Ako funguje dôverný režim routstr

Príručka pre programátorov

Juraj Bednar <juraj@bednar.io>, s pomocou AI

Ako funguje dôverný režim routstr

Príručka k protokolu v2 pre programátorov: čo sa počíta, ako, kto čo vie a prečo jednotlivé vlastnosti platia

Juraj Bednar <juraj@bednar.io>, s pomocou AI

10. októbra 2026

Sprievodca k článku *Confidential and verifiable upstream inference over standard TLS 1.3*. Opisuje kód na vetve `twopc-v2` repozitára routstr-confidential a na vetve `confidential-2pc` repozitárov routstr-core a routstr-sdk.

Obsah

Ako čítať túto knihu viii

1 Problém a nápad 1
1.1 Čo je dnes smerovací uzol 1
1.2 Nápad v jednom odseku 2
1.3 Ciele 2

2 Model hrozieb a dôvera 4
2.1 Kto môže podvádzať 4
2.2 Čomu dôverujeme 4
2.3 Čo je mimo rozsahu 5

3 TLS 1.3, len to, čo potrebujeme 6
3.1 Odvodzovanie kľúčov je strom 6
3.2 Pozorovanie, na ktorom protokol stojí 7

4 Rozdelenie: kto dostane ktorú vetvu 8
4.1 Kto drží ktoré tajomstvo 8
4.2 Epochy na linke 9

5 Čo musí poskytovateľ spĺňať 11

6 Komponenty 13

7 Jedna relácia krok za krokom 15
7.1 Krok 0: ponuka 15
7.2 Krok 1: setup a rezervácia 17
7.3 Krok 2: predspracovanie a spoločný key share (nabíjanie) 19
7.4 Krok 3: TLS handshake s dvoma online fázami 2PC 21
7.5 Krok 4: epocha 0, záznam s API kľúčom 23
7.6 Krok 5: telo a zadržanie záznamu 25
7.7 Krok 6: odpoveď 26
7.8 Krok 7: odhalenie a vyúčtovanie 27
7.9 Kto čo vie v jednotlivých krokoch 28

8 Dvojstranný výpočet 29
8.1 Stavebné bloky ako API 30
8.2 Čo obvody počítajú 32
8.3 Ako sa C uistí, že uzol garbloval poctivo 33
8.4 Predspracovanie a online fáza 35

8.5 Únik: najviac 2,3 bitu na reláciu 36

9 Dôkazy s nulovou znalosťou 37
9.1 Nástroj 38
9.2 π_NK: záznam uzla zodpovedá zverejnenej šablóne 38
9.3 π_C2S: telo končí zafixovaným sufixom 39
9.4 π_C3S: odhalená spotreba pochádza priamo od poskytovateľa . . 40
9.5 Prečo uzol nemôže dôkazy zneužiť ako dešifrovacie orákulum . . 41
9.6 Kanál pre dôkazy 42

10 Zafixovanie požiadavky: sufix a zadržanie 43
10.1 Sufix 43
10.2 Prečo rozhoduje sufix 44
10.3 Zadržanie 44

11 Vlastnosť po vlastnosti 46
11.1 P1: klient nemôže ukradnúť API kľúč 46
11.2 P2: uzol nemôže čítať prompt ani odpoveď 47
11.3 P3: uzol nemôže zmeniť požiadavku ani odpoveď 47
11.4 Zvyškové riziko: epochu 0 ovláda uzol 48
11.5 P4: účet je číslo od samotného poskytovateľa 48
11.6 P5: poskytovateľa overujú obe strany 49
11.7 P6: čerstvosť 49

12 Účtovanie 50
12.1 Rezervácia: prečo celé kontextové okno 50
12.2 Vyúčtovanie podľa údaja o spotrebe od poskytovateľa 51
12.3 Poplatky: cena za prácu uzla 52
12.4 Chyby poskytovateľa 52
12.5 Čo sa stane pri prerušení v jednotlivých bodoch 53

13 Potvrdenka a atestácia 55

14 Limity proti DoS 57

15 Náklady a časy 59

16 Predhrievanie 61

17 Obrázky 63

18 Čo neskrýva a ďalšie obmedzenia 65

19 Predchodcovia: v1 a dual execution 66

A Slovník: názvy a symboly z článku **67**

B Kde čo nájdeš v kóde **69**

Ako čítať túto knihu

Kniha prerozpráva protokol z článku *Confidential and verifiable upstream inference over standard TLS 1.3* (protokol v2, ten nasadený) pre programátora. Matematickú notáciu nepoužíva. Každý kryptografický nástroj opisuje jeho kontraktom, ako API: čo doň vstupuje, čo z neho vychádza, kto sa čo dozvie a čo garantuje. Výklad sa drží kódu.

