How routstr confidential mode works

A programmer's guide

Juraj Bednar <juraj@bednar.io>, AI assisted

How routstr confidential mode works

A programmer's guide to protocol v2: what is computed, how, who knows what, and why each property holds

Juraj Bednar <juraj@bednar.io>, AI assisted

10 October 2026

Companion to the paper *Confidential and verifiable upstream inference over standard TLS 1.3*. Describes the code on branch `twopc-v2` of routstr-confidential and `confidential-2pc` of routstr-core and routstr-sdk.

Contents

How to read this viii

1 The problem and the idea 1
- 1.1 What a routing node is today . . . 1
- 1.2 The idea in one paragraph . . . 2
- 1.3 The goals . . . 2

2 Threat model and trust 3
- 2.1 Who may cheat . . . 3
- 2.2 What is trusted . . . 3
- 2.3 What is out of scope . . . 3

3 TLS 1.3, only the parts we need 4
- 3.1 The key schedule is a tree . . . 4
- 3.2 The observation the protocol is built on . . . 5

4 The split: who gets which branch 6
- 4.1 Who holds which secret . . . 6
- 4.2 Epochs on the wire . . . 7

5 What the provider must do 9

6 The components 11

7 One session, step by step 13
- 7.1 Step 0: the offer . . . 13
- 7.2 Step 1: setup and reservation . . . 15
- 7.3 Step 2: preprocessing and the joint key share (arming) . . . 17
- 7.4 Step 3: the TLS handshake with two online 2PC phases . . . 19
- 7.5 Step 4: epoch 0, the credential record . . . 21
- 7.6 Step 5: the body and the record hold . . . 22
- 7.7 Step 6: the response . . . 23
- 7.8 Step 7: disclosure and settlement . . . 24
- 7.9 Who knows what, step by step . . . 25

8 The two-party computation 26
- 8.1 The building blocks, as APIs . . . 27
- 8.2 What the circuits compute . . . 28
- 8.3 How C makes sure the node garbled honestly . . . 29
- 8.4 Preprocessing versus online . . . 31

8.5 The leakage: at most 2.3 bits per session 32

9 The zero-knowledge proofs 34
9.1 The tool 35
9.2 π_NK: the node's record is the published template 35
9.3 π_C2S: the body ends with the pinned suffix 36
9.4 π_C3S: the disclosed usage is the provider's own 37
9.5 Why the node cannot use the proofs as a decryption oracle . . . 38
9.6 The proof channel 39

10 Pinning the request: the suffix and the hold 40
10.1 The suffix 40
10.2 Why the suffix decides 41
10.3 The hold 41

11 Property by property 43
11.1 P1: the client cannot steal the API key 43
11.2 P2: the node cannot read the prompt or the response 44
11.3 P3: the node cannot alter the request or the response 44
11.4 Residual risk: the node controls epoch 0 45
11.5 P4: the bill is the provider's own number 45
11.6 P5: both verify the provider 46
11.7 P6: freshness 46

12 Billing 47
12.1 The reservation: why a full context window 47
12.2 Settlement from the provider's usage event 48
12.3 Fees: pricing the node's own work 48
12.4 Upstream errors 49
12.5 What happens on abort, at each point 50

13 The receipt and the attestation 52

14 Denial-of-service limits 54

15 Costs and timings 56

16 Pre-warming 58

17 Images 60

18 What it does not hide, and other limits 62

19 The predecessors: v1 and dual execution 63

A Glossary: the paper's names and symbols **64**

B Where things are in the code **66**

How to read this

This book retells the protocol of the paper *Confidential and verifiable upstream inference over standard TLS 1.3* (protocol v2, the deployed one) for a programmer. There is no math notation. Every cryptographic tool is described by its contract, like an API: what goes in, what comes out, who learns what, what it guarantees. The description follows the code.

The cast:

- **C**, the client. In production that is routstrd with the routstr SDK (a reference client, `crypto/client/src/driver.ts`, does the same). C spawns two helper binaries: `cu-2pc` (its half of the two-party computation) and `cu-prover` (its half of the zero-knowledge proofs). They read their secrets on stdin, so no secret touches the disk.
- **N**, the node. Two processes: **routstr-core** (Python, FastAPI: the offer, the setup gate, billing, receipts; it holds the node's Nostr signing key and the money) and **cu-sidecar** (Rust: the TLS relay, N's half of the 2PC and of the proofs; it holds the provider API key). Neither holds both.
- **U**, the upstream provider (`api.venice.ai` in the case study). Unmodified; it sees an ordinary TLS 1.3 client.

Chapter 7 walks through one complete session with sequence diagrams. If you read one chapter, read that one. Chapters 8 to 10 open the three black boxes (the 2PC, the proofs, the request pinning). Chapters 11 onwards go property by property: security, billing, limits, costs.

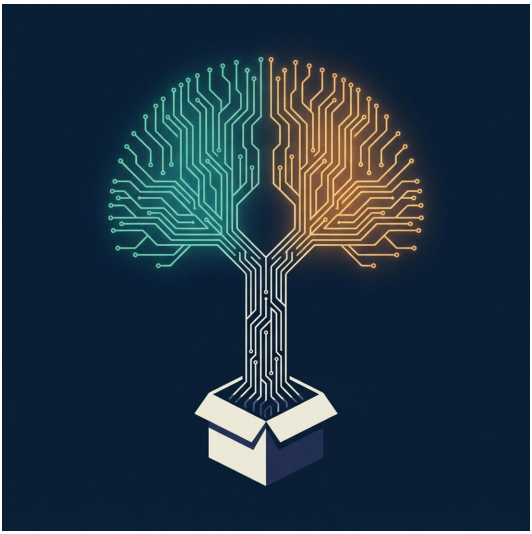
Two kinds of boxes recur:

> **Why this is safe**
> The argument for a property, in a few lines.

> **Attack: what a cheater could try**
> **Why it fails:** the mechanism that stops it.

1

The problem and the idea



1.1 What a routing node is today

A routstr node holds an account with an LLM provider. You pay the node (an `sk-` balance or an e-cash token); it forwards your chat request with its own API key, and pays the provider. It is a full man in the middle: it reads your prompt and the response, it could change either (swap the model, edit a tool call), and it reports the bill. The node, in turn, must never give you its API key.

Enclaves (TEEs) are the usual answer, but they move trust to a hardware vendor and to a software measurement every user would have to review after every

update. This protocol uses something users already rely on: the provider's TLS certificate.

1.2 The idea in one paragraph

The client is the TLS 1.3 client of the provider; the node only relays records. The node contributes exactly one TLS record to the stream, the HTTP request head with `Authorization: Bearer <api key>`, under a key only it has. Everything else (the JSON body with your prompt, the streamed response) is encrypted under keys only the client has. Three tools make that safe for both sides:

1. A **two-party computation (2PC) of the TLS key schedule**: client and node jointly act as one TLS client; the computation hands each party only its own keys. Nobody ever holds the ECDHE secret or the master secret.
2. **Zero-knowledge proofs** that pin what each side may write: the node proves its record is the published template (π_NK); the client proves its body ends with a public JSON suffix that fixes the model and the token cap (π_C2S), and that the usage it reports is the provider's own (π_C3S).
3. The node's **ordinary billing**: a worst-case reservation at setup, a charge from the provider's own usage event, a signed receipt the client checks.

1.3 The goals

ID	Property	In plain words
P1	Credential confidentiality	C never learns the API key, even if it records every byte N sends to U
P2	Content confidentiality	N never reads the prompt or the response; it sees ciphertext plus billing data (the usage event, or on an upstream error the response head)
P3	Integrity	U got C's body behind the agreed parameters; C reads exactly what U sent, tool calls included
P4	Verifiable accounting	the charge comes from the usage U reported in this session
P5	Upstream authenticity	both verify the certificate and hostname of U
P6	Freshness	every session is live, bound to one payment, never reusable

2

Threat model and trust

2.1 Who may cheat

C and N are each assumed **malicious towards the other**: either may send garbage, deviate from the protocol, abort at any moment, or collude with someone watching the network. A plain **network observer** between any two parties sees TLS or WebSocket traffic only.

2.2 What is trusted

- **The provider U.** It follows TLS 1.3 and its own API, and must fit the provider profile of chapter 5. It sees the prompt (it runs it) and the API key (it issued it), as it does today.
- **The provider's certificate and the WebPKI.** Both C and N verify the chain to the WebPKI roots and the hostname. C accepts only upstream hostnames on its own trusted list and only offers signed by a node key it pinned.
- **Your own machine** (C's process holds C's keys), and for the node its own processes (the sidecar holds the API key in memory).
- **Standard cryptography**: SHA-256, HMAC, AES-GCM, ECDH on P-256, and the oblivious-transfer, VOLE and garbling building blocks of TLSNotary's `mpz2` library.

Nothing else is trusted: the protocol uses no enclave or attested software, and it never relies on the operator behaving well.

2.3 What is out of scope

Hiding the prompt from U; proving which weights ran; hiding traffic volume and timing from N; and proofs a third party could check. The proofs are interactive and convince only the other party of the session, which is what settlement needs.

3

TLS 1.3, only the parts we need

3.1 The key schedule is a tree

After the key exchange both TLS peers hold a shared ECDHE secret. TLS derives a tree of secrets from it. Every arrow in figure 3.1 is HMAC-SHA-256 (HKDF): one-way. From a child you cannot go back to its parent, and from one child you cannot compute a sibling.

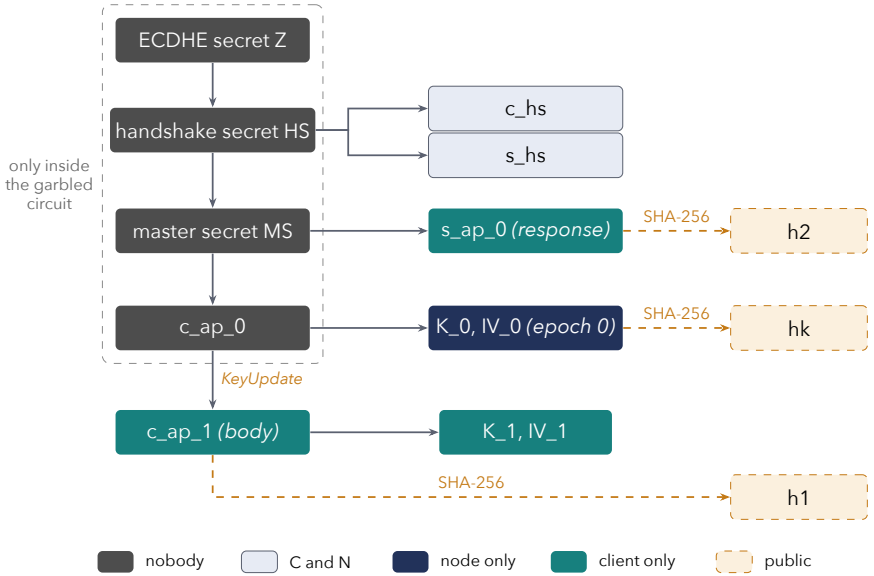


Figure 3.1: The TLS 1.3 key schedule, colored by who ends up holding each secret.

The parts the protocol uses:

- Each traffic secret yields an AES-128 key and a 12-byte IV (labels “key” and “iv”).
- A record with sequence number `n` uses the nonce `IV xor n`. AES-GCM encrypts it: CTR mode starting at counter 2, plus a GHASH tag. Sequence numbers restart at 0 in each epoch.
- **KeyUpdate** lets a sender rotate its own sending key at any record boundary (`c_ap_0` to `c_ap_1`). With `update_not_requested` only that direction changes; the server keeps sending under `s_ap_0`.
- Two transcript hashes enter the derivations: `th_SH` (ClientHello and ServerHello) and `th_SF` (everything through the server Finished).

3.2 The observation the protocol is built on

`K_0` (the record key of epoch 0) and `c_ap_1` (the client secret of epoch 1) are siblings under `c_ap_0`. Knowing `c_ap_1` and `s_ap_0` tells you nothing about `K_0`, and knowing `K_0` tells you nothing about `c_ap_1` or `s_ap_0`. So if a trusted machine computed the tree and gave each party only its own branch, the node could write epoch 0 and the client could write epoch 1 and read the response, and neither could read or forge the other’s part. There is no trusted machine, so the parties emulate it with a two-party computation.

4

The split: who gets which branch

4.1 Who holds which secret

Secret	C	N	U	Used for
ECDHE scalar	own share	own share	own	key exchange
ECDHE secret Z	additive share	additive share	yes	input to the 2PC only
HS, MS, c_ap_0	**never**	**never**	yes	exist only inside the garbled circuit
c_hs, s_hs	yes	yes	yes	both verify the handshake (P5)
K_0, IV_0	**never**	yes	yes	N's credential record (P1)
c_ap_1	yes	**never**	yes	C's request body (P2)
s_ap_0	yes	**never**	yes	the response (P2)
h1 = SHA256(c_ap_1)	yes	yes	no	public anchor for π_C2S
h2 = SHA256(s_ap_0)	yes	yes	no	public anchor for π_C3S
hk = SHA256(K_0, IV_0)	yes	yes	no	public anchor for π_NK
the API key	**never**	yes	yes	only inside N's epoch-0 record

The three **anchors** are hashes of 32 random-looking bytes (28 for `hk`), so publishing them reveals nothing. They let a later proof say "the secret I use is the one the 2PC gave me" without recomputing the key schedule inside the proof. Each party also recomputes its own anchors from the secrets it received, and aborts on a mismatch.

4.2 Epochs on the wire

The client-to-server direction is cut into epochs by one KeyUpdate. The node owns epoch 0, the client owns epoch 1.

