

Private AI use in four layers

How PARANOID keeps an AI question from revealing who asked it, where it came from, who it is about, and what it says, and what each of those claims rests on.

Who asked

Zero-knowledge membership on Solana

From where

Oblivious HTTP through an independent relay

About whom

Personal details removed in the browser

What was asked

Sealed to an attested chip

Abstract

Every request to a hosted AI model leaks four things: the identity of the person paying for it, the network address it came from, the people named inside it, and the question itself. Most AI services ask users to trust a privacy policy for all four. PARANOID replaces that trust, layer by layer, with mechanisms a user's own browser can check.

Payment is a zero-knowledge proof of membership in a set of paid members held by a Solana program, using rate-limit nullifiers so that a ticket spent twice reveals the member's secret and forfeits the membership. Requests travel as Oblivious HTTP through a relay run by an independent company, so no single operator sees both the address and the request. Names, email addresses, card numbers and similar details are replaced with placeholders by a model running inside the browser before anything is sent. Finally, the question is sealed with ML-KEM-768 to a key generated inside a confidential-computing machine (Intel TDX with NVIDIA GPUs in confidential mode, operated by Chutes), and the browser checks the machine's signed attestation against public references before it sends anything.

This paper describes each layer as it is implemented today, states precisely what each party can and cannot observe, lists the assumptions each guarantee depends on, and shows how the hardest cases are handled.

Contents

1	Vision
2	What an AI request leaks today
3	Threat model and goals
4	Architecture
5	Layer one: anonymous payment
6	Layer two: network separation
7	Layer three: people named in the question
8	Layer four: the question sealed to a chip
9	Cover traffic
10	Economics
11	Measurements and verification
12	Hard cases
	References and appendices

1 Vision

People have started telling AI models what they would not tell a search engine, and in many cases not a doctor or a lawyer either. They type in debts, diagnoses, a plan to leave a job or a marriage, and the names of everyone involved. All of it lands in a record tied to an account, a card and an IP address, and the record goes back years now.

On 16 October 2026 Vitalik Buterin described a "paranoid mode" for his own AI agent: zkAPI payments so the provider cannot identify the payer, a separate Tor circuit for each conversation, random cover traffic, and a local model that rewrites prompts before they leave the machine [2]. It is a strong design, and he was candid about its costs: it is slow, it is a power-user setup, and Tor is "far from optimal" for this traffic. Two weeks earlier the Ethereum Foundation and the Open Anonymity Project had launched zkAPI on Ethereum mainnet, built on the anonymous usage-credit design Davide Crapis and Buterin published in February [1][3].

PARANOID starts from that work and asks one question: what would it take for this level of protection to be the default for anyone, in a normal chat window, with no software to install? Our answer has four layers, one for each thing a request leaks. We did not invent the ideas underneath them. Rate-limit nullifiers, Oblivious HTTP, named-entity recognition and confidential computing all predate us. What PARANOID contributes is an implementation that puts all four in one path, on Solana, behind an interface that looks like any other AI chat, with every claim checkable by the user's own browser.

As far as we know, PARANOID is the first AI service that protects all four: who asked, from where, about whom, and what. If an earlier service does all four, we want to hear about it and will correct this paper.

Design principles

- **Checks over promises.** Wherever a guarantee can be checked by the client, the client checks it, and a failed check stops the request before anything is paid.
- **Split knowledge.** No single operator, including us, should hold two of the four things at once.
- **Fail visibly.** When a stronger path is unavailable and a weaker one is used, the interface says so next to the answer.
- **Name every trust assumption.** Section 3 names every party the system relies on, including the hardware vendors, and Section 12 shows how each hard case is handled.

2 What an AI request leaks today

A question sent to a hosted model does not travel alone. The provider learns who paid, which network it came from and who is named in it, on top of the text itself, and the table below takes those four one at a time. They do not overlap. A service that hides the payer has done nothing about the address, and one that hides the address has done nothing about the names in the question, which is why the fixes we have seen so far each leave three of the four exactly where they were.

Leak	How it leaks today	Why it matters
Who asked	Accounts, payment cards, API keys and phone verification	Every prompt joins a permanent profile that can be subpoenaed, breached, sold or used for training decisions.
From where	The IP address of every connection	An address identifies a household or workplace and links sessions even without an account.
About whom	Names, emails, phone numbers and account numbers of third parties typed into the prompt	People who never used the service end up in its records, without consent.
What was asked	The prompt and answer in plain text on the provider's servers	Logs, staff access, breaches and legal orders all reach the content itself.

Why policies are not enough

"We do not log" and "we do not train on your data" are promises about behaviour, and a promise is all they are. The user cannot test them, the wording can change next quarter, and neither a buyer of the company nor a judge is bound by them. In 2025 a court ordered an AI provider to keep every user conversation, deleted ones included, because the conversations might be evidence in a lawsuit [18]. An operator that means every word of its policy is still one building where identity, address and content sit together, and a building can be served with an order.

Why the existing pieces fall short alone

Each of the four leaks already has a known fix. The table compares the usual approaches by which of the four they close on a single request.