Postavy:

- **C**, klient. V produkcii je to routstrd s routstr SDK (to isté robí aj referenčný klient `crypto/client/src/driver.ts`). C spúšťa dva pomocné binárne súbory: `cu-2pc` (svoju polovicu dvojstranného výpočtu) a `cu-prover` (svoju polovicu dôkazov s nulovou znalosťou). Tajomstvá čítajú zo stdin, takže žiadne sa nedostane na disk.
- **N**, uzol. Tvoria ho dva procesy: **routstr-core** (Python, FastAPI: ponuka, kontrola setupu, účtovanie, potvrdenky; drží nostrový podpisový kľúč uzla a peniaze) a **cu-sidecar** (Rust: TLS relay, polovica 2PC a dôkazov na strane N; drží API kľúč poskytovateľa). Ani jeden nedrží obe veci naraz.
- **U**, poskytovateľ (v prípadovej štúdii `api.venice.ai`). Nič sa na ňom nemení, vidí obyčajného TLS 1.3 klienta.

Kapitola 7 prejde sekvenčnými diagramami jednu celú reláciu. Ak si prečítaš len jednu kapitolu, nech je to táto. Kapitoly 8 až 10 otvárajú tri čierne skrinky (2PC, dôkazy a fixovanie požiadavky). Od kapitoly 11 ide výklad vlastnosť po vlastnosti: bezpečnosť, účtovanie, limity, náklady.

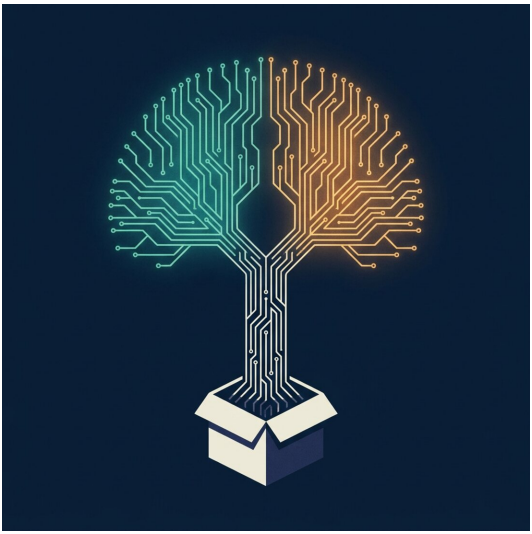
Opakovane sa objavujú dva druhy rámčekov:

> **Prečo je to bezpečné**
> Argument pre danú vlastnosť na pár riadkov.

> **Útok: čo by mohol skúsiť podvodník**
> **Prečo to nevyjde:** mechanizmus, ktorý útok zastaví.

1

Problém a nápad



1.1 Čo je dnes smerovací uzol

Routstr uzol má účet u poskytovateľa LLM. Zaplatíš uzlu (zostatkom na `sk-` kľúči alebo e-cash tokenom), ten prepošle tvoju chatovú požiadavku so svojím API kľúčom a poskytovateľovi zaplatí sám. Je to plnohodnotný man in the middle: číta tvoj prompt aj odpoveď, mohol by ktorékoľvek z nich zmeniť (vymeniť model, upraviť volanie nástroja) a účet ti vystavuje on. Uzol ti zas nesmie nikdy dať svoj API kľúč.

Bežnou odpoveďou sú enklávy (TEE), lenže tie presúvajú dôveru na výrobcu hardvéru a na meranie softvéru, ktoré by každý používateľ musel po každej aktualizácii skontrolovať. Tento protokol stavia na niečom, na čo sa používatelia už spoliehajú: na TLS certifikáte poskytovateľa.

1.2 Nápad v jednom odseku

Klient je TLS 1.3 klientom poskytovateľa a uzol iba preposiela záznamy. Uzol prispeje do prúdu presne jedným TLS záznamom, hlavičkou HTTP požiadavky s `Authorization: Bearer <api key>`, zašifrovaným kľúčom, ktorý má len on. Všetko ostatné (JSON telo s tvojím promptom, streamovaná odpoveď) je zašifrované kľúčmi, ktoré má iba klient. Aby to bolo bezpečné pre obe strany, treba tri nástroje:

1. **Dvojstranný výpočet (2PC) odvodzovania kľúčov TLS** (TLS 1.3 key schedule): klient a uzol spolu vystupujú ako jeden TLS klient a výpočet dá každej strane iba jej vlastné kľúče. ECDHE tajomstvo ani master secret nedrží nikto, nikdy.
2. **Dôkazy s nulovou znalosťou**, ktoré určujú, čo smie každá strana zapísať: uzol dokazuje, že jeho záznam zodpovedá zverejnenej šablóne (π_NK); klient dokazuje, že jeho telo končí verejným JSON sufixom, ktorý zafixuje model a strop tokenov (π_C2S), a že spotreba, ktorú hlási, je tá od poskytovateľa (π_C3S).
3. **Bežné účtovanie** uzla: rezervácia na najhorší prípad pri setupe, platba podľa údaja o spotrebe (usage event) od samotného poskytovateľa a podpísaná potvrdenka, ktorú klient overí.

1.3 Ciele

ID	Vlastnosť	Jednoducho povedané
P1	Dôvernosť API kľúča	C sa nikdy nedozvie API kľúč, ani keď si zaznamená každý bajt, ktorý N pošle U
P2	Dôvernosť obsahu	N nikdy nečíta prompt ani odpoveď; vidí šifrový text a údaje na účtovanie (údaj o spotrebe, pri chybe poskytovateľa hlavičku odpovede)
P3	Integrita	U dostal telo od C za dohodnutými parametrami; C číta presne to, čo U poslal, vrátane volaní nástrojov
P4	Overiteľné účtovanie	platba vychádza zo spotreby, ktorú U nahlásil v tejto relácii

ID	Vlastnosť	Jednoducho povedané
P5	Autenticita poskytovateľa	obaja overujú certifikát a hostname U
P6	Čerstvosť	každá relácia je živá, viazaná na jednu platbu a nedá sa použiť znova

2

Model hrozieb a dôvera

2.1 Kto môže podvádzať

O C aj N predpokladáme, že sa voči tomu druhému správajú **zlomyseľne**: ktorýkoľvek môže posielať nezmysly, odchýliť sa od protokolu, kedykoľvek reláciu prerušiť alebo sa spolčiť s niekým, kto sleduje sieť. Obyčajný **pozorovateľ siete** medzi ľubovoľnými dvoma stranami vidí iba TLS alebo WebSocket prevádzku.

2.2 Čomu dôverujeme

- **Poskytovateľovi U.** Dodržiava TLS 1.3 a svoje vlastné API a musí spĺňať profil poskytovateľa z kapitoly 5. Prompt vidí (spúšťa ho) aj API kľúč (sám ho vydal), presne ako dnes.
- **Certifikátu poskytovateľa a WebPKI.** C aj N overujú reťazec až po koreňové certifikáty WebPKI a hostname. C prijme iba hostnamy poskytovateľov zo svojho zoznamu dôveryhodných a iba ponuky podpísané kľúčom uzla, ktorý si zafixoval.
- **Tvojmu vlastnému počítaču** (proces C drží kľúče C) a v prípade uzla jeho vlastným procesom (sidecar drží API kľúč v pamäti).
- **Štandardnej kryptografii**: SHA-256, HMAC, AES-GCM, ECDH na P-256 a stavebným blokom pre oblivious transfer, VOLE a garblovanie z knižnice `mpz2` od TLSNotary.

Ničomu inému sa neverí: protokol nepoužíva enklávu ani atestovaný softvér a nikdy nestavia na tom, že sa prevádzkovateľ správa slušne.

2.3 Čo je mimo rozsahu

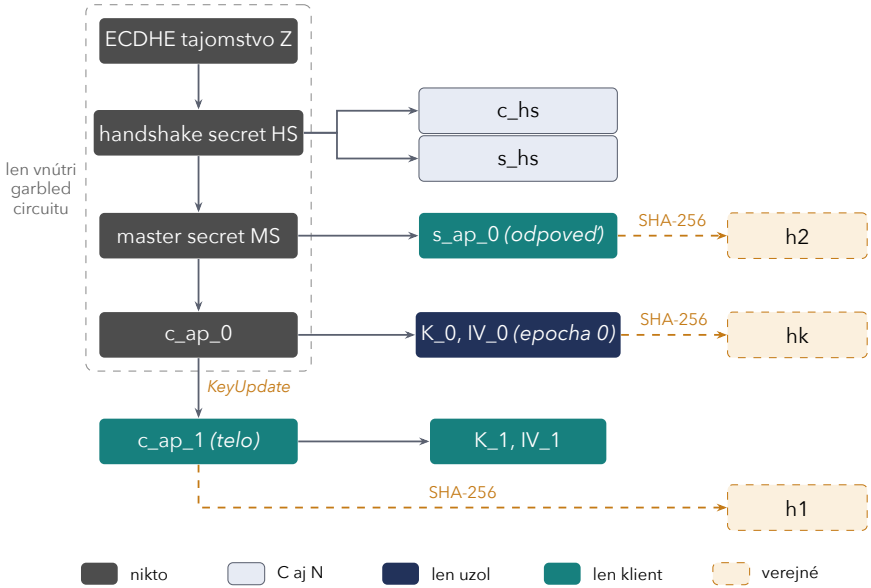
Skrytie promptu pred U, dôkaz o tom, ktoré váhy bežali, skrytie objemu a časovania prevádzky pred N a dôkazy, ktoré by vedela overiť tretia strana. Dôkazy sú interaktívne a presvedčia iba druhú stranu relácie, čo na vyúčtovanie stačí.

3

TLS 1.3, len to, čo potrebujeme

3.1 Odvodzovanie kľúčov je strom

Po výmene kľúčov majú oba TLS konce spoločné ECDHE tajomstvo. TLS z neho odvodí strom tajomstiev. Každá šípka na obrázku 3.1 je HMAC-SHA-256 (HKDF), teda jednosmerná funkcia. Z potomka sa nedostaneš späť k rodičovi a z jedného potomka nevypočítaš jeho súrodenca.



Obr. 3.1: Odvodzovanie kľúčov v TLS 1.3, farebne podľa toho, kto napokon drží ktoré tajomstvo.

Protokol využíva tieto časti:

- Z každého traffic secretu vznikne AES-128 kľúč a 12-bajtový IV (labely „key“ a „iv“).
- Záznam s poradovým číslom `n` používa nonce `IV xor n`. Šifruje ho AES-GCM: režim CTR od počítadla 2 a k tomu GHASH tag. Poradové čísla začínajú v každej epoche znova od 0.
- **KeyUpdate** umožní odosielateľovi vymeniť vlastný odosielací kľúč na hranici ľubovoľného záznamu (z `c_ap_0` na `c_ap_1`). S `update_not_requested` sa mení len tento smer; server ďalej posiela pod `s_ap_0`.
- Do odvodení vstupujú dva hashe transkriptu: `th_SH` (ClientHello a ServerHello) a `th_SF` (všetko až po serverový Finished).

3.2 Pozorovanie, na ktorom protokol stojí

`K_0` (kľúč záznamov epochy 0) a `c_ap_1` (klientske tajomstvo epochy 1) sú súrodenci pod `c_ap_0`. Z `c_ap_1` a `s_ap_0` sa o `K_0` nedozvieš nič a z `K_0` zas nič o `c_ap_1` ani `s_ap_0`. Keby teda strom vypočítal dôveryhodný stroj a každej strane dal iba jej vetvu, uzol by mohol písať epochu 0, klient epochu 1 a čítať odpoveď, a ani jeden by nevedel čítať ani falšovať časť toho druhého. Dôveryhodný stroj neexistuje, takže ho strany emulujú dvojstranným výpočtom.

4

Rozdelenie: kto dostane ktorú vetvu

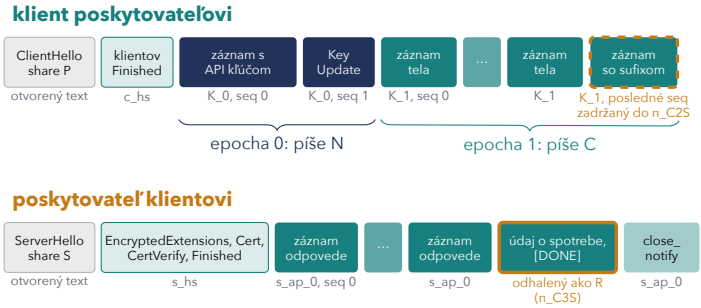
4.1 Kto drží ktoré tajomstvo

Tajomstvo	C	N	U	Načo slