. arXiv:2306.06070.

[9] S. Zhou, F. F. Xu, et al. *WebArena: A Realistic Web Environment for Building Autonomous Agents*. ICLR 2024. arXiv:2307.13854.

[10] B. Zheng, B. Gou, et al. *GPT-4V(ision) is a Generalist Web Agent, if Grounded (SeeAct)*. ICML 2024. arXiv:2401.01614.

[11] T. Schick, J. Dwivedi-Yu, et al. *Toolformer: Language Models Can Teach Themselves to Use Tools*. NeurIPS 2023. arXiv:2302.04761.

[12] S. G. Patil, T. Zhang, et al. *Gorilla: Large Language Model Connected with Massive APIs*. NeurIPS 2024. arXiv:2305.15334.

[13] Anthropic. *Model Context Protocol*. 2024. https://modelcontextprotocol.io.

[14] D. Dolev, A. C. Yao. *On the Security of Public Key Protocols*. IEEE Trans. Information Theory 29(2), 1983. DOI: 10.1109/TIT.1983.1056650.

[15] S. Josefsson, I. Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. RFC 8032, IRTF CFRG, 2017.

[16] *ERC-8004: Trustless Agents*. Ethereum Improvement Proposals (Draft). https://eips.ethereum.org/EIPS/eip-8004

[17] J. Camenisch, A. Lysyanskaya. *An Efficient System for Non-transferable Anonymous Credentials with Optional Anonymity Revocation*. EUROCRYPT 2001, LNCS 2045. IACR ePrint 2001/019.

[18] F. Baldimtsi, K. Chalkias, et al. *zkLogin: Privacy-Preserving Blockchain Authentication with Existing Credentials*. ACM CCS 2024. arXiv:2401.11735.

[19] Semaphore Protocol. *Semaphore: anonymous signaling on Ethereum*. https://github.com/semaphore-protocol/semaphore

[20] T. P. Pedersen. *Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing*. CRYPTO 1991, LNCS 576. DOI: 10.1007/3-540-46766-1_9.

[21] R. C. Merkle. *A Digital Signature Based on a Conventional Encryption Function*. CRYPTO 1987, LNCS 293. DOI: 10.1007/3-540-48184-2_32.

[22] B. Laurie, A. Langley, E. Kasper. *Certificate Transparency*. RFC 6962, IETF, 2013.

[23] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. https://bitcoin.org/bitcoin.pdf

[24] A. Yakovenko. *Solana: A New Architecture for a High Performance Blockchain*. 2018. https://solana.com/solana-whitepaper.pdf

[25] Protocol Labs. *IPFS / IPLD Merkle-DAG specifications*.

[26] A. Vaswani, N. Shazeer, et al. *Attention Is All You Need*. NeurIPS 2017. arXiv:1706.03762.

[27] W. Kwon, Z. Li, et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. SOSP 2023. arXiv:2309.06180; https://github.com/vllm-project/vllm. https://github.com/ipfs/specs

[28] S. Chacon, B. Straub. *Pro Git*, 2nd ed., §10.2 Git Internals — Git Objects. https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

[29] D. McGrew. *An Interface and Algorithms for Authenticated Encryption*. RFC 5116, 2008.

[30] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[31] curl-impersonate. *A special build of curl that impersonates real browsers (TLS/HTTP fingerprints)*. https://github.com/lwthiker/curl-impersonate

[32] Salesforce. *JA3: TLS client fingerprinting*. https://github.com/salesforce/ja3

[33] FoxIO. *JA4+ network fingerprinting suite*. https://github.com/FoxIO-LLC/ja4

[34] C. Komlo, I. Goldberg. *FROST: Flexible Round-Optimized Schnorr Threshold Signatures*. SAC 2020, LNCS 12804. IACR ePrint 2020/852; RFC 9591.

[35] J. R. Boyd. *The Essence of Winning and Losing*, 1995; in *A Discourse on Winning and Losing*, ed. G. Hammond, Air University Press, 2018.

[36] M. Burgess. *An approach to understanding policy based on autonomy and voluntary cooperation*. DSOM 2005, LNCS 3775, pp. 97–108. See also J. A. Bergstra, M. Burgess. *Promise Theory: Principles and Applications*, 2014.

[37] B. Liskov, L. Shrira. *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems*. PLDI 1988, pp. 260–267. Originating the future construct: H. C. Baker, C. Hewitt. *The Incremental Garbage Collection of Processes*. Proc. Symp. on AI and Programming Languages, ACM, 1977.