Approach	Who asked	From where	About whom	What was asked
Typical AI service	Exposed	Exposed	Exposed	Exposed
Anonymous payment (zkAPI, Anonymous Credit Tokens [4])	Hidden	Exposed	Exposed	Exposed
VPN or Tor	Exposed	Hidden	Exposed	Exposed
No-log "private AI" services	Promised	Promised	Promised	Promised
Confidential computing	Exposed	Exposed	Hidden	Hidden
PARANOID	Hidden	Hidden	Hidden	Hidden

Hidden means a mechanism that the user's browser or the chain can check keeps it out of view. Promised means only the operator's word does. The Open Anonymity Project's own documentation notes that zkAPI covers the payer and leaves the rest to other tools [3]. PARANOID exists to close all four on the same request, because taking knowledge away from one party is only worth something if it does not reappear with the next party down the line.

3 Threat model and goals

Parties

Party	Role, and what we assume about it
Member	A person or agent who bought questions. Its device and browser are trusted; a compromised device is out of scope.
PARANOID gateway	Our server. Checks payment proofs and forwards requests. Treated as curious: it may try to learn who asked and what was asked.
Relay	Oblivious Network, an independent company running an Oblivious HTTP relay. Treated as curious, and assumed not to collude with the gateway.
Model host	Chutes, which operates the confidential-computing machines. Treated as curious about content; trusted only as far as its published, attested software build.
Hardware vendors	Intel and NVIDIA, whose signatures root the attestation. Trusted for the integrity of their confidential-computing features.
Network observer	Anyone watching traffic on part of the path. Section 12 covers pairing traffic by timing and size.
Other members	May try to spend more than they paid for.

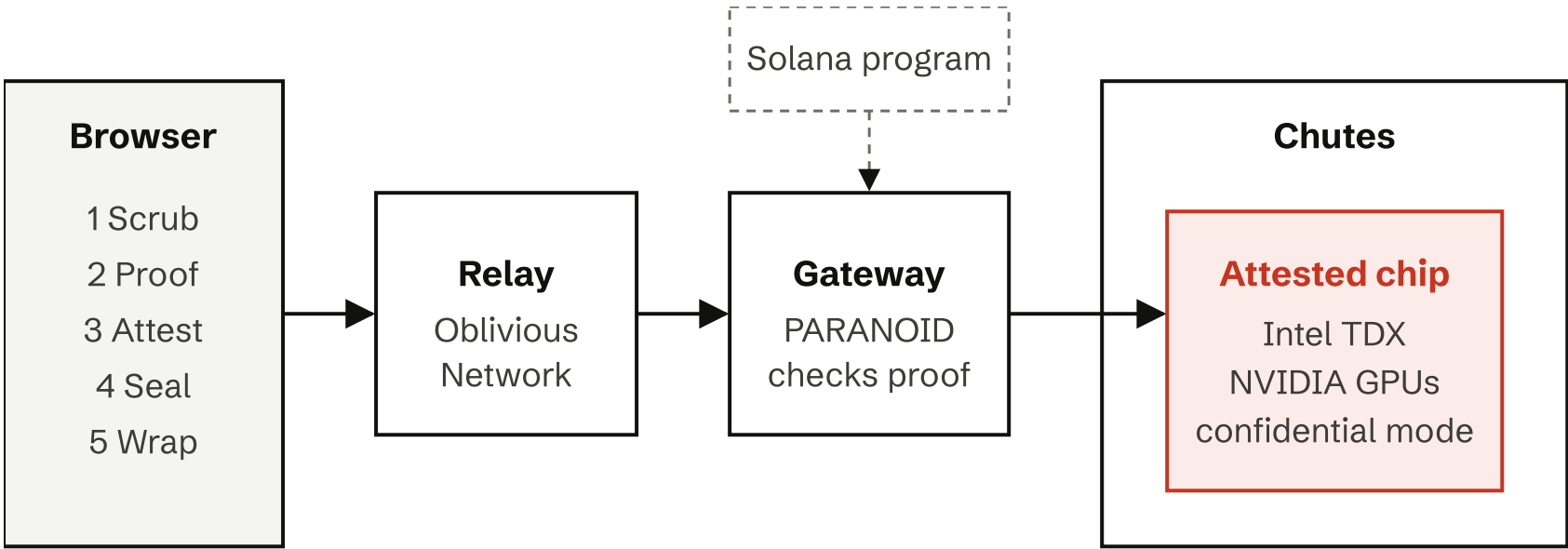
Goals

1. **Payer unlinkability.** The gateway cannot tell which member made a request, beyond the fact that it was some member with unspent tickets.
2. **Request unlinkability.** The gateway cannot tell whether two requests came from the same member.
3. **Address separation.** The party that sees a member's IP address never sees the request, and the party that handles the request never sees the address.
4. **Third-party minimisation.** Names and contact details of other people do not leave the browser.
5. **Content confidentiality.** Only an attested machine running published software can read the question or the answer.
6. **Fail closed and fail visibly.** A request whose protections cannot be verified is not sent on that path, and any fallback is labelled.

7. **No free riding.** A member cannot spend more questions than it bought without forfeiting its membership.

4 Architecture

A question passes through five parties. The figure shows what each one receives; the table that follows shows what each one can learn.



Sees the member's address and an opaque blob at the relay; sees a proof and a sealed blob at the gateway; opens the question only inside the chip. The answer returns the same way, sealed to a key that exists only in the member's browser.

Figure 1. The request path. The gateway reads the current member roots from the Solana program; it never sees the member's address or the question.

Who can learn what

	Who you are	Your IP address	People you name	Your question
Relay	No	Yes	No	No, sealed
PARANOID gateway	No, only a proof	No, sees the relay	No, placeholders	No, sealed
Chutes servers	No	No, sees the gateway	No	No, sealed
Attested chip	No	No	No, placeholders	Yes, to answer it

The table holds when the member uses the IP shield, Scrub, and a model served on a chip. Section 12 covers the cases where it does not.

5 Layer one: anonymous payment

A member pays once, on chain, and then proves for each question that it is some paid member with an unspent ticket, without saying which.

Membership

A member buys a pack of questions with USDC. On chain the purchase is one leaf in a Poseidon Merkle tree of depth 20. The program remembers the last 64 roots rather than only the newest, so a proof built against a root from a few minutes ago still verifies after newer members have joined. The leaf binds three values:

```
leaf = Poseidon( Poseidon(secret), endEpoch, questions )
```

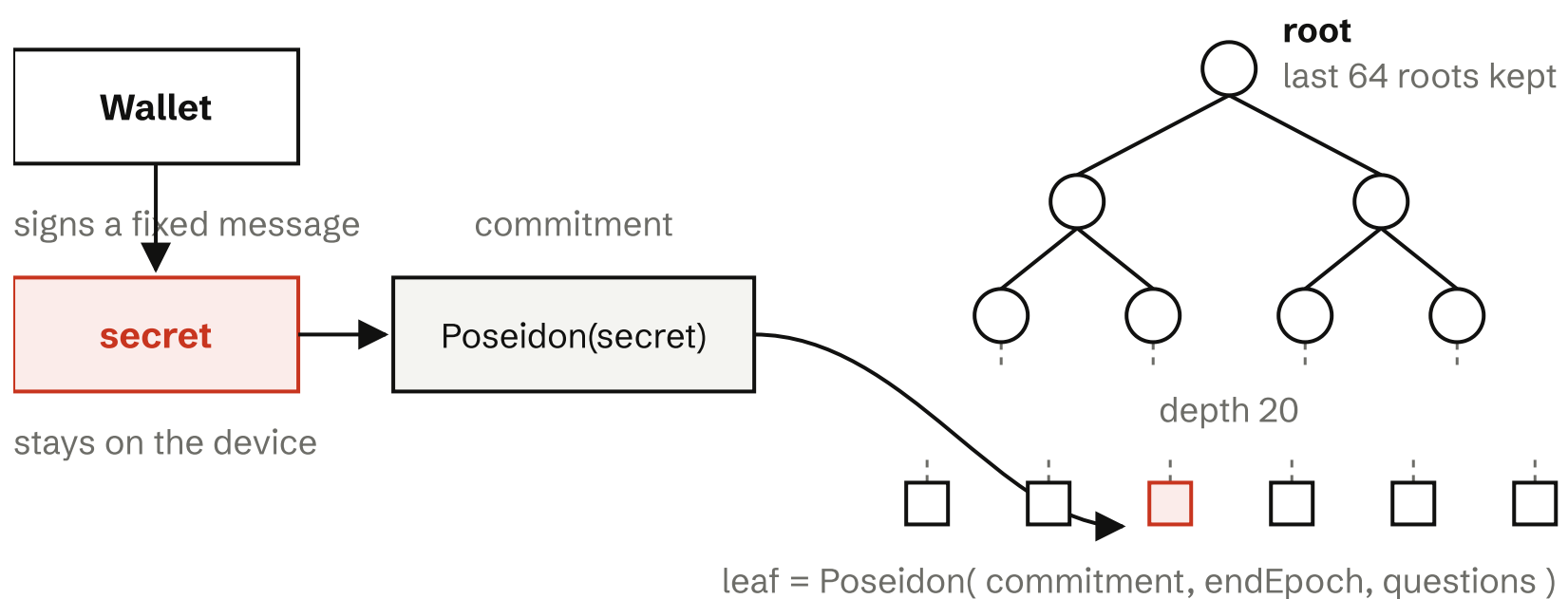


Figure 2. Membership. The wallet's signature becomes a secret that never leaves the device; only its hash goes into the leaf, and the chain keeps the last 64 roots so a proof made against a slightly older root still verifies.

The secret never leaves the member's devices. It is derived from a signature by the member's wallet over a fixed message, reduced into the BN254 scalar field, so the same wallet recovers it on any device and nothing about it is stored by us. The depositing wallet is public and linked to the leaf; the requests are not linked to either.

The request proof

Each request carries one or more Groth16 proofs over BN254 for the circuit Request(20, 4): 7,467 constraints, five public inputs (root, epoch, x, limit, weight) and up to four tickets per proof. The proof states that the prover knows a secret and a path to a leaf under a recent root, that the membership is valid on this day, and that it is spending tickets start to start + weight, all below the number of questions bought. Which tickets, and which leaf, stay private.

Algorithm 1. Request proof, made in the browser

```
Private: secret, path to the leaf, endEpoch, questions, start
Public: root, epoch, x, limit, weight

1. commitment  $\leftarrow$  Poseidon(secret); leaf  $\leftarrow$  Poseidon(commitment, endEpoch,
   questions)
2. require MerkleRoot(leaf, path) = root, with root among the 64 the
   chain keeps
3. require epoch  $\leq$  endEpoch
4. require  $1 \leq \text{weight} \leq 4$  and  $\text{start} + \text{weight} \leq \text{questions} \leq \text{limit}$ 
5. for each ticket  $t$  in start ... start + weight - 1:  $a_1 \leftarrow$  Poseidon(secret,
   Poseidon(0, "paranoid.v2"),  $t$ ); nullifier_ $t$   $\leftarrow$  Poseidon( $a_1$ );  $y_t \leftarrow$ 
   secret +  $a_1 \cdot x$ 
6. output a Groth16 proof over BN254 of steps 1 to 5, with every
   nullifier_ $t$  and  $y_t$ 

Gateway: verify the proof against a root it read from Solana; refuse if
any nullifier_ $t$  is already in its book; record them; forward the request.
```

Ticket accounting uses rate-limit nullifiers in the form of RLN v2 [5]. For each ticket t :

```
 $a_1 = \text{Poseidon}(\text{secret}, \text{Poseidon}(0, \text{"paranoid.v2"}), t)$ 
nullifier = Poseidon( $a_1$ )
 $y = \text{secret} + a_1 \cdot x$ 
```

Here x is a fresh random field element chosen by the client for each key. The gateway records every nullifier it accepts. Because the namespace never resets, a ticket can be spent once in the life of a membership. A ticket spent twice publishes two points (x, y) on the same line, from which anyone can compute the secret; the program then accepts that secret as proof of double use, checks Poseidon(secret) against the member's commitment, and forfeits the membership. No reward goes to the reporter, so there is nothing to front-run.

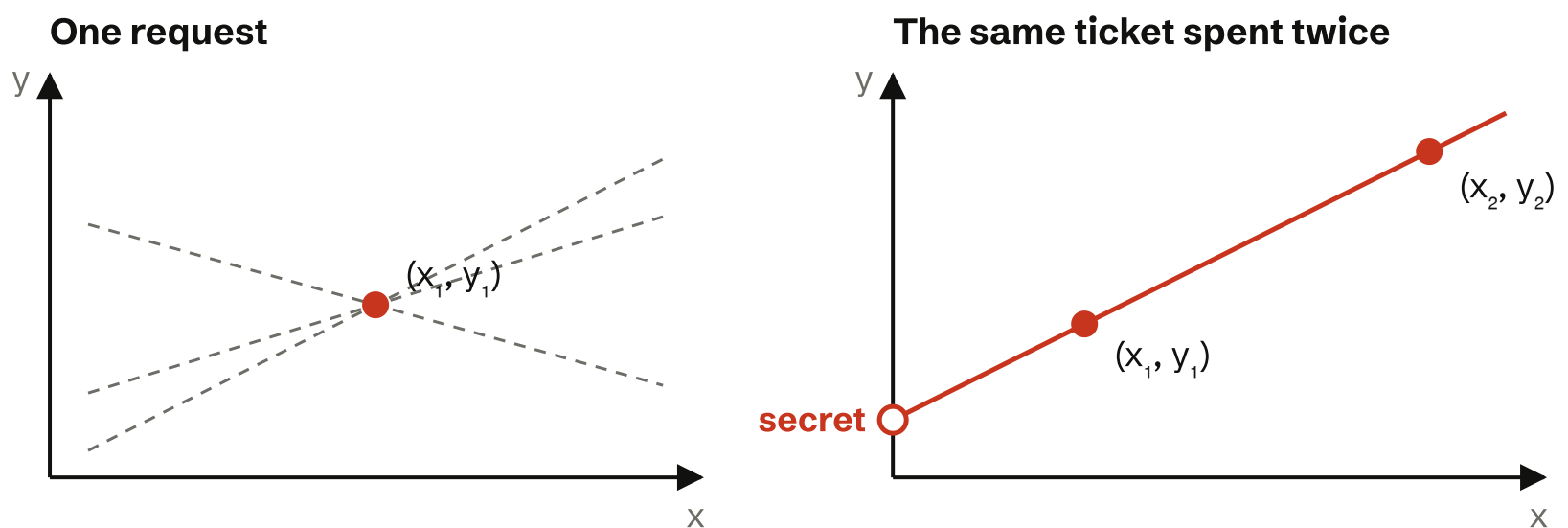


Figure 3. Rate-limit nullifiers. Each request publishes one point on a line whose intercept is the member's secret. One point says nothing; a ticket used twice gives two points, and anyone can read the secret off the line and slash the membership.

What the gateway learns

Per request it learns the day, which recent root was used, the number of tickets, one nullifier and one share per ticket, and nothing else. Nullifiers of different tickets are unrelated hashes, and shares are single points on independent lines, so two requests from the same member cannot be linked. The anonymity set of a request is every member whose membership covers that day.

Why rate-limit nullifiers

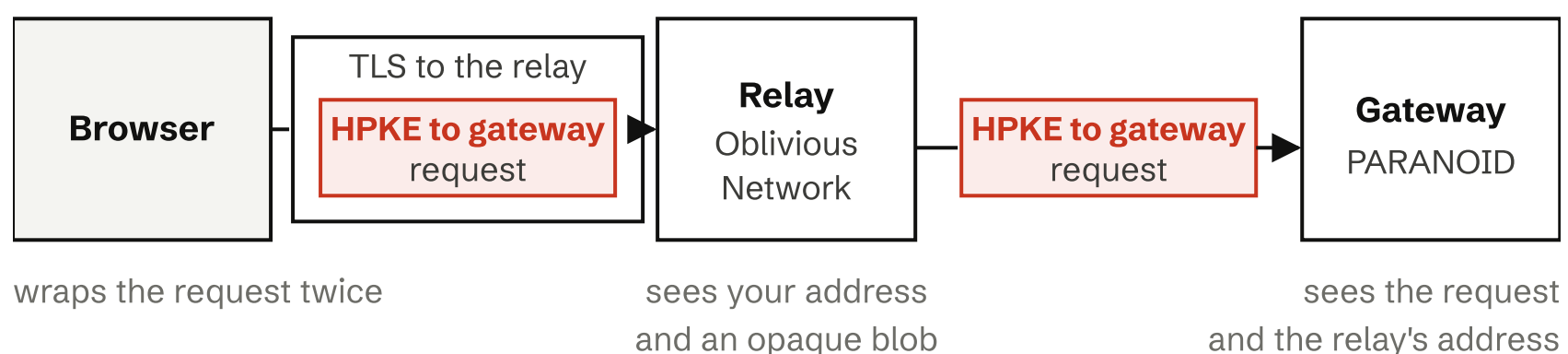
When zkAPI v2 shipped it had moved away from rate-limit nullifiers to balance commitments signed by the server [1][3]. That buys exact metering, and it pays for it with a server signing key, requests that have to go one after another, and a challenge game when a member leaves. We looked at the same trade and went the other way. With nullifiers the gateway needs public data and its own book of spent nullifiers, nothing else. A member's requests do not depend on one another, so they can run in parallel, and a double spend proves itself on chain, with no challenge window to sit through. What we give up is fine-grained pricing. A message to a large model spends several tickets instead of one, and that is as fine as the pricing gets.

6 Layer two: network separation

With the IP shield on, the party that sees a member's address and the party that reads the request are different companies.

Every request leaves the browser as Oblivious HTTP (RFC 9458) [6]. The browser fetches the gateway's key configuration, encrypts the whole request to it with HPKE (DHKEM over X25519, HKDF-SHA256, AES-128-GCM) and posts the ciphertext to a relay, not to us. Plain OHTTP has no notion of a streamed answer, so we use the chunked extension from draft-ietf-ohai-chunked-ohttp [7]; each chunk is sealed on its own as the model produces it.

- **The relay**, run by Oblivious Network, an independent company, sees the member's address and an opaque blob. It forwards the blob to the gateway from its own address.
- **The gateway** opens the blob and sees the inner request and the relay's address; the member's address was never in the blob to begin with. It decapsulates requests for a short fixed list of its own API paths and nothing else, which keeps it from being turned into an open proxy.



The relay peels the outer layer; the gateway peels the inner one. Neither sees what the other sees.

Figure 4. Oblivious HTTP. The request is sealed to the gateway's key before it is sent to the relay, so the relay holds the address and the gateway holds the request, and no single party holds both.

The payment proofs of Section 5 and the attestation exchange of Section 8 take the same route, so the gateway does not learn the address of someone merely checking a chip either.

Keeping the shield up

Address separation rests on the relay and the gateway being run by different parties that do not share what they see. If the relay cannot be reached, the client retries the same sealed request through a backup relay on separate infrastructure, so the shield stays on rather than quietly switching off.

7 Layer three: people named in the question

Questions often contain other people's details. Scrub removes them before anything leaves the browser and restores them only on the member's screen.

People's names are found by a named-entity recognition model, bert-base-NER quantised to 8 bits, that runs inside the browser through transformers.js [8]. The browser downloads it once from a public CDN; the CDN sees that a model was fetched and nothing else. Everything else has a shape, so email addresses, card numbers (checked with Luhn so an order number is not caught), wallet addresses, phone numbers, street addresses and identity numbers are matched by pattern. Every detail found is swapped for a placeholder like [NAME_1] or [EMAIL_1], and the table of swaps lives for the whole conversation, which is what makes "Anna Kim" in the first message and plain "Anna" in the fourth come out as the same placeholder. Places and company names stay in; strip those and most ordinary questions stop making sense.

The model answers in terms of the placeholders, and the browser swaps the real details back in before it draws the answer, nowhere else. A member who wants to know what actually left the machine can open any question and read the sent text, placeholders and all.

Scrub works on names and contact details, the parts of a question that most often identify other people. Every change it makes is visible to the member, so what leaves the browser can always be checked.

8 Layer four: the question sealed to a chip

The question is encrypted in the browser to a key that exists only inside one attested machine, and the browser checks that machine's proof before it sends anything.

The machines

Chutes serves models on machines that run as Intel TDX trust domains with NVIDIA GPUs in confidential-computing mode [9][10][11]. Memory inside a trust domain is encrypted and isolated from the host, and the GPUs refuse to expose their state to the host. Each running instance publishes an ML-KEM-768 public key [12] generated inside the trust domain, together with short-lived nonces.

What the browser checks

Before the first question to an instance in a visit, the browser draws a fresh 32-byte random number n and asks for the instance's evidence for it. That evidence comes back through our gateway, and through the relay as well when the IP shield is on. It does not matter that it passes through our hands, because every part the browser relies on carries an Intel or NVIDIA signature, and a gateway that edited any of it would simply fail the check. Five things have to hold before the browser will use the instance:

#	Check	How
1	Intel signed the quote	The TDX quote (version 4) is verified with Intel's signed collateral from Intel's Provisioning Certification Service, using the dcap-qvl verifier compiled to WebAssembly [13]. The status must be UpToDate.
2	The key was made in that chip	The quote's report data must begin with SHA-256($n \parallel \text{key}$), binding this nonce and this exact public key to this trust domain.
3	Debug mode is off	The trust domain's attributes must show debugging disabled.
4	It runs published software	MRTD and runtime registers RTMR0 to RTMR3 must equal a build that Chutes publishes at a public endpoint, fetched by the browser directly from Chutes [14].
5	NVIDIA passed every GPU	NVIDIA's attestation service returns ES384-signed tokens, verified against NVIDIA's published keys fetched directly from NVIDIA [15]. The overall result must be true, the token must carry the same SHA-256 value as its nonce, and every GPU must report successful measurements with debugging disabled.

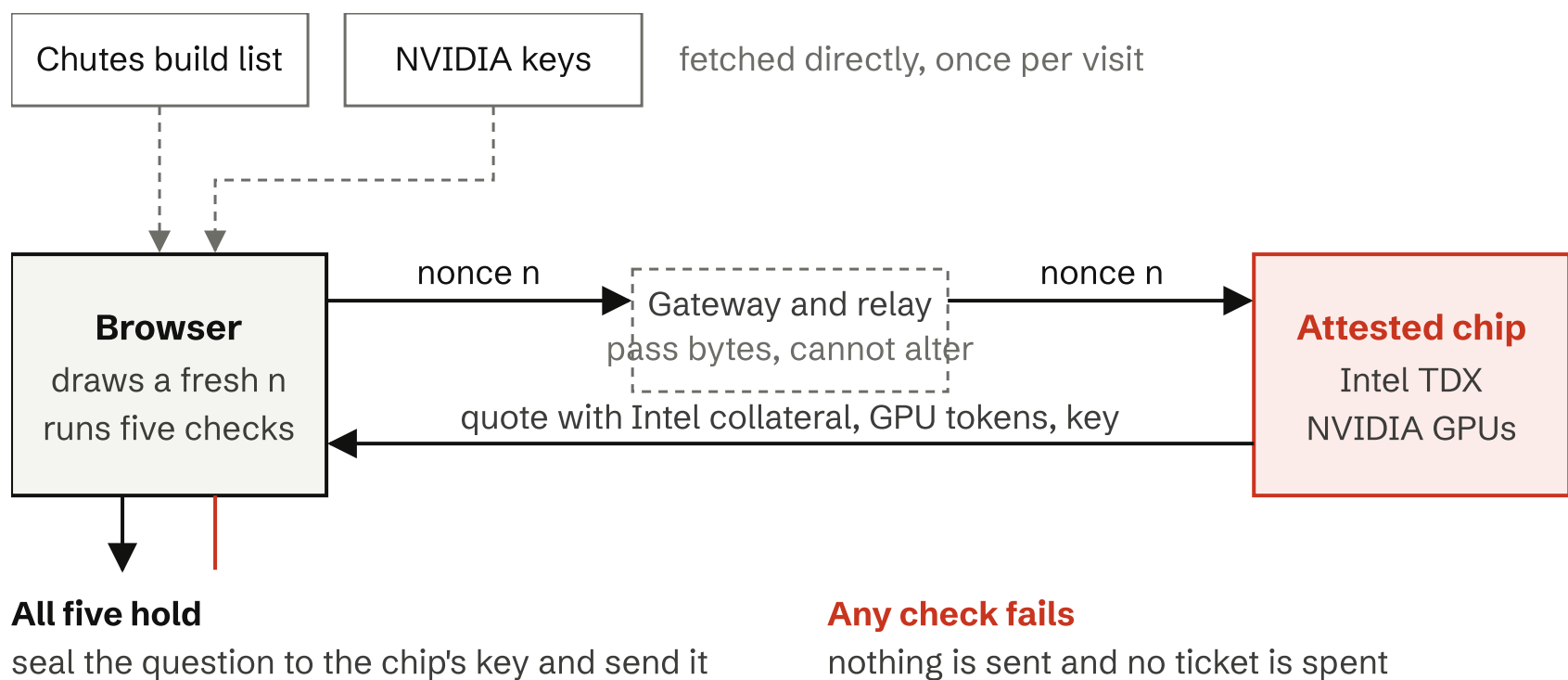


Figure 5. Attestation. The browser challenges the chip with a fresh nonce. The quote comes back with Intel's signed collateral; the build list and NVIDIA's keys the browser fetches itself. The gateway carries the bytes and cannot change any of them without the check failing.

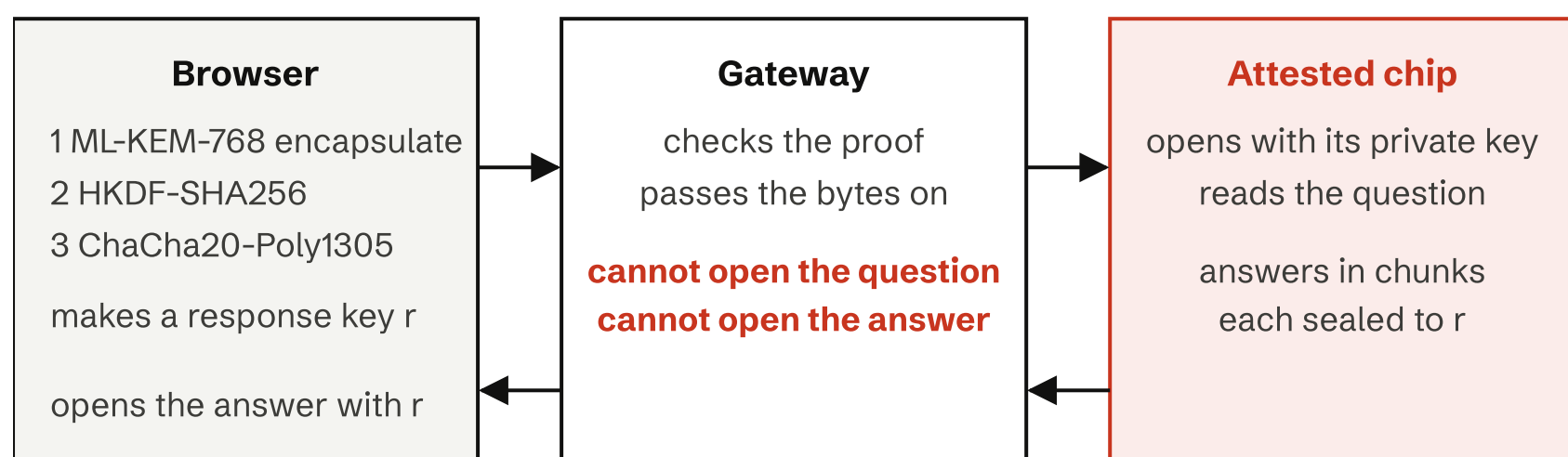
Fail any one of these and the question stays in the browser, with no ticket spent. The two reference lists, Chutes' published builds and NVIDIA's signing keys, are fetched by the browser straight from Chutes and from NVIDIA. They never pass through our servers. The fetch happens once per visit, before the first question, so neither company can tie a download to a question.

Algorithm 2. Accepting a chip, run in the browser before the first question

- Input:** instance key pk , fresh nonce n (32 random bytes), Chutes build list B and NVIDIA signing keys (both fetched directly)
1. $(\text{quote}, \text{collateral}, \text{gpuTokens}) \leftarrow \text{evidence for } n$, fetched through the gateway (and the relay when the shield is on)
 2. verify quote with dcap-qvl against the Intel-signed collateral that came with it; require status = UpToDate
 3. require quote.reportData begins with $\text{SHA-256}(n \parallel pk)$
 4. require quote.tdAttributes shows debugging disabled
 5. require $(\text{MRTD}, \text{RTMR0}, \text{RTMR1}, \text{RTMR2}, \text{RTMR3}) \in B$
 6. verify the ES384 signatures of the NVIDIA tokens with NVIDIA's keys; require overall result true and the nonce in the overall token and in every GPU token equal to $\text{SHA-256}(n \parallel pk)$; require every GPU measured successfully with debugging disabled
 7. if any step fails: stop, send nothing, spend no ticket; otherwise seal the question to pk (Figure 6)

Sealing

Sealing follows the Chutes end-to-end scheme. The browser encapsulates to the instance's ML-KEM-768 key, derives a key with HKDF-SHA256 using the first 16 bytes of the ciphertext as salt, and encrypts the compressed request with ChaCha20-Poly1305. Inside that request it also includes a fresh response key of its own. The chip streams its answer back in pieces, each sealed to that response key, and the only place that key exists is the member's browser. Our gateway checks the payment proof, hands the sealed bytes to Chutes and relays the sealed answer back. It cannot open the question and it cannot open the answer.



The request carries r inside the seal, so only the browser that made r can read the answer.

Figure 6. Sealing. The question is encrypted to a key that exists only inside the attested chip, and the answer is encrypted to a key that exists only in the member's browser. The gateway in between can open neither.

Payment without a readable request

The gateway holds the member's tickets while the chip is asked and spends them only once the chip starts answering. If the chip never starts, for example because it is overloaded, the tickets are released and the browser may send the same question by the ordinary path. When that happens, the answer carries a visible note saying the question did not go through a chip.

Web search without revealing the question

A chip cannot browse the web, which means some words have to leave it for a search to happen at all. The question we had to settle was which words. With web search on, the browser first sends the chip a sealed request asking it to write a short search query for the conversation. That query, and nothing else, goes to the search route, together with a one-use receipt issued for the paid request. The route therefore cannot be used without paying and

never sees the question. The results come back, are sealed to the chip together with the original question, and the answer shows the exact words that went out. The price is one extra question per searched message.

What this layer does not claim

The chip reads the question; that is how it answers. The guarantee is that only an attested machine running Chutes' published build can read it, not that no machine can. The guarantee depends on the confidential-computing features of Intel and NVIDIA hardware and on the published build behaving as described. Section 12 shows how the browser's checks keep up with security updates to that hardware.

9 Cover traffic

A member can turn on decoys. These are short questions drawn from a list of ordinary prompts, sent every 20 to 60 seconds down the same path as real questions, sealed to the same chips and paid for with real tickets. By route, by proof and by content the gateway has no way to tell a decoy from a real question. Decoys make it more expensive to pair a member's connection with their questions by timing.

10 Economics

Members pay for questions up front and spend them with proofs. Because the money changes hands when the pack is bought, nothing is charged per request, and the gateway can be paid without ever knowing who asked or what was asked.

Parameter	Value
Price	0.002 USDC per question, or 500 questions per USDC
Smallest purchase in the app	2,500 questions (5 USDC); the program's own floor is 250
Validity	365 days from purchase or top-up
Most tickets per membership	65,535
Larger models	Spend several tickets per message, according to their cost to run

11 Measurements and verification

Measurement	Result
Request proof, generated in the browser	Under one second
Proof verification at the gateway	About 49 milliseconds
Groth16 verification on chain (benchmark on the previous circuit version; not needed in the flow)	174,478 compute units
Engine test suite (program, circuit, solvency fuzz)	All passing
Sealing tests against live chips, including six forged proofs refused before payment	27 of 27
Sealed web search, direct and through a relay	18 of 18

Each forged-proof test changes exactly one thing in otherwise valid evidence and checks that the browser refuses to send. The six changes are a key the chip did not make, edited Intel collateral, an edited quote, an edited NVIDIA verdict, one GPU missing from the report, and a build measurement Chutes has not published.

12 Hard cases

Some situations press harder on privacy than an ordinary question does. These are the ones PARANOID was designed around, and what it does in each.

Hard case	What PARANOID does
Pairing traffic by timing and size	Requests and answers are sealed end to end and pass through an independent relay. Decoys (Section 9) send real, paid, sealed questions at random intervals, so any pairing an observer attempts has to be sorted out of cover traffic first.
Hiding in a crowd	A request could have come from any member whose membership covers that day. Every purchase adds to that crowd, and decoys from every member add volume to it.

Trusting confidential hardware	The browser accepts a chip only when Intel reports its platform UpToDate, which means every security update Intel has published is applied, when debug mode is off and when its build matches one Chutes publishes. A chip that falls behind is refused before any question is sent.
Models served without a chip	The model menu lists them apart from sealed models, and payment, address separation and Scrub still cover them.
A busy chip	The ticket is held, not spent, until the chip starts answering. If it never starts, the question takes the ordinary path and the answer carries a note saying so.
An unreachable relay	The client retries through a backup relay on separate infrastructure, so the IP shield stays on.
The gateway itself	It checks proofs against public roots on Solana and keeps a single book of spent nullifiers. With the IP shield, Scrub and a sealed model in use, it never holds an address, a name or a question, so it has nothing about a member to hand over.
The member's secret	It is derived from a wallet signature on the member's own device. It is never sent to us or stored by us, and the same wallet re-creates it on any device.

Conclusion

Private use of an AI model should not depend on a setup that only a handful of people could assemble. None of the parts in this paper are ours. Zero-knowledge usage credits came out of research, Oblivious HTTP is an IETF standard, redaction models already run in browsers and confidential computing ships in off-the-shelf servers. What we did was put them on a single path and hide the path behind a chat window that looks like every other one. Each layer takes one kind of knowledge away from one party, and for each one the browser runs a check or this paper shows how the hard case is handled.

PARANOID delivers a system in which the question is readable only inside one checked machine, the payer is one member among many, the address and the request are held by different companies, and other people's names stay on the member's screen. That should be the starting point for every AI service, and this paper shows how each part of it can be checked.

References

1. D. Crapis and V. Buterin. ZK API Usage Credits: LLMs and Beyond. ethresear.ch, February 2026. ethresear.ch/t/24104
2. V. Buterin. "Paranoid mode" for a personal AI agent, announced 16 October 2026; summary in Bitcoin.com News. news.bitcoin.com
3. Ethereum Foundation and Open Anonymity Project. zkAPI. github.com/vbuterin/zkapi
4. S. Schlesinger et al. Anonymous Credit Tokens. IETF CFRG Internet-Draft draft-schlesinger-cfrg-act.
5. Privacy and Scaling Explorations. Rate-Limiting Nullifier (RLN v2).
6. M. Thomson and C. A. Wood. Oblivious HTTP. RFC 9458, IETF, 2024.
7. Chunked Oblivious HTTP Messages. IETF OHAI Internet-Draft draft-ietf-ohai-chunked-http.
8. Hugging Face. transformers.js; dslim/bert-base-NER.
9. Intel. Intel Trust Domain Extensions (Intel TDX). intel.com
10. NVIDIA. Confidential Computing on NVIDIA GPUs. nvidia.com
11. Chutes. Confidential model serving. chutes.ai/docs
12. NIST. FIPS 203, Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM), 2024. csrc.nist.gov
13. Phala Network. dcap-qvl, an Intel DCAP quote verification library. github.com/Phala-Network/dcap-qvl
14. Chutes. Published TEE build measurements. api.chutes.ai/servers/tee/measurements
15. NVIDIA. Remote Attestation Service, published signing keys. nras.attestation.nvidia.com
16. J. Groth. On the Size of Pairing-Based Non-interactive Arguments. EUROCRYPT 2016.
17. L. Grassi et al. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. USENIX Security 2021.
18. The New York Times Co. v. Microsoft Corp. and OpenAI, No. 1:23-cv-11195 (S.D.N.Y.), order to preserve output log data, May 2025.

Appendix A. Public parameters

Item	Value
Solana program	9JJqzEoDGkTE8CMLATRw7CqFhcyWVegfhFuEeGfBfVGm
Circuit	Request(20, 4), 7,467 constraints, Groth16 over BN254
Proving key (SHA-256)	d909d787fd228867708c361fef699676 bc49a5c0adfe140a6d50a539779e69cd
Verifying key (SHA-256)	ee128f7ee298574f043e7710e5c798e1 c0bb6c5f35fe05b9564f05bd4daf483e
Member tree	Poseidon, depth 20, 64 recent roots
Epoch	One day
Nullifier namespace	Poseidon(0, "paranoid.v2"), never reset
Oblivious HTTP suite	DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, AES-128-GCM
Chip sealing	ML-KEM-768, HKDF-SHA256, ChaCha20-Poly1305

Appendix B. Checking it yourself

- Under any answer from a chip, open "Read only inside a sealed chip" to see the instance, the build and the number of GPUs that passed.
- Compare that build with the list Chutes publishes at api.chutes.ai/servers/tee/measurements.
- NVIDIA's signing keys used to check GPU verdicts are at nras.attestation.nvidia.com/well-known/jwks.json.
- Open any question you sent to see the exact text that left your browser after Scrub.
- The program and the member tree can be read directly from Solana at the addresses above.