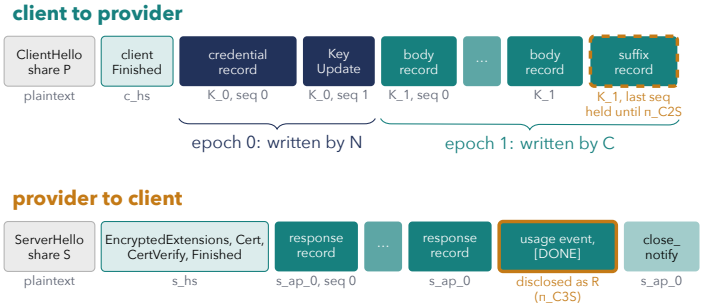


Figure 4.1: The records of one session as U sees them. The node writes only the two navy records.

Epoch	Key	Who writes	What
handshake	c_hs / s_hs	C (U answers)	ClientHello with the joint key share; client Finished, computed by C from c_hs
0	c_ap_0 (K_0, IV_0)	**N**	seq 0: the credential record (request head with `Authorization`); seq 1: `KeyUpdate`
1	c_ap_1	C	the JSON body from seq 0; its last record is the public suffix, released to U only after π_C2S
response	s_ap_0	U	the streamed response; N relays ciphertext

C writes nothing in epoch 0. Its TLS stack (a fork of an MIT-licensed TLS 1.3 client with an "external" hook) never computes a key itself: after the handshake the 2PC injects `c_ap_1` as its sending secret (starting at sequence 0) and `s_ap_0` as its receiving secret. U sees an ordinary client that sent one record, rotated keys, and continued.

Why the node's record and the client's records need different keys

With one shared key, whoever holds it can decrypt the other's records (the client would read the API key, the node the prompt) and forge records in the other's name: AES-GCM gives no separation between two writers of one key. KeyUpdate is the only standard TLS 1.3 way to switch the client's sending key mid-connection without the server noticing anything unusual, so it serves as the hand-over point. The response direction never rotates, and only C holds its key.

5

What the provider must do

The protocol works with any provider that satisfies this profile. For Venice, items 1 to 7 are checked live by `crypto/tools/provider-profile-check.py` (checks A1 to A11, all passing on 2026-10-10; rate limits and timeouts count as inconclusive, never as a pass).

#	Requirement	Why
1	TLS 1.3, `TLS_AES_128_GCM_SHA256`, ECDHE on P-256; no HelloRetryRequest, no 0-RTT	the 2PC computes exactly this key schedule; the joint key share needs point addition, which X25519 lacks
2	HTTP/1.1, `Connection: close`, `Accept-Encoding: identity`	one request per connection, records map to bytes, no compression around the usage event
3	accepts a client KeyUpdate in the middle of a request	that is the hand-over from N to C
4	does nothing until the whole `Content-Length` body has arrived	N can hold the last body record until π_C2S verifies; nothing runs or costs before
5	JSON: last duplicate key wins; a later object replaces an earlier one; malformed JSON and unknown fields are rejected	the public suffix overrides anything in the private part, and the private part cannot leave a string or object open for the suffix to land in
6	every cost-relevant parameter can be set by the suffix (model, caps including reasoning tokens, `n`, price-changing flags); other price-changing fields are rejected	the reservation and the bill depend on them

#	Requirement	Why
7	a streaming usage event with prompt and completion token counts before the stream ends	that is what C discloses and N bills
8	never reflects the request head (for example `Authorization`) into the response	C reads the response; a reflection would hand it the key (not tested by the gate)

Items 5 and 6 quantify over every possible field, which no finite test proves: the gate samples them and must be re-run after provider changes. A provider change that breaks pinning or usage reporting makes sessions fail; it never makes them fall back to an unverified path.

For Venice the pinned suffix also sets `max_completion_tokens` (it overrides `max_tokens`, a bypass found in testing) and `venice_parameters` with the injected system prompt and web search switched off (both change the bill).

6

The components

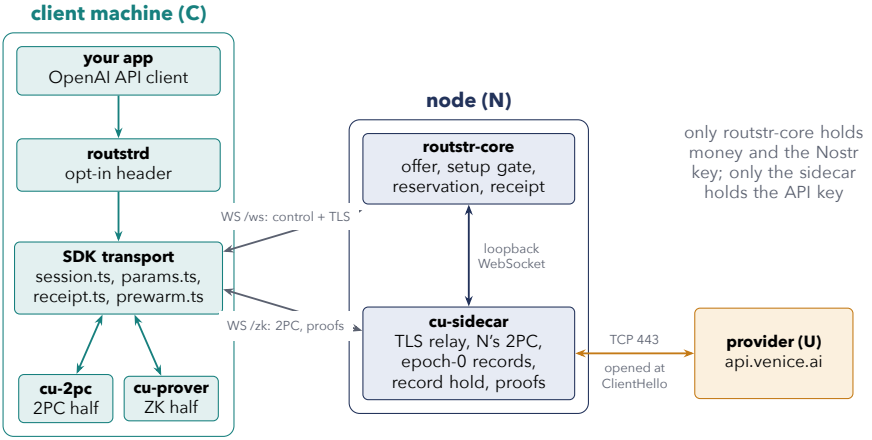


Figure 6.1: Processes and channels. routstr-core copies the 2PC and proof bytes verbatim; it intercepts only a few control frames.

The channels are WebSockets through routstr-core, all under `/v1/confidential/`:

Path	Carries
`ws`	control frames (JSON) and the TLS records (binary); query `session_id` (a UUID) and `v=2`
`zk?proof=twopc`	the 2PC protocol bytes; query `session_id` is the protocol `sid`
`zk?proof=zkchan`	the pre-warmed proof channel for π_NK, π_C2S and π_C3S
`zk?proof=pi_nk`	fallback: one connection per proof (likewise `pi_c2s`, `pi_c3s`)

The sidecar opens its TCP connection to U on port 443 only when C's ClientHello arrives.

Opting in: routstrd forwards a request confidentially when it carries the header `x-routstr-verify: confidential`. Only `POST /v1/chat/completions` is supported. A request that opted in **fails closed**: it is never silently sent on the plain path. The SDK routes it only to nodes whose `/v1/models` entry for the model carries a `confidential_upstream` field and whose offer verifies.

7

One session, step by step

This chapter follows one request from the offer to the receipt. Arrows between C and the sidecar pass through routstr-core, which copies them unchanged; the diagrams draw them straight to show who really talks to whom. Teal is the client, navy the node, amber the provider.

7.1 Step 0: the offer

The node publishes a signed offer at `GET /v1/confidential/offer`. routstr-core builds it from the sidecar's static description (`GET /offer` on the sidecar: upstream host, head template, key length and alphabet, suffix keys, token cap) plus its own per-model rates, and signs it with its Nostr key (BIP-340 Schnorr over the SHA-256 of canonical JSON: sorted keys, no whitespace). A made-up but realistic example:

```
{
  "v": 2, "min_v": 2, "max_v": 2,
  "upstream_host": "api.venice.ai",
  "ws": "wss://node.example/v1/confidential/ws",
  "notary_pubkey": "7e4f...c1a2",
  "notary_pubkey_curve": "secp256k1-schnorr",
  "max_tokens_cap": 4096,
  "key_length": 42,
  "key_alphabet": "[A-Za-z0-9_-]",
  "head_template": "POST /api/v1/chat/completions HTTP/1.1\r\n
    Host: api.venice.ai\r\nAuthorization: Bearer {KEY:42}\r\n
    Content-Type: application/json\r\nAccept-Encoding: identity
    \r\nConnection: close\r\nContent-Length: {LEN}\r\n\r\n",
  "suffix_keys": ["model", "max_tokens", "max_completion_tokens",
    "n", "stream", "stream_options", "venice_parameters"],
  "price_list": {
    "deepseek-v4-flash": {"in": 21.7, "cached_in": 5.4,
                          "out": 160.0}
```

```
  },
  "mints": ["https://mint.example"],
  "abandon_fee_msat": 2000,
  "sig_payload": "{\"abandon_fee_msat\":2000,...}",
  "sig": "c03b...9f1e"
}
```

(The template is one string; it is wrapped here for the page.) Rates are millisatoshis per 1,000 tokens. `{KEY:42}` marks the 42 bytes of the API key, `{LEN}` the decimal body length. Fees appear only when the node charges them. The client:

1. takes every field from `sig_payload`, the exact signed bytes, never from the unsigned envelope;
2. requires the signature to verify **with a node key it pinned** in its configuration (not just the key the offer names);
3. requires `upstream_host` to match its trusted-host list (an empty list refuses every host);
4. requires a supported version and a prover binary.

`/v1/models` gains one additive field per model, `confidential_upstream: {v, offer_url, ...}`, only for models the node can price.

7.2 Step 1: setup and reservation

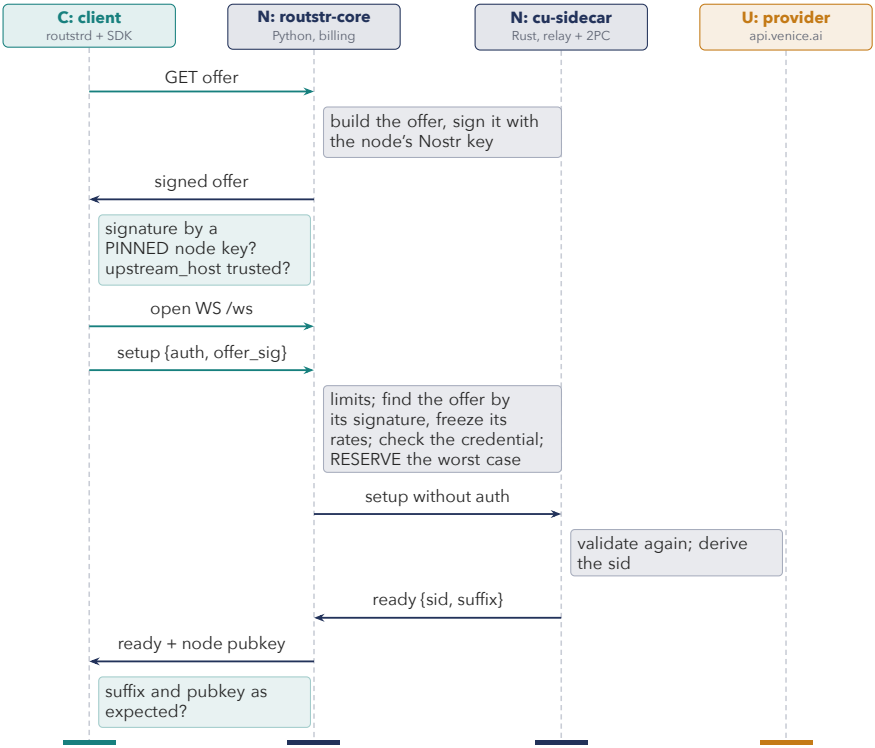


Figure 7.1: Setup: nothing is reserved before the credential checks, and the sidecar never sees the credential.

The client opens the session WebSocket and sends:

```
{
  "type": "setup",
  "nonce_c": "3f9a...(32 bytes hex)...07d1",
  "model": "deepseek-v4-flash",
  "max_tokens": 40,
  "len": 4096,
  "armed": true,
  "offer_sig": "c03b...9f1e",
  "abandon_fee_msat": 2000,
  "auth": "sk-...(or a cashuB... token)"
}
```

`len` is the exact length of the JSON body in bytes, fixed now. routstr-core checks the shape (`max_tokens` an integer from 1 to the cap, `len` from 1 to 8 MiB), looks the offer up by `offer_sig` among the offers it signed in the last ten minutes and freezes that offer's rates for this session (unknown or expired: HTTP 409 `offer_expired`, and the SDK refetches once), requires the client to echo every fee the offer charges (else 409, so a client that does not know about fees is never charged one), validates the credential with the node's ordinary `validate_bearer_key`, and **reserves the worst case** with `pay_for_request` (chapter 12). Only then does it dial the sidecar and forward the setup **without** `auth`, `offer_sig` and the fee fields, adding the reservation's release id as `escrow`.

The sidecar validates the setup again and derives the session id:

```
sid = SHA256(nonce_c || nonce_n || SHA256(reservation_release_id))
```

`ready` comes back with `sid`, `nonce_n`, `head_len`, the canonical `suffix` and `v: 2`. routstr-core replaces `pubkey` with its own Nostr key. The client checks that the suffix equals the one it computed itself and that `pubkey` equals the key of the signed offer.

7.3 Step 2: preprocessing and the joint key share (arming)

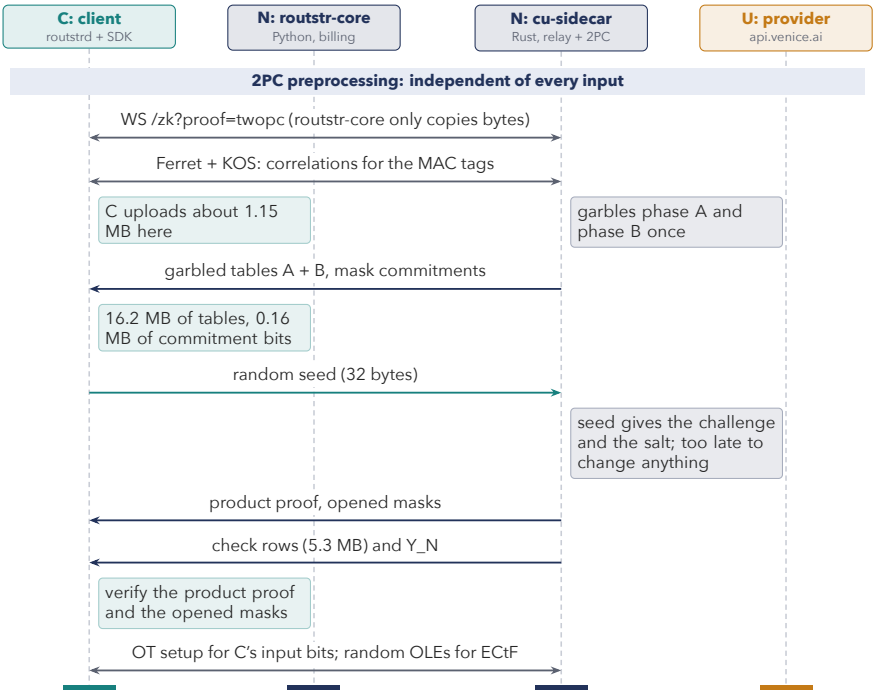


Figure 7.2: Preprocessing: the expensive, input-independent part of the 2PC. Every byte of it can happen before a request exists.

The client spawns `cu-2pc` and connects it to `/zk?proof=twopc`. The node garbles both circuits, streams the tables, authenticates its wire masks to the client, and runs the checks that do not depend on any input (chapter 8 explains each message). This moves about 22 MB from N to C and 1.2 MB from C to N.

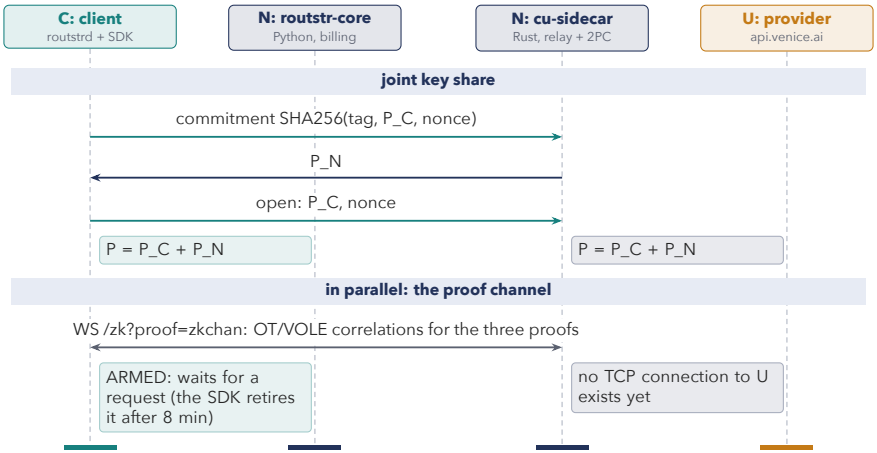


Figure 7.3: The joint key share (commit, then open) and the proof channel.

The **joint key share** is the TLS key share both parties will offer together:

```
# C                                   # N
a_C = random_scalar()                 a_N = random_scalar()
P_C = a_C * G
send(SHA256(tag | P_C | nonce))  -->
                                 <--  send(P_N = a_N * G)
send(P_C, nonce)                 -->  check the commitment
P = P_C + P_N                         P = P_C + P_N
```

The commitment comes first so that C cannot pick its point after seeing `P_N`. (If it could, it would choose `P_C` as "a point whose secret I know, minus `P_N`", and the joint point would be a point C alone controls.) The client also opens `/zk?proof=zkchan`, a long-lived proof channel that prepares the random correlations for all three proofs of the session.

A session in this state is **armed**. With pre-warming (chapter 16) the SDK arms sessions before requests exist; a cold request does all of this on the request path.

7.4 Step 3: the TLS handshake with two online 2PC phases

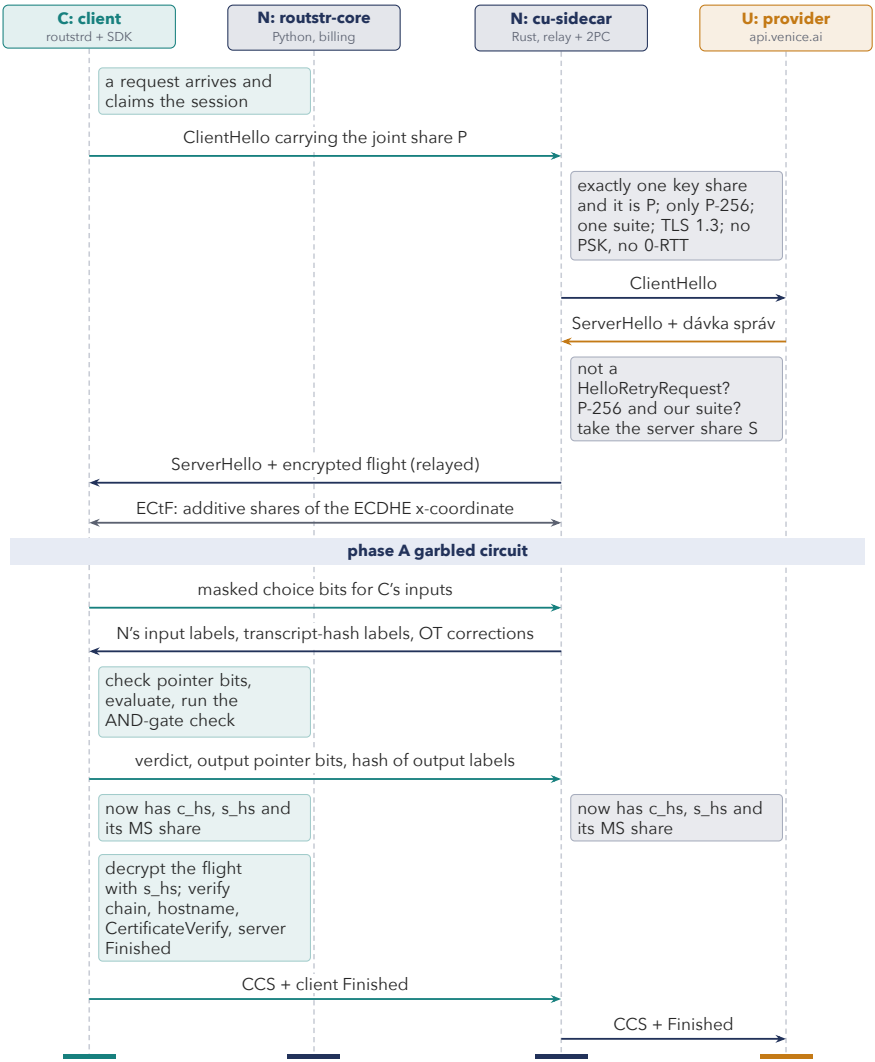


Figure 7.4: The handshake. The sidecar relays the ClientHello only if its single key share is the joint point.

1. C's TLS fork builds a ClientHello with exactly one key share: the joint point `P`.
2. **The sidecar parses it and relays it only if** the only cipher suite is `TLS_AES_128_GCM_SHA256`, the only version TLS 1.3, `supported_groups` lists only P-256, the key share list has exactly one entry and it equals `P`, compression is null, there is no PSK and no early data, and no extension is duplicated. Only now does it open TCP to U.
3. U answers with ServerHello and its encrypted flight. The sidecar checks the ServerHello: not a HelloRetryRequest, TLS 1.3, the offered suite, a P-256 share. It extracts U's share `S`.
4. **ECtF** turns `a_C` times `S` and `a_N` times `S` (two halves of the shared point) into additive shares of its x-coordinate, the ECDHE secret, without either party learning it.
5. **Phase A** of the garbled circuit computes the handshake secrets and the master secret. Both get `c_hs` and `s_hs`; the master secret leaves split in two random-looking shares.
6. C decrypts U's flight with `s_hs` and verifies the certificate chain to the WebPKI roots, the hostname, CertificateVerify and the server Finished; it sends its client Finished, computed from `c_hs`. The sidecar relays it.

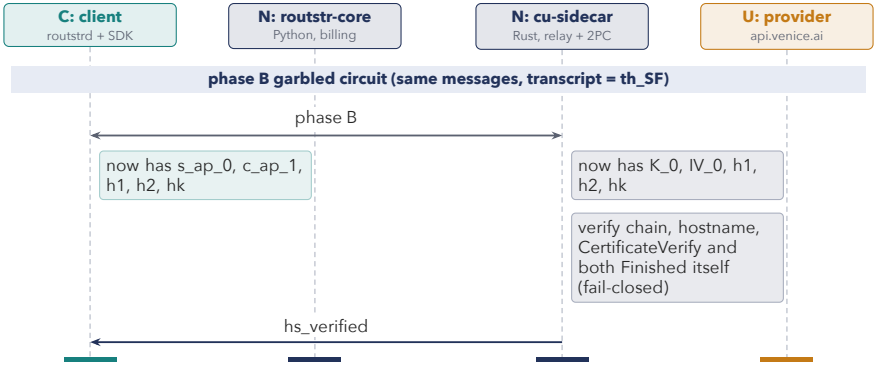


Figure 7.5: Phase B gives each party its own application keys; the sidecar then verifies the whole handshake itself.

7. **Phase B** (input: the master-secret shares and `th_SF`) gives C `s_ap_0` and `c_ap_1`, gives N `K_0` and `IV_0`, and gives both the anchors `h1`, `h2`, `hk`.
8. The sidecar runs its own fail-closed verification with `c_hs` and `s_hs`: it decrypts the flight, rebuilds the transcript, verifies the chain to the Mozilla roots, the hostname, CertificateVerify and **both** Finished messages. It also checks that C sent nothing but ClientHello, CCS and Finished so far. Then it sends `hs_verified`.

U must receive the client Finished within its handshake timeout (a CDN closes an idle handshake after about 15 s); phase A takes about 33 ms locally, so this is not tight.

7.5 Step 4: epoch 0, the credential record

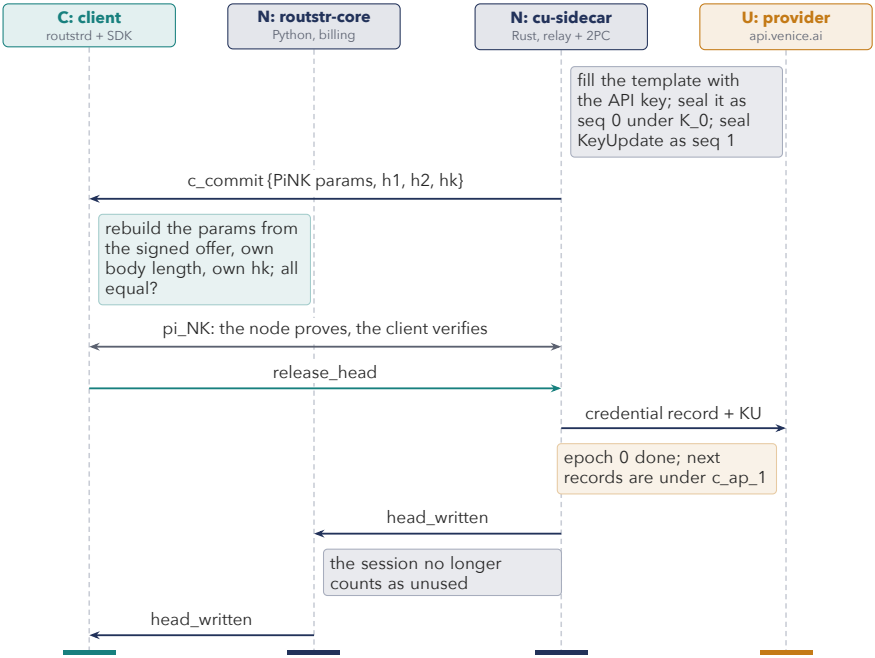


Figure 7.6: Epoch 0. The node commits to its record, proves it matches the template, and writes it only when the client says go.

The sidecar fills the published template with the API key and the body length, and seals two records under `K_0`: sequence 0, the head (inner content type 0x17, application data), and sequence 1, `KeyUpdate(update_not_requested)`. It sends the client a commitment to the first record and the public parameters of π_NK:

```
{
  "type": "c_commit",
  "c_commit": "SHA256(header || ciphertext), hex",
  "h1": "...", "h2": "...", "hk": "...",
  "params": {"PiNK": {"hk": [...], "head_len": 229,
```

```
    "key_len": 42, "key_alphabet": "[A-Za-z0-9_-]",
    "key_offsets": [83, 84, ...], "template": [...],
    "c_commit": [...], "header": [23, 3, 3, 0, 246]}}
}
```

The client rebuilds every one of those parameters from the **signed offer**, its own body length and its own `hk`, and refuses any difference (pseudocode in chapter 9). Then it verifies π_NK and sends `release_head`. The sidecar writes the two records to U in one write and answers `head_written`. From here the session no longer counts as "unused" for the per-address limit.

7.6 Step 5: the body and the record hold

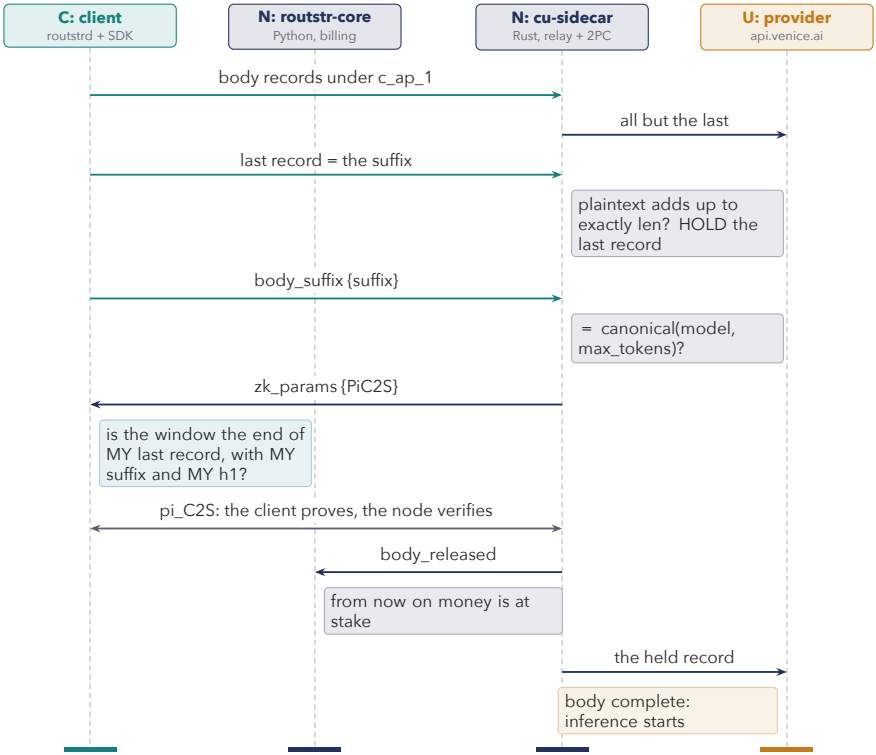


Figure 7.7: Epoch 1. The record that completes the body is held until the suffix proof verifies.

C writes the body under `c_ap_1` from sequence 0: first the private JSON part (possibly several records), then **the pinned suffix as its own final record**. The sidecar counts plaintext bytes (record length minus 16 bytes of tag minus 1 content-type byte); more than `len` is an error, and the record that makes the total exactly `len` is **held**. U now has an incomplete request and waits (profile item 4).

C sends `body_suffix`; the sidecar requires it to equal `canonical_suffix(model, max_tokens)` from the setup, builds the π_C2S parameters from the held record, and the client checks them and proves. On success the sidecar tells routstr-core `body_released` **before** writing the held record, so the node's billing knows the provider may now do work. If the proof fails, the connection closes: no inference ran.

7.7 Step 6: the response

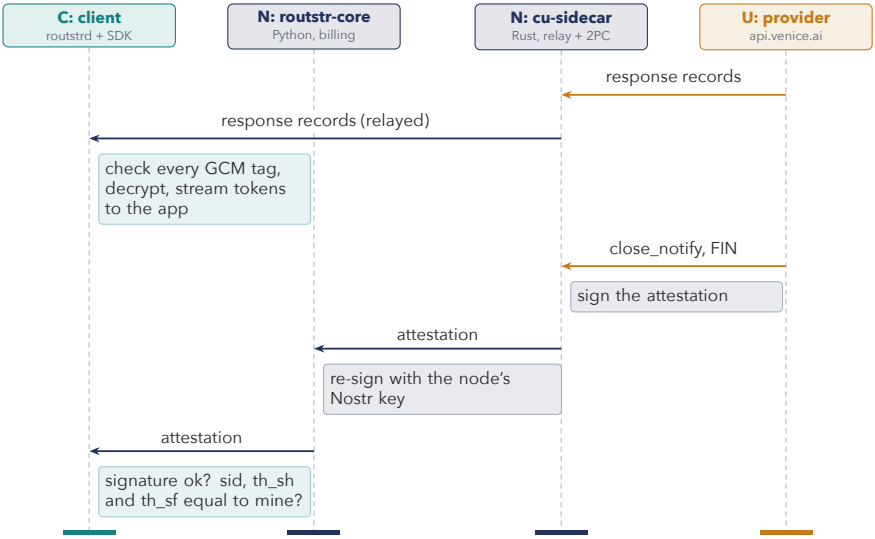


Figure 7.8: The response streams to the client; the node attests what it relayed.

U streams the response under `s_ap_0`. The sidecar relays the records; C checks every GCM tag, decrypts, and streams tokens to the application as they arrive. When U closes, the sidecar signs an attestation of what it relayed and routstr-core re-signs it with the node's Nostr key:

```
{
  "type": "attestation",
  "a": {
    "sid": "5b0e...a4",
    "cert_chain_hash": "SHA256 of the DER chain",
    "th_sh": "...", "th_sf": "...",
    "sha256_sent": "SHA256 of every byte sent to U",
    "sha256_recv": "SHA256 of every byte from U",
    "client_records": 4, "server_records": 41,
    "c_commit": "...", "time": 1791661234
  },
  "a_json": "{canonical JSON of a}",
  "sidecar_sig": "ed25519 signature (sidecar key)",
  "sig": "BIP-340 signature (node key)",
  "sig_scheme": "schnorr-secp256k1"
}
```

C requires the node signature over `a_json` and that `sid`, `th_sh` and `th_sf` equal its own.

7.8 Step 7: disclosure and settlement

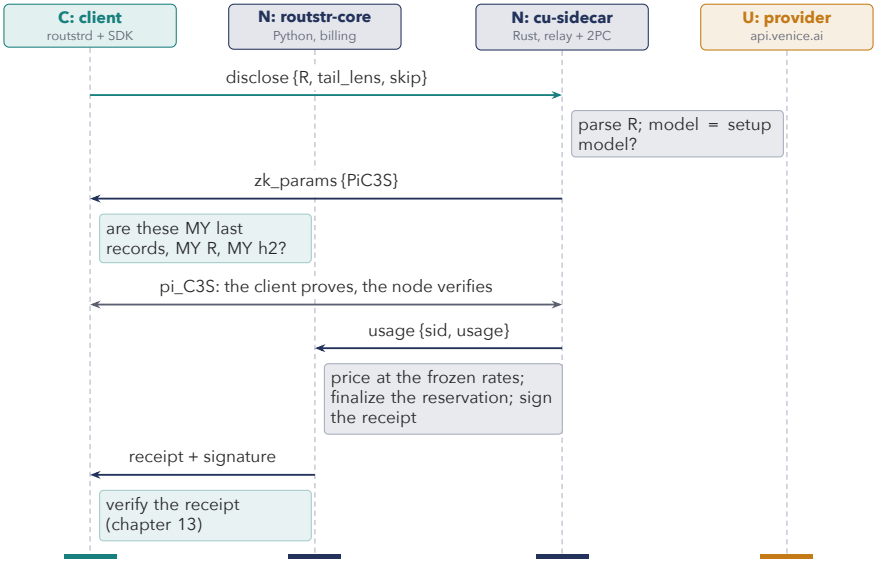
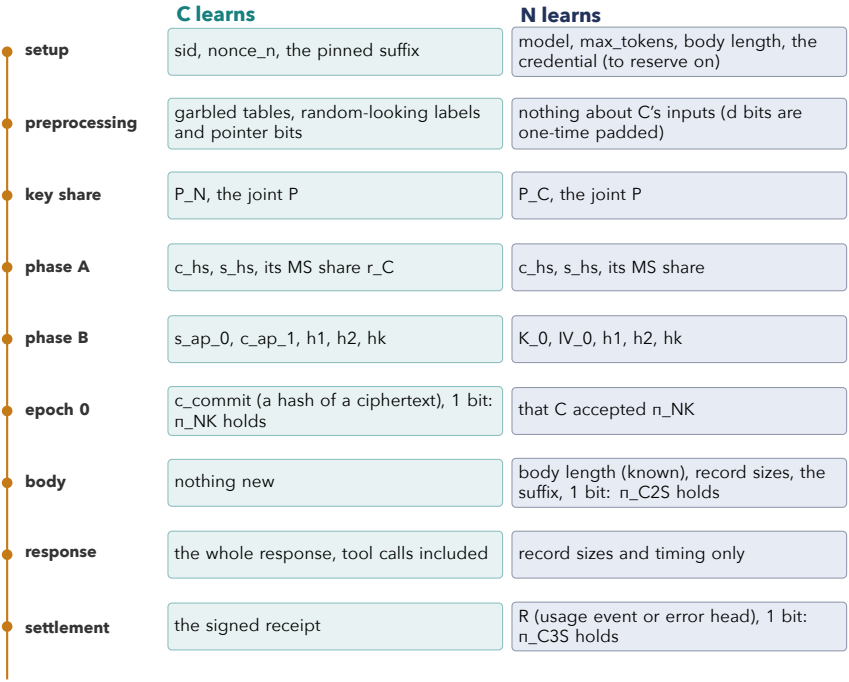


Figure 7.9: Disclosure, the usage proof and settlement.

C picks **R**: the plaintext from the start of the last usage-carrying event through the end of the stream, including the HTTP chunk framing (or, for an HTTP error, only the response head). It sends `disclose {R, tail_lens, skip}`. The sidecar parses R (dechunk; the last `data:` event before `data: [DONE]` must be JSON with empty `choices` and both token counts; its `model` must equal the setup model) and returns the π_C3S parameters: the last server records, located from the end of the transcript. C checks they are exactly its own last records and proves. The sidecar forwards the verified usage to routstr-core, which prices it at the frozen rates, finalizes the reservation and returns a signed receipt (chapter 13).

7.9 Who knows what, step by step

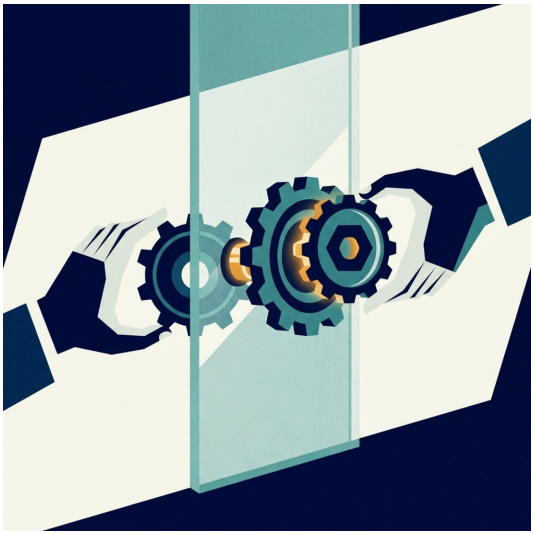


Never learned by anyone: the ECDHE secret, HS, MS, c_ap_0. Never learned by C: K_0, IV_0, the API key. Never learned by N: c_ap_1, s_ap_0, the prompt, the response.

Figure 7.10: What each party learns at each step. U sees an ordinary TLS session with everything in it.

8

The two-party computation



Goal: compute the TLS key schedule of figure 3.1 from inputs split between C and N, and deliver each output only to the party it belongs to, against a peer that cheats. The 2PC covers only the key schedule: 31 SHA-256 compressions in total, never the payload. That is why its cost is constant and can be paid before a request exists.

8.1 The building blocks, as APIs

Oblivious transfer (OT)

In: the sender has two messages `m0`, `m1`; the receiver has a choice bit `b`. **Out:** the receiver gets `m_b`. **Who learns what:** the sender learns nothing about `b`; the receiver learns nothing about the other message. **Used for:** handing C the circuit labels for its own input bits without N learning those bits.

Correlated OT, KOS and Ferret: pre-made randomness

Out: a large batch of random pairs with a fixed hidden relationship: one party holds a secret global offset (call it `Delta`) and, for each item, a random key `K`; the other holds a random bit `x` and `M = K xor (x ? Delta : 0)`. **Cost:** a handful of expensive base OTs, then KOS and Ferret stretch them into millions of correlations cheaply. **Used for:** OT for C's input bits, and the MAC tags below.

IT-MAC: a tag that catches a lying node

Setup: C holds a secret 128-bit `Delta_B` (127 random bits) and one key `K` per tagged bit. N holds the bit `x` and its tag `M`, where `M` equals `K`, XORed with `Delta_B` if `x` is 1. **Guarantee:** N can later "open" `x` (send `x` and `M`), and C checks the tag. To open a false value N would have to know `Delta_B`, which it never sees. **Bonus:** tags are linear: the XOR of tagged bits is tagged by the XOR of their tags, for free.

Commitment: a sealed envelope

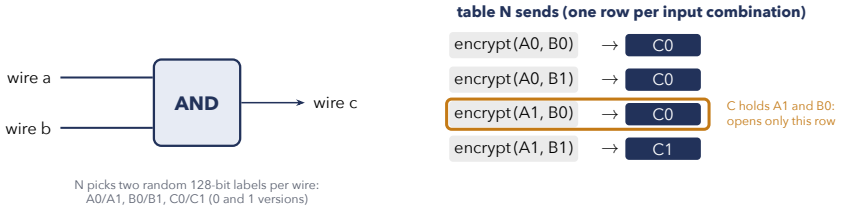
In: a value and a random nonce. **Out:** `SHA256(tag | value | nonce)`. You cannot change the value after sending the hash, and the hash reveals nothing about it until you open it (send the value and nonce).

OLE and share conversion (ECtF)

OLE in: one party has `a`, the other `b`. **Out:** two random-looking numbers that add up to `a` times `b`, one per party. **A2M / M2A** use OLEs to convert "a sum split across two parties" into "a product split across two parties" and back. **ECtF** (EC to field) chains two A2M and one M2A over the P-256 base field: C holds `a_C` times `S`, N holds `a_N` times `S` (two points whose sum is the ECDHE point), and they end with numbers `x_C` and `x_N` that add up (mod the field prime) to the x-coordinate of the sum, which is the ECDHE secret. Neither learns it. It is the same conversion TLSNotary uses; it is semi-honest (a cheater can only make the result wrong, which the TLS Finished check catches).

Garbled circuit: an encrypted lookup-table program

Garbler (N) in: a Boolean circuit (public), a secret global offset `Delta_A`. **Garbler out:** for each wire two random 128-bit labels (one meaning 0, one meaning 1, differing by `Delta_A`), and for each AND gate a small encrypted table. **Evaluator (C) in:** one label per input wire. **Evaluator out:** one label per output wire, plus the decoding bits that turn output labels into values. **Who learns what:** C learns the outputs it is allowed to decode and nothing about intermediate values. **Caveat:** plain garbling trusts the garbler to garble the right circuit.



C gets one label per input wire and opens exactly one row per gate, obtaining one label per output wire. A label looks random; C cannot tell whether it means 0 or 1. Its last bit (the **pointer bit**) is the true value XOR a secret per-wire **mask** that only N knows; it tells C which row to open. Real tables are compressed (three-halves garbling: about 197 bits per AND gate), and XOR gates need no table at all.

Figure 8.1: One garbled AND gate.

8.2 What the circuits compute

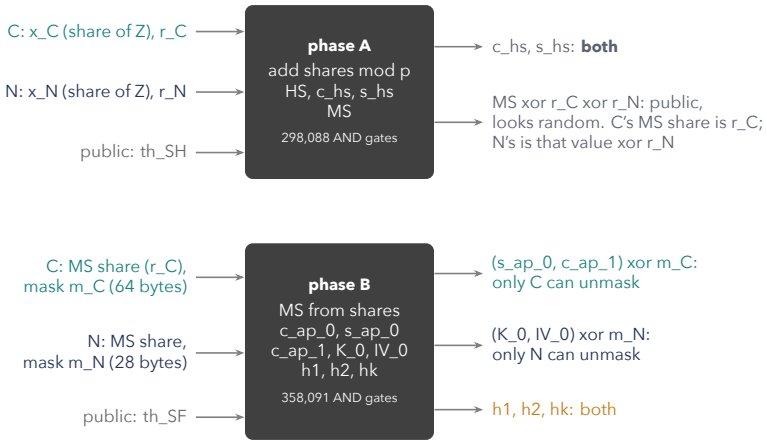


Figure 8.2: The two circuits. Teal inputs come from C, navy from N, gray are public; outputs are colored by who can read them.

Phase A runs once U's ServerHello is known. It adds the two shares of the ECDHE secret modulo the P-256 prime (about 770 AND gates), then computes HS, the keyed HMAC states for HS, `c_hs`, `s_hs`, the "derived" secret and MS. `c_hs` and `s_hs` come out in the clear (both need them). MS comes out XORed with a random mask from each party, `r_C` and `r_N`. Afterwards C's share of MS is `r_C` and N's share is "the public output XOR `r_N`". Each share alone is random noise.

Phase B runs after the server Finished. It recombines MS from the two shares and computes `c_ap_0`, `s_ap_0`, `c_ap_1`, `K_0`, `IV_0` and the three anchors. Private outputs leave the circuit XORed with a mask their owner fed in: `s_ap_0` and `c_ap_1` under C's 64-byte mask `m_C`, `K_0` and `IV_0` under N's 28-byte mask `m_N`. Each party may see the other's masked output; it is one-time padded. The anchors come out in the clear.

Every HKDF-Expand-Label fits one SHA-256 block and keyed HMAC states are computed once per secret, so phase A has 14 compressions (298,088 AND gates) and phase B 17 (358,091 AND gates): 656,179 AND gates per session.

8.3 How C makes sure the node garbled honestly

A malicious garbler could garble a different circuit, for example one that copies `c_ap_1` into N's output. The deployed construction is **node-only authenticated garbling**: only N garbles, C evaluates and checks every AND gate, using tags on N's secret masks.

The mask and the pointer bit. Every wire has a secret mask bit chosen by N (the last bit of N's 0-label). The label C holds for a wire ends in a **pointer bit**: the wire's true value XOR that mask. C sees pointer bits, never masks, so pointer bits look random to it.

Preprocessing, in order (the code is `preprocess_n / preprocess_c` in `crypto/twopc/src/lib.rs`):

1. Correlated OT with **C as the sender** (Ferret over KOS; C holds `Delta_B`). With it, N tags (IT-MACs) the mask of every circuit input and every AND output ("base wires") and, for every AND gate, the product of its two input masks: about 1.3 million tagged bits. Masks of other wires follow by linearity. N sends one correction bit per tagged value.
2. N garbles phase A and phase B with three-halves garbling and streams the tables (16.2 MB).
3. **Only after all of that has arrived**, C sends a 32-byte random seed. From it both derive the random challenge for the checks and a per-session salt. N can no longer change anything.

4. N proves all the products at once (a QuickSilver-style batched check: two field elements).
5. N opens, with their tags, the masks of C's input wires, of the transcript-hash wires and of every output wire. C needs these to check its inputs and decode outputs.
6. N sends one **check row** per base wire (truncated to 64 bits) and a value `Y_N`.
7. KOS OT with N as sender sets up the transfer of C's 1,280 input bits; random OLEs for ECtF.

Online, per phase (about 1.5 round trips, 35 to 45 ms of CPU):

```
# C, one phase (simplified from gc::run_check and phase_c)
send(choice_bits xor ot_randomness)        # N learns nothing
labels_N, labels_th, corrections = recv()  # N's inputs, th, OT
labels_C = derive_my_labels(corrections)
assert pointer_bits(labels_C)  == my_bits  xor opened_masks_C
assert pointer_bits(labels_th) == th_bits  xor opened_masks_th
active = evaluate(tables, labels_C, labels_N, labels_th)
# the AND-gate check: one equation for all gates
y_C = xor_all(hash(salt, lbl) xor (lsb(lbl) and row)
              for lbl, row in zip(active.base_wires, check_rows))
if low64(Y_N xor y_C xor my_mac_key_total) != 0:
    send(ABORT); raise Abort
outputs = decode(active.outputs, opened_masks_out)
send(OK, active.output_pointer_bits, SHA256(active.out_labels))
```

Why the check works, intuitively. For an AND gate evaluated correctly, the pointer bits C saw, the committed masks and the committed mask product satisfy a fixed XOR relation. C cannot see the masks, but because tags are linear it can compute what the tag of a random combination of all those relations must be, from its keys alone. N must supply the matching value, but N does not know which pointer bits C saw, so it cannot compute it directly. Instead each check row is "padded" with hashes of both labels of its wire, arranged so that only the label C actually holds unlocks the correct contribution. If even one gate was evaluated against something other than what N committed, the combined value is off by a quantity scaled by C's secret `Delta_B`; N fixed everything before learning the challenge and knows nothing about `Delta_B`, so it must hit a 64-bit value blind: about one chance in 9 billion billion per phase.

Why the salt. The rows travel as 64 bits to save 5.3 MB. A node that reused labels across sessions could, with about 2 to the 64th hash evaluations done once, prepare a fake label whose truncated hash difference vanishes, and use it to leak which path C's evaluation took without causing an abort. The row hash

is keyed by a salt derived from C's seed, which N learns only after committing its tables, its offset and its keys, so that precomputation is impossible. An honest node never reuses labels anyway; the salt makes the check sound against one that does.

Output delivery. C decodes its outputs with the opened, tagged masks. Only then does it send N the output pointer bits and a hash of its output labels. N recomputes the labels from its own keys, checks the hash and decodes its outputs. C cannot change N's outputs: that needs an "inactive" label (the one C did not get), and those need `Delta_A`. **No output is sent to U, used to encrypt, or revealed before these checks pass.**

> **Why this is safe for the client**
>
> Every mask and mask product is fixed under C's secret key before the challenge exists. Any inconsistent gate fails the check except with probability of about one in 2 to the 63rd per phase. C's own input labels and the transcript-hash labels are pinned by pointer checks against opened, tagged masks, so N can neither swap C's inputs nor feed a different transcript. Once the check passes, C's outputs are the correct function of its inputs, N's (effective) inputs and the real transcript.

> **Why this is safe for the node**
>
> C is the evaluator: it sees tables, its own active labels, random-looking pointer bits, masks of values it already knows or decodes anyway, check rows padded with hashes of labels it does not hold, and `Y_N`, which it could compute itself. N's private outputs reach C only XORed with `m_N`. The API key is not an input to the 2PC at all.

8.4 Preprocessing versus online

Step	When	C sends	C receives	Time (local)
preprocessing (tables, tags, checks, OT setup, OLEs)	before the ClientHello; can be pre-warmed	1.21 MB	22.07 MB	0.55 s in-process, 0.70 to 0.84 s live
joint key share	pre-warmed too	41 B	74 B	2 ms
ECtF	after ServerHello	0.2 kB	12 kB	5 ms
phase A	after ServerHello	73 B	8 kB	about 33 ms
phase B	after server Finished	238 B	24 kB	about 40 ms

Of the 22 MB C downloads, 16.2 MB are garbled tables and 5.3 MB check rows; of the 1.21 MB it uploads, 1.15 MB is its part of Ferret (C is the Ferret sender). At 80 ms round-trip time the online part takes about half a second: key share 84 ms, ECtF 170 ms, phase A 121 ms, phase B 126 ms. A garbled circuit is single-use, so preprocessing can never be shared between requests; pre-warming only moves it off the request path.

8.5 The leakage: at most 2.3 bits per session

A cheating node can do one thing the checks do not prevent: make an abort **depend on C's secret values**. Example: corrupt one table row that C opens only if a certain secret bit is 1. If C aborts, N learns that bit. This is called selective failure.

How much can leak? In each phase N observes one of three outcomes: C aborted in its check; C's output report was rejected (an outcome N can engineer); success. The session ends at the first failure, so the whole session has at most five distinguishable outcomes:

```
phase A: C aborts                     -> outcome 1
phase A: report rejected              -> outcome 2
phase A ok, phase B: C aborts         -> outcome 3
phase A ok, phase B: report rejected  -> outcome 4
both phases ok                        -> outcome 5
```

Five outcomes carry at most log2(5), about 2.3 bits.

Why that does not add up to anything. The bits are about C's inputs **to that session**: its ECDHE share, its master-secret share, its masks. All of them are fresh random values drawn for one session and never reused, and an aborted session is thrown away. Think of a game of twenty questions where you get at most two or three yes/no answers per game and every game uses a new secret: you never get closer to anything. In the worst case (all checks pass) N may know about two bits of the 512-bit mask that hides C's two 32-byte traffic secrets; the AES-128 keys derived from them keep their full strength. The API key is never a 2PC input, and a node that makes sessions abort on purpose is visible to its clients.

Inputs are never opened. Every private input enters only as a garbled input or an OLE input, and nothing is revealed afterwards. Designs such as TLSNotary's DEAP or Garble-then-Prove get malicious security cheaply by having one party open its shares after the session; here that would hand C the master secret, hence `K_0`, hence the API key.

What is proven and what is argued

The QuickSilver product check and the IT-MAC openings are standard. Three-halves garbling has its own proof (Rosulek and Roy), but this implementation was rebuilt from the scheme's structure and its evaluator matrices must be compared with the paper by a reviewer. The AND-gate check with garbler-held masks, the 64-bit truncation and the salt are argued in the project's docs but have **no published proof**, and the construction has had no external review yet. Ferret and KOS are assumed secure against a malicious receiver (here N). ECtF is semi-honest.

9

The zero-knowledge proofs

9.1 The tool

VOLE-based zero knowledge (QuickSilver, from mpz2)

In: a Boolean circuit (here built from SHA-256 compressions, AES-128 blocks, XORs and a byte equality gadget), public inputs both parties know, and a secret witness only the prover knows. **Out, to the verifier:** only the values the statement chooses to decode. In these statements that is a few 32-byte hashes, which must equal public anchors, and one equality bit per plaintext comparison. **Guarantee:** the prover cannot make a false statement come out true (except with negligible probability), and the verifier learns nothing about the witness beyond the decoded values. **Shape:** interactive and designated-verifier: it convinces the other party, now, and nobody else. It runs on pre-made OT/VOLE correlations (Chou-Orlandi base OT, KOS, Ferret).

Every comparison against plaintext happens **inside** the circuit. When the client proves "this ciphertext decrypts to the suffix", the node receives one bit, never decrypted bytes, even when the proof fails.

9.2 π_NK: the node's record is the published template

prover	the node (sidecar), verifier: the client
secret	`K_0`, `IV_0` (28 bytes) and the API key bytes
public	`hk`; the template with zero bytes at the key positions and `Content-Length` filled in; the key offsets and length; the alphabet `[A-Za-z0-9_-]`; the 5-byte record header; `c_commit`
decoded	`hk`, `c_commit` (both compared to public values), one alphabet bit

What the circuit checks, as code:

```
def pi_nk(K0_IV0, key):                      # secret
    assert sha256(K0_IV0) == hk
    head = template.copy()
    for i, off in enumerate(key_offsets):
        head[off] = key[i]                   # splice the key in
    plain = head + b"\x17"                   # TLS inner content type
    ks = aes_ctr(K0, nonce=IV0 xor 0, start_counter=2)
    C1 = plain xor ks[:len(plain)]
    assert sha256(header + C1) == c_commit
    assert all(b in ALPHABET for b in key)
```

What it gives the client. The record the node committed to is exactly the published request head (same path, same headers, the declared `Content-Length`) with 42 key characters in the key slot, encrypted under the `K_0` that the 2PC produced (bound by `hk`, which C got from its own 2PC run). The node cannot point the request at another endpoint or add a header in the record it committed to.

What it does not give. The client never sees the bytes the node actually writes to U, and the node holds `K_0`: it could write a different record, or several. Section 11.4 discusses why that cannot touch the client's data.

9.3 π_C2S: the body ends with the pinned suffix

prover	the client, verifier: the node
secret	`c_ap_1` (32 bytes)
public	`h1`; the sequence number of the final body record; a ciphertext window at its end and the window offsets; the expected suffix
decoded	`h1` (compared), one equality bit

```
def pi_c2s(c_ap_1):                          # secret
    assert sha256(c_ap_1) == h1
    K1, IV1 = hkdf_label(c_ap_1, "key"), hkdf_label(c_ap_1, "iv")
    pt = aes_ctr_decrypt(K1, IV1 xor seq, window, offset=off)
    return pt[sel_start : sel_start + len(suffix)] == suffix
```

The window is the last `len(suffix) + 14` ciphertext bytes of the record (without the tag), and the selected bytes end right before the inner content-type byte, so the suffix is proven to be the **end** of the body. Because `h1` is a public output of the same 2PC that produced the keys, the client cannot prove against a secret of its choosing. The proof costs 7 SHA-256 compressions and about 15 AES blocks: 0.16 s locally, 0.04 s over a pre-warmed proof channel.

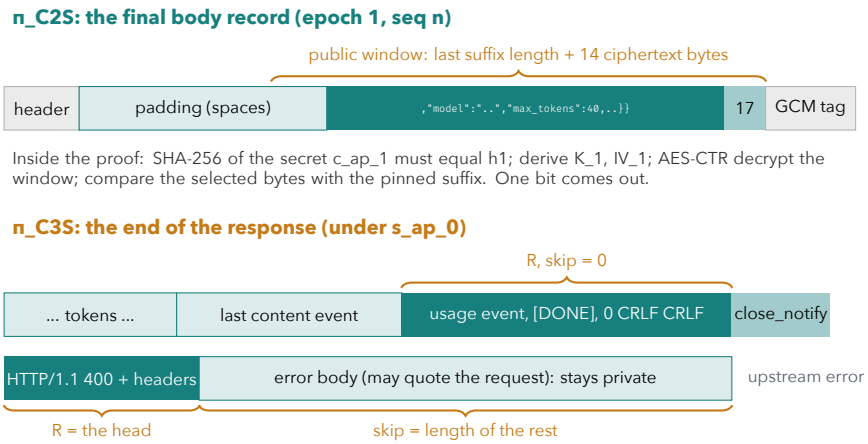


Figure 9.1: What π_C2S and π_C3S look at.

9.4 π_C3S: the disclosed usage is the provider's own

prover	the client, verifier: the node
secret	`s_ap_0` (32 bytes)
public	`h2`; the last server records (ciphertext without tags, with their sequence numbers), taken from the node's own copy of the transcript; R; `skip`
decoded	`h2` (compared), one equality bit

```
def pi_c3s(s_ap_0):                          # secret
    assert sha256(s_ap_0) == h2
    K, IV = hkdf_label(s_ap_0, "key"), hkdf_label(s_ap_0, "iv")
    content = b"".join(decrypt_record_content(K, IV, r.seq, r.ct)
                       for r in records)
    start = len(content) - len(R) - skip
    return content[start : start + len(R)] == R
```

For a usage disclosure `skip` is 0: R ends exactly where the response ends. For an upstream error R is only the response head and `skip` is the length of the rest, so the error body stays private but the head is anchored at a known distance from the end.

Why R cannot be faked from model output. The usage event is a raw SSE event preceded by HTTP chunk framing (`<hex size> CRLF`). Model output only

ever appears inside JSON strings, where CR and LF are escaped, so no model output can produce an event boundary or chunk framing. The project tested this by asking the model to emit fake usage events: every imitation stayed escaped inside a string. The same argument covers an error head: a strict status line followed by `Name: value` header lines cannot come from model output or chunk sizes.

9.5 Why the node cannot use the proofs as a decryption oracle

The node chooses the public parameters of π_C2S and π_C3S. A cheating node could try to aim a proof at other ciphertext and learn, from the one bit, whether some plaintext guess is right. The client therefore checks every parameter against its own records **before proving** (`client/confidential/params.ts`):

```
def check_pi_c2s(p, my_last_record, my_suffix, my_h1, my_seq):
    assert p.h1 == my_h1                        # anchor from MY 2PC
    payload = my_last_record[5:]                # without header
    ct_len  = len(payload) - 16                 # without tag
    w   = min(len(my_suffix) + 1 + 13, ct_len)
    off = ct_len - w
    assert p.seq == my_seq
    assert (p.off, p.w) == (off, w)
    assert p.sel_start == w - 1 - len(my_suffix)
    assert p.ct_window == payload[off : off + w]
    assert p.expected == my_suffix

def check_pi_c3s(p, my_server_records, my_R, my_skip, my_h2):
    assert p.h2 == my_h2 and p.expected == my_R
    assert p.skip == my_skip
    recs = my_server_records
    if len(recs[-1]) == 19:                     # one close_notify
        recs = recs[:-1]                        # may trail
    mine = recs[-len(p.records):]
    assert seqs_contiguous(p.records)
    for theirs, ours in zip(p.records, mine):
        assert theirs.ct == ours[:-16]          # exactly my records
    assert covered_bytes(mine) >= len(my_R) + my_skip

def check_pi_nk(p, signed_offer, my_body_len, my_hk):
    t = signed_offer.head_template
    filled = (t.replace(f"{{KEY:{signed_offer.key_length}}}",
                        "\0" * signed_offer.key_length)
```

```
            .replace("{LEN}", str(my_body_len))).encode()
    assert p.template == filled and p.head_len == len(filled)
    assert p.key_offsets == [i for i, b in enumerate(filled)
                             if b == 0]
    assert p.key_alphabet == signed_offer.key_alphabet
    assert p.hk == my_hk
    n = len(filled) + 1 + 16
    assert p.header == [0x17, 3, 3, n >> 8, n & 0xff]
```

The sidecar, symmetrically, runs each proof only against **its own** stored parameters for that proof name, and acts on a result only if the parameters the proof ran with are byte-identical to the stored ones.

9.6 The proof channel

Each proof needs fresh OT/VOLE correlations. Generating them per proof costs about twelve round trips and up to 1.5 MB: 5.4 s for π_NK and 3.0 s for π_C2S over a home uplink. So each session opens one long-lived channel (`/zk?proof=zkchan`) while it is armed and generates the correlations for both directions up front: chain “nk” (N proves, C holds the secret `Delta`) and chain “cp” (C proves π_C2S and π_C3S, N holds the `Delta`). Rules: fresh `Delta` per session, every correlation used once, each proof served at most once, a failed proof poisons the channel so a cheater can never probe twice, and only one proof runs at a time per session. Without a channel the client falls back to one WebSocket per proof.

10

Pinning the request: the suffix and the hold

10.1 The suffix

The sidecar and the client compute the same byte-exact suffix from the setup's model and token cap (`canonical_suffix` in `relay/src/main.rs`, `canonicalSuffix` in `suffix.ts`):

```
,"model":"deepseek-v4-flash","max_tokens":40,
"max_completion_tokens":40,"n":1,"stream":true,
"stream_options":{"include_usage":true},
"venice_parameters":{"include_venice_system_prompt":false,
"enable_web_search":"off"}}
```

(One line on the wire; 217 bytes for this model and cap.) It starts with a comma and ends by closing the top-level object. The SDK builds the body like this:

```
PINNED = {"model", "max_tokens", "max_completion_tokens",
          "max_output_tokens", "n", "stream", "stream_options",
          "venice_parameters"}

def build_pinned_body(request, model, max_tokens):
    private = {k: v for k, v in request.items()
               if k not in PINNED}
    prefix = json.dumps(private)[:-1]       # drop the closing }
    suffix = canonical_suffix(model, max_tokens)
    if prefix == "{":                       # no private keys
        return "{" + suffix[1:]
    return prefix + suffix
```

`max_tokens` is the caller's value clamped to the offer's cap (or the cap if none is given). Pre-warmed sessions have a fixed `len`, so a shorter body is padded with

spaces between the private part and the suffix; JSON ignores whitespace before the comma.

10.2 Why the suffix decides

With the provider's semantics (profile item 5) the **last** occurrence of a top-level key wins, and a later object replaces an earlier one. So whatever the private part says about `model` or `max_tokens`, the effective request uses the suffix's values. Strict parsing (malformed JSON and unknown fields rejected) means the private part cannot leave a string or an object open so that the suffix lands inside it as data.

10.3 The hold

The sidecar enforces that epoch-1 plaintext is exactly `len` bytes, so the record that completes the body is identifiable, and it holds that record until π_C2S verifies. The provider does no work before the complete body arrives (profile item 4). So when a proof fails, the provider has an incomplete request, the connection is closed, no inference ran and nothing is billed beyond the optional abandon fee.

Attack: a client names a cheaper model in setup and a pricier one in the body

Why it fails: the suffix with the setup model is proven to be the end of the body, and the last key wins, so the provider runs the setup model. A body that withholds the suffix never becomes complete, so the provider never starts. The adversarial gate (`crypto/tools/adversarial-model-swap.sh`) runs exactly this, plus a decoy model in the private part (settled as the setup model) and a body ending in another model's suffix while claiming the canonical one (π_C2S fails).

Attack: append a second request after the body

Why it fails: the sidecar refuses any byte past the declared length and any record after the held one; `Connection: close` and the template's fixed `Content-Length` leave no room for a smuggled request.

Attack: split the suffix across two records

Why it fails: π_C2S covers a window of the final record only; a split suffix cannot be proven, the held record is never released, and the session ends without inference.

Attack: raise the token cap or turn on web search in the private part

Why it fails: `max_tokens`, `max_completion_tokens` and `venice_parameters` are all in the suffix and win. Other price-changing fields (`service_tier`, `web_search_options`, `prediction`) are rejected by the provider (sampled by the profile gate). Image parts are rejected by every offered model (`crypto/tools/regression-modality.py`), and the reservation covers a full context window in any case.

11

Property by property

11.1 P1: the client cannot steal the API key

The key exists in the protocol only as plaintext of the node's epoch-0 record, encrypted under `K_0`. C never obtains `K_0`, `IV_0` or any secret they derive from (Z, HS, MS, `c_ap_0`): the 2PC gives them to nobody, or to N alone under N's mask, and nothing is opened afterwards. The secrets C does get (`c_hs`, `s_hs`, `s_ap_0`, `c_ap_1`) are one-way HKDF outputs and none of them is an ancestor of `K_0`.

Attack: record the node's egress and decrypt the credential record

Why it fails: the ciphertext is under `K_0`, which C does not have. This is exactly what protocol v2 adds over v1: in v1 the client knew the whole epoch-1 key, so anyone who saw the node's egress and colluded with the client could read the key.

Attack: put the client's own key share in the ClientHello

Why it fails: the sidecar relays a ClientHello only if it carries exactly one key share and that share is the joint point. With a second share of its own, C could finish the handshake alone and know every secret. The adversarial gate tests this case; it is refused before anything reaches the provider.

Attack: choose the client's point after seeing the node's

Why it fails: C commits to its point before it sees N's point. Otherwise it could pick its point so that the joint point is one whose secret C knows.

Attack: misreport the node's 2PC outputs, or learn them from the circuit

Why it fails: changing what N decodes needs a label C never received, which needs `Delta_A`. N's outputs pass through C only XORed with N's random mask. The API key is not a 2PC input.

Attack: brute-force the key from c_commit or hk

Why it fails: `c_commit` is a hash of a ciphertext; testing a guess needs `K_0`. `hk` is a hash of 28 uniformly random bytes. π_NK reveals only these digests and one alphabet bit.

Remaining assumptions: the provider never echoes the request head back (profile item 8), and the node's own process and the provider are not compromised.

11.2 P2: the node cannot read the prompt or the response

The body runs under `c_ap_1` and the response under `s_ap_0`; the 2PC gives both only to C, masked with C's mask. N's outputs are not ancestors of either. A node that garbles a wrong circuit (say, one that leaks `c_ap_1` into its own output) is caught by C's AND-gate check before C uses any key, except with probability about one in 2 to the power 62 per session, and can learn at most about 2.3 bits about C's fresh values through aborts. The proofs that decrypt under C's keys output one bit on parameters C has checked. What N learns in plaintext is public by design: the suffix, the usage event, or on an error the response head.

11.3 P3: the node cannot alter the request or the response

Why the response, tool calls included, cannot be altered

Every response record is AES-GCM sealed under `s_ap_0`, which only C (and U) hold. C's TLS stack checks every tag; a flipped byte, an injected record, a reordered record (the sequence number is in the nonce) or a record replayed from another session (fresh keys per session) fails the tag and ends the request with an error. Tool calls are just JSON inside the same stream, protected the same way. The node can only cut the stream; then C has no complete response and no usage event, and the request fails visibly.

The request body is sealed under `c_ap_1`, which N does not have, so N can neither read nor change it. The suffix is proven to be its tail, so the effective parameters are the agreed ones. After the response, C also checks that the model named in the response equals the requested one.

11.4 Residual risk: the node controls epoch 0

N holds `K_0`, so it can write any epoch-0 records, not only the proven head, and π_NK binds the content of the record N committed to, not the bytes it writes (C never sees them). N could use a different account or endpoint, or add a header such as one enabling provider-side retention. C cannot detect that. The damage is bounded: epoch 0 carries none of C's data; C's own records stay authenticated under `c_ap_1`; the response is readable only by C; and a broken request shows up as a failed response. A node can always misuse its own provider account. What the protocol guarantees is that it cannot read or alter C's content.

11.5 P4: the bill is the provider's own number

The billed usage is a substring of the authenticated response, proven under the session's server secret (π_C3S) and anchored at the end of the transcript the node itself relayed. `h2` is an output of the same 2PC that produced the keys, and N accepts it only through C's authenticated output report, so C cannot prove against a secret of its choosing without breaking SHA-256. N derives the suffix from the setup values, accepts only that suffix, and releases the complete request only after π_C2S, so U runs exactly the reserved model and cap.

Attack: the client discloses an earlier, smaller usage event

Why it fails: for a usage disclosure `skip` must be 0: R must end exactly at the end of the response (only a trailing close_notify may follow), and the sidecar parses R to end with `data: [DONE]` and the terminal chunk.

Attack: the client withholds the disclosure

Why it fails: once the complete request reached U, a missing disclosure costs the full reservation, so disclosing is always in the client's interest.

Attack: the client claims an upstream error

Why it fails: an error disclosure must parse as a strict HTTP status line (400 to 599) followed by well-formed header lines, and π_C3S proves where it sits in the real response. Model output cannot produce raw CR/LF, so a fake head cannot hide in a 200 response.

11.6 P5: both verify the provider

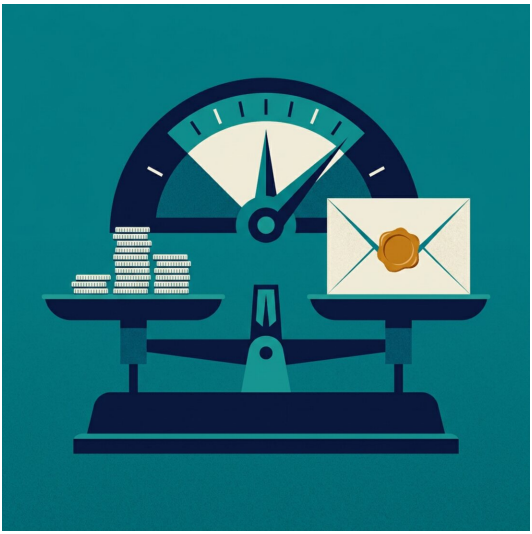
C verifies the chain to the WebPKI roots, the hostname (RFC 6125), CertificateVerify and the server Finished in its TLS stack, against an upstream host it pinned as trusted. The sidecar verifies the same from `c_hs` and `s_hs`, plus both Finished messages, fail-closed, before anything is written under `K_0`. The client's validation does RFC 5280 path validation without revocation checks.

11.7 P6: freshness

The session id hashes fresh nonces from both sides and the node's one-time reservation id. The TLS randoms, ECDHE scalars, masks, garbled tables, both global offsets, the MAC keys, the salt and the proof-channel correlations are all drawn fresh per session.

12

Billing



Nothing about money is specific to the protocol: confidential sessions use the node's ordinary accounts (the same reservation and finalize path as its EHBP opaque-body flow). The client's credential, top-ups, balance and refunds (`/v1/wallet/refund`) are the existing ones.

12.1 The reservation: why a full context window

The prompt is opaque to the node, so it cannot be priced in advance. The node reserves the worst case at the frozen rates:

```
prompt_limit = context_length - max_completion_tokens  # model
reserve = max(min_request_msat,
              ceil((prompt_limit * rate_in
                    + max_tokens * rate_out) / 1000))
reserve += max(session_fee_msat, abandon_fee_msat)
```

If the model has no known context length, the node's undiscounted maximum cost for the model is used instead (with the surcharge applied). Worked example from the paper: `deepseek-v4-flash` with a prompt limit of 104,858 tokens and `max_tokens` 40 reserves 2,282 msat; that session actually cost 63 msat. A model with a 1M-token context reserves about 157 sats at the case-study rates. The reservation is released down to the actual cost at settlement, but the credential needs that much balance to start a session. The token cap is enforced by π_C2S and the hold, and the prompt size by the context window itself, so the reservation always covers the real cost.

12.2 Settlement from the provider's usage event

```
def frozen_cost(rates, usage, reserved):
    p = usage["prompt_tokens"]
    o = usage["completion_tokens"]
    cached = usage.get("prompt_tokens_details", {}) \
                  .get("cached_tokens", 0)
    cached = min(max(cached, 0), p)
    cached_rate = min(rates["cached_in"], rates["in"])
    cost = ceil((rates["in"] * (p - cached)
                 + cached_rate * cached
                 + rates["out"] * o) / 1000)
    return min(max(cost, 0), reserved)

charge = min(frozen_cost(...) + session_fee_msat, reserved)
```

Cached prompt tokens are billed at the cached rate, never above the input rate. If the usage event's model differs from the setup model, the node charges the full reservation (the suffix pins the model, so that is a protocol breach). The charge goes through the node's ordinary `finalize_ehbp_actual_cost_payment`.

12.3 Fees: pricing the node's own work

A session costs the node about 22 MB of upload and 0.5 s of CPU before the provider sees a byte, and a pre-warmed session costs that before any request

exists. The node may publish, in the signed offer, three optional knobs (all 0 by default):

- **Rate surcharge** (`CONFIDENTIAL_RATE_SURCHARGE_PCT`): raises the per-token rates. It is already folded into the signed price list, so the client's rate check does not change.
- **Session fee** (`CONFIDENTIAL_SESSION_FEE_MSAT`): added to a completed request, and charged alone for a provider refusal (4xx, 503), which also reached the provider.
- **Abandon fee** (`CONFIDENTIAL_ABANDON_FEE_MSAT`): charged instead of nothing when a session the sidecar accepted ends without a request reaching U. Never below the session fee.

The abandon fee works like a deposit: it is reserved at setup and, when the session is used, everything above the session fee is in effect returned. A paying user never pays it; an armed session that is never used costs exactly that. The client accepts fees by echoing them in its setup; the node refuses a setup that does not, so an old client is never charged a fee it does not check. The deployed case-study node charges no session fee, no surcharge, and a 2-sat abandon fee; a completed short chat was charged 25 msat, an abandoned session 2,000 msat.

12.4 Upstream errors

An HTTP error from U carries no usage event. C discloses only the response head (status line, headers, blank line) with `skip` set to the rest of the response; the error body, which may quote the request (Venice quotes an unknown field's name), stays private. For 4xx and 503 the provider did no work: the node releases the reservation, keeping only the session fee. Other 5xx may have cost the node money upstream and are charged like a missing disclosure.

12.5 What happens on abort, at each point

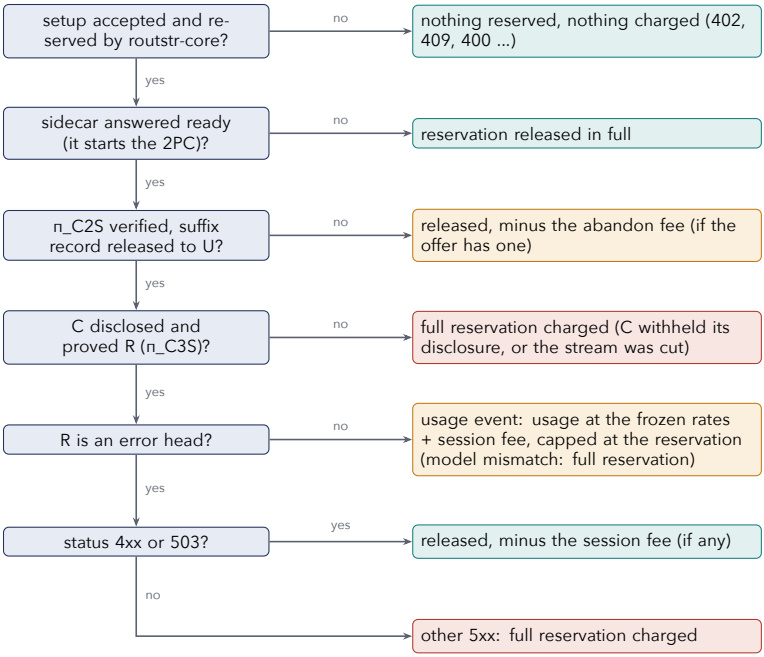


Figure 12.1: How a session's reservation ends. Teal: free, amber: fee or usage, red: full reservation.

When it stops	Outcome
setup refused (bad credential, no funds, unknown model, expired offer, fees not accepted)	nothing reserved; an explicit error frame with the status (402, 409, 400 …)
sidecar refuses the setup (no `ready`)	reservation released in full
2PC check fails, ClientHello rejected, handshake fails, π_NK fails, client leaves, armed session never used, π_C2S fails	U never got a complete request: released, minus the abandon fee if any
usage disclosed and proven	usage at frozen rates plus session fee, capped at the reservation
4xx or 503 disclosed	released, minus the session fee if any

When it stops	Outcome
other 5xx, or no disclosure after the body was released	full reservation charged

routstr-core decides from two flags: `engaged` (the sidecar answered `ready`) and `body_released` (the sidecar announced it is about to release the suffix record). The sidecar sends `body_released` before the write, and the node drains in-flight frames for up to 10 s after the client leaves, so the money decision sees the sidecar's real final state. Settlement runs shielded from client disconnects and finalizes exactly once.

> **Trust that remains**
>
> If the node cuts the stream before the usage event, it charges the reservation and the client cannot prove who aborted. That is the trust routstr already places in a node; the amount at the node's discretion is the reservation for the requested model.

13

The receipt and the attestation

After settlement routstr-core signs a receipt (BIP-340 Schnorr over SHA-256 of canonical JSON) and sends the exact signed bytes along:

```
{
  "v": 1,
  "sid": "5b0e...a4",
  "model": "deepseek-v4-flash",
  "usage": {"model": "deepseek-v4-flash", "choices": [],
            "usage": {"prompt_tokens": 2604,
                      "completion_tokens": 40,
                      "prompt_tokens_details":
                        {"cached_tokens": 0}}},
  "cost_msats": 63,
  "reserved_msats": 4282,
  "balance_msats": 918037,
  "rates": {"in": 21.7, "cached_in": 5.4, "out": 160.0},
  "abandon_fee_msat": 2000,
  "notary_pubkey": "7e4f...c1a2",
  "attestation_hash": "SHA256 of the attestation bytes",
  "time": 1791661236
}
```

(Made-up numbers chosen to reproduce the paper's example: 2,282 msat for the context plus the 2,000 msat abandon fee is reserved, 63 msat charged.) The client's checks (`client/confidential/receipt.ts`, `session.ts`):

```
def verify_receipt(msg, my):
    r = parse(msg.receipt_json)              # the signed bytes only
    assert schnorr_verify(msg.receipt_json, msg.sig, my.node_pubkey)
    assert r.sid == my.sid
    assert r.attestation_hash == sha256(my.attestation_json)
    if my.upstream_status >= 400:
        assert r.usage.error_status == my.upstream_status
        if my.upstream_status == 503 or my.upstream_status < 500:
```

```
        assert r.cost_msats <= offer.session_fee_msat
else:
    u = my.own_usage_event                # from my own plaintext
    assert r.usage.prompt_tokens == u.prompt_tokens
    assert r.usage.completion_tokens == u.completion_tokens
    assert r.model == u.model
    rate = offer.price_list[my.model]     # the SIGNED rates
    assert r.cost_msats <= ceil((rate.in * u.prompt_tokens
        + rate.out * u.completion_tokens) / 1000) \
        + offer.session_fee_msat
    assert r.session_fee_msat <= offer.session_fee_msat
assert effective_model(my.response) in (None, my.model)
```

The cost check prices every prompt token at the full input rate, so a node that bills cached tokens cheaper passes, and a node that overcharges fails. Earlier, the client checked the attestation's node signature and that its `sid`, `th_sh` and `th_sf` are its own: a validly signed attestation of another session is not evidence about this one. A receipt is the node's signed statement, useful to the client; it is not a proof to outsiders (the client holds `s_ap_0` and could forge server records for anyone but the node).

14

Denial-of-service limits

A session is expensive for the node before the provider sees a byte (22 MB of upload, 0.5 s of CPU, a reservation held), and pre-warming makes an unused session cheap to create. The limits, from free and always on to priced and optional:

Limit	Where	Value
nothing reaches U before a request	sidecar	TCP to U is opened only when C's ClientHello arrives, after setup, preprocessing and key share
validation and reservation before work	core, then sidecar	a client without funds gets no garbled circuit
concurrent sessions	core and sidecar	32 each (`MAX_SESSIONS`, `--max-sessions`)
time to the setup frame	core and sidecar	15 s
session lifetime	core	600 s from the WebSocket accept (the sidecar has its own 600 s timers)
2PC preprocessing plus key share	sidecar	180 s
one proof	sidecar	30 s; one proof at a time per session
body length	core and sidecar	1 byte to 8 MiB
`max_tokens`	core and sidecar	1 to the offer's cap (4096)
unused sessions per client address	core	12 (`CONFIDENTIAL_MAX_UNUSED_PER_IP`), HTTP 429; a session counts until the credential record is written
client bytes buffered before relaying	sidecar	`len + len/256 + 4` maximum-size TLS records (ClientHello, CCS, Finished, a full body)

Limit	Where	Value
proof-channel preallocation	both	each side proposes a size and the smaller wins (capped at 2 to the 25th), so C cannot make N preallocate more than N itself asked for
WebSocket message size	core	64 MiB

The client address is the TCP peer or, only for peers configured as trusted proxies, the rightmost `X-Forwarded-For` entry (the one the proxy appended); a client cannot name its own address. All Tor users of one onion service share an address and therefore the limit. Above that, the abandon fee prices every unused session (chapter 12). Many addresses can still open sessions, each paying the fee: that is a priced service, not an attack.

15

Costs and timings

All component figures are from an Apple M2 Max with release builds; “local” means no network delay.

The 2PC, cold, until both parties finish (median of 3; “up/down” is the client’s link):

Link	Dual execution (old)	Node-only (current)
local	0.17 s	0.63 s
40 ms one way, 2 / 50 Mbit/s	85.2 s	7.38 s
40 ms one way, 7 / 50 Mbit/s	24.9 s	5.41 s
25 ms one way, 100 Mbit/s symmetric	2.27 s	3.05 s
40 ms one way, 2 / 10 Mbit/s	85.2 s	20.0 s
client sends / receives	21.1 / 21.1 MB	1.21 / 22.1 MB

Proofs:

Proof	Local	50 ms RTT, per-proof setup	Pre-warmed channel, home link	Data
π_NK	0.16 s	1.23 s	0.21 to 0.25 s	1.45 MB
π_C2S	0.16 s	1.22 s	0.04 s	1.45 MB
π_C3S	0.20 s	1.27 s	0.06 s	1.47 MB

End to end against the live provider (`deepseek-v4-flash`, 40 tokens):

- one host, cold: request start to body sent about 1.2 s; the held suffix proof 0.10 s;
- one host, pre-warmed: claim to body sent 5 ms;

- a good home link (about 100 Mbit/s down, 60 to 80 ms to the node), cold: preprocessing 5.1 to 11.7 s, first token after 13.8 to 16.2 s; fully armed: 73 ms from claim to body and the first token after 1.05 s, most of it the provider's own latency;
- a poor, download-limited link: a cold request waits about a minute.

Per request: about 23.3 MB of 2PC traffic plus 4.4 MB of proofs, about 28 MB in all; the node uploads about 24 MB (100 requests per minute is about 0.3 Gbit/s) and spends about 0.5 s of CPU on the 2PC. None of this depends on the size of the prompt or the response: after the handshake the payload streams at native TLS speed.

16

Pre-warming

The garbled tables are single-use, so the protocol hides their cost instead of sharing it. What an armed session has already done, and what it still has to do:

Precomputed while armed	Done when a request claims it
setup, credential check, reservation	ClientHello, ECtF, phases A and B
the whole 2PC preprocessing (22 MB)	handshake verification on both sides
the joint key share	π_NK, the credential record and the KeyUpdate
proof-channel correlations for π_NK, π_C2S, π_C3S	the body, π_C2S, the response, π_C3S, settlement

An armed session stops **before the ClientHello**: a provider closes a connection that sits idle in the handshake within seconds, and an armed session holds no provider connection at all. (The reference client's `--arm` mode goes further, through the handshake and π_NK, for benchmarks.)

The SDK pool (`client/confidential/prewarm.ts`, defaults as shipped):

- **Key** of the pool: node, model, `max_tokens` (the suffix carries it) and credential.
- **Traffic only.** Nothing is armed at startup or for a key without a request in the last 10 minutes; an idle pool runs no timer.
- **How many.** One per active key, two after a burst (3 requests in 5 minutes), or the peak number of overlapping requests; at most 6 per key and 6 in total, which stays below the node's 12-per-address limit. One arm at a time (it is bandwidth-heavy), never while a cold request is still in its own preprocessing.
- **Body length.** `len` is fixed at setup, so the pool picks a power-of-two bucket: the largest of the last 20 bodies times 1.25, from 4 KiB to 8 MiB; shorter bodies are padded with spaces. A larger request goes cold and raises the bucket.

- **Retire** at 480 s after setup (the node ends sessions at 600 s); close a key's sessions as soon as it goes idle.
- **Money.** Arm only after a request for the key has settled (the receipt tells the reservation size) and while the free balance is at least twice that; arm only if the abandon fee is at most 5,000 msat, and at most 6 sessions in total for nodes that charge one. Fees paid for unused sessions are logged.
- **Failures.** A failed arm backs the key off (30 s doubling to 10 min). A claimed session that fails before `release_head` falls back to a cold session; after that it is never retried (the request may have reached the provider).

17

Images

The same session can carry an image request instead of a chat completion. Nothing in the TLS epochs, the 2PC or the three statements changes. What changes is the HTTP request, the price and how the session settles. The design and the binding decisions are in `docs/CONFIDENTIAL-IMAGES.md` (section 0).

The offer. A node that serves images adds `capabilities: ["chat", "images"]` and an `images` block to its signed offer:

```
"images": {
  "head_template": "POST /api/v1/image/generate HTTP/1.1\r\n...",
  "count_key": "variants", "max_count": 4,
  "fixed_pins": [["return_binary", false], ["enhance_prompt", false],
                 ["enable_web_search", false], ["style_references", []]],
  "param_space": {"resolution": ["1K", "2K", "4K"], "format": ["webp"]},
  "price_list": {"qwen-image-2-1-turbo": [
    {"id": "1K-webp", "pins": [["resolution", "1K"], ["format", "webp"]],
     ↪  "msat_per_image": 24346}]}
}
```

The node builds the price list from its per-image price books. A model is offered only if its price is a function of the pinned parameters alone; token-metered models and providers that report their own cost are left out. The block comes from the sidecar offer file's `endpoints.images`: without it the node advertises no images.

The suffix. For model M, count k and variant V the suffix is, byte for byte, `,"model":M,"variants":k,` then the fixed pins, then V's pins, then `}`, in compact JSON. The node renders it and forwards it to the sidecar, which checks it; the client renders its own from the signed offer and requires `ready.suffix` to match. π_C2S proves it exactly as it proves the chat suffix. The prompt and every option that does not change the price (`aspect_ratio`, `seed`, `negative_prompt`, safety mode) stay in the private prefix.

Setup. The client's setup names `kind: "images"`, the `model`, the `variant` id and the `count`, with no `max_tokens`, and is never `armed`. The node reserves `msat_per_image × count + max(f_s, f_a)`.

Settlement: `accept`. The charge is fixed at setup, so no usage needs proving. After the attestation the client sends `{"type": "accept"}` for a 2xx; the sidecar reports a flat usage and the node charges `msat_per_image × count + f_s`. For an error status the client discloses the response head as for chat (4xx/503: released less `f_s`). A client that sends neither is charged the reservation, which is never less than the flat price. The receipt adds `kind`, `variant`, `count` and `msat_per_image`, and the client checks `cost_msats == ceil(msat_per_image × count + f_s)`.

What the node learns. The model, the variant and the count (as it learns the model and token cap of a chat), the body length, the response size and timing, and success or the error status. Not the prompt, the private options, the image, or the success headers (which say whether the image was blurred).

APIs. In the browser, `client.images.generate({model, prompt, count, resolution, aspectRatio})` returns the images as `Blob`s with the verified session; a node without the capability throws `offer_no_images`, a provider error `UpstreamImageError`. In the SDK and routstrd, `POST /v1/images/generations` (OpenAI shape) and `POST /v1/image/generate` (Venice shape) with the confidential opt-in run an image session and answer `{created, data: [{b64_json}]}`.

No pre-warming. Image sessions are never armed and never touch the chat pool: each one pays its cold 2PC, about 26 MB of protocol traffic before the image. On the live node a one-image request took 10 s from the reference client; in the browser a cold request took 16.7 s, 7.2 s of it the provider generating the image.

18

What it does not hide, and other limits

- **The provider sees everything.** The prompt, the response, and the API key. The protocol removes trust in the node, not in the provider.
- **Traffic analysis.** N sees every record size and its timing. The body length is visible, and streaming providers emit about one record per token or token group, so the response's token-length sequence leaks; such side channels have reconstructed large parts of LLM responses in published attacks. Prompts can be padded inside the JSON body; the response side needs provider-side padding.
- **Error metadata.** On an upstream error N learns the status line, the headers and the body length, not the body.
- **The node controls epoch 0** (section 11.4).
- **The 2PC is a new variant** without a published proof or external review (chapter 8), and leaks up to 2.3 bits per session about fresh values.
- **P-256.** The joint key share needs point addition, so X25519 cannot be used.
- **No third-party proof.** Everything convinces only the two parties of the session.
- **Provider drift.** A provider change can break pinning or usage reporting; the profile gate must be re-run, and broken sessions fail rather than fall back.
- **Endpoints.** Keys live in the client's process and the API key in the node's memory. The client's certificate validation does no revocation checking.

19

The predecessors: v1 and dual execution

v1, the single-party variant (branch `v1-poc`). No 2PC: C runs the TLS handshake alone with an X25519 share and discloses only `c_hs` and `s_hs`. It sends a KeyUpdate in epoch 0 and hands N **one-shot record material** for epoch-1 sequence 0 (the keystream truncated to the head length, the GHASH key and the tag mask). Proof π_C1 shows N that this material comes from the session's key schedule; N forms its record by XOR and proves its content with π_N; C then rotates again and writes the body in epoch 2. π_C2 and π_C3 are like π_C2S and π_C3S but recompute the key schedule from HS. Its weakness is that C holds the whole epoch-1 key, so the API key equals the record's ciphertext XOR a keystream C knows. P1 holds only as long as C never sees that ciphertext, an operational assumption (anyone on the node's egress path colluding with C breaks it). v1 needs about 5.9 MB per request instead of about 28 MB, and suits nodes that control their upstream path.

Dual execution (tag `v2-symmetric`) was the first v2 2PC: each phase was garbled twice, once by each party, and the outputs compared with a commit-then-open hash of output labels. It rests on simpler, better-studied constructions and leaks less (about 1.6 bits per session), but each party garbled 656,179 AND gates and uploaded about 21 MB: 85 s cold on a 2 Mbit/s uplink against 7.4 s now.

A

Glossary: the paper's names and symbols

Paper	This book	Meaning
C, N, U	client, node, provider	the client daemon, the routing node (core and sidecar), the upstream API
P1 to P6	the goals	credential, content, integrity, accounting, authenticity, freshness
Z	the ECDHE secret	x-coordinate of the shared point; split into additive shares
a_C, a_N; P_C, P_N; P; S	scalars; points; joint share; server share	ECDHE contributions; P is the only key share U sees
HS, MS	handshake secret, master secret	inner nodes of the key schedule; nobody holds them
c_hs, s_hs	handshake traffic secrets	both parties get them to verify the handshake
c_ap,0; K_0, IV_0	the node's epoch key	secret and record key/IV of epoch 0
c_ap,1; K_1, IV_1	the client's body key	secret and record key/IV of epoch 1
s_ap,0	the response key	server application secret
th_SH, th_SF	transcript hashes	through ServerHello; through server Finished
h_1, h_2, h_k	anchors h1, h2, hk	public hashes of c_ap,1, s_ap,0, K_0 and IV_0
r_C, r_N; m_C, m_N	masks	MS masks (phase A); output masks (phase B)
ECtF, A2M, M2A, OLE	share conversion	turn two points into additive shares of the x-coordinate
Delta_A	the node's garbling offset	difference between the two labels of every wire
Delta_B	the client's MAC key	global key of the IT-MAC tags; 127 random bits

Paper	This book	Meaning
lambda_w	a wire's mask	N's secret bit; last bit of N's 0-label
Lambda_w	pointer bit	true value XOR mask; what C sees
t_k	mask product	product of an AND gate's two input masks, tagged
chi	the challenge	from C's seed, after N committed
e_k, E	gate error, combined error	zero for every consistently evaluated gate
Y_N, check rows, pi(.)	check material, 64-bit projection	what N sends for the AND-gate check; truncation to 64 bits
KOS, Ferret, COT, RCOT	correlated OT	pre-made random correlations
QuickSilver, VOLE-ZK	the proof system	interactive, designated-verifier ZK of mpz2
π_NK, π_C2S, π_C3S	the three proofs	record template; suffix; usage disclosure
π_C1, π_N, π_C2, π_C3	v1 proofs	record material; template; suffix; usage
R, skip	the disclosure	final plaintext of the response; undisclosed bytes after it
L, T, m	prompt limit, token cap, minimum	inputs of the reservation
r_in, r_out, r_cached	rates	msat per 1,000 tokens from the frozen offer
f_s, f_a	session fee, abandon fee	optional fees in the signed offer
sid	session id	SHA-256 of both nonces and the reservation id
DEAP, Garble-then-Prove	open-after designs	2PC-TLS designs that reveal shares afterwards; unusable here
WRK17, KRRW18, DILO22, CWYY23	authenticated garbling	the literature the node-only check follows

B

Where things are in the code

Concern	File
sidecar state machine, setup gate, buffers, record hold, R parsing	`crypto/relay/src/main.rs` (`handle_setup`, `handle_control`, `relay_body`, `v2_pump_client`, `v2_progress`, `validate_r`, `validate_error_head`)
ClientHello and ServerHello rules	`crypto/relay/src/tls_v2.rs` (`check_client_hello`, `server_hello_share`)
handshake verification	`crypto/relay/src/tls.rs` (`verify_handshake`)
the 2PC	`crypto/twopc/src/lib.rs` (preprocessing, phases), `gc.rs` (three-halves garbling, check rows), `circuits.rs`, `ectf.rs`; binary `bin/cu-2pc.rs`
proof statements	`crypto/zkbench/src/statements_v2.rs` (`build_pi_nk`, `build_pi_c2s`, `build_pi_c3s`), `gadgets.rs`, `channel.rs` (proof channel)
offer, reservation, settlement, fees	`routstr-core/routstr/confidential.py`
node WebSocket relay, per-address limit, abort handling, receipt	`routstr-core/routstr/confidential_ws.py`
client session and checks	`routstr-sdk/client/confidential/session.ts`, `params.ts`, `receipt.ts`, `offer.ts`, `suffix.ts`, `transport.ts`
pre-warming	`routstr-sdk/client/confidential/prewarm.ts`, `zkchan.ts`, `twopc.ts`
provider and adversarial gates	`crypto/tools/provider-profile-check.py`, `adversarial-model-swap.sh`, `regression-modality.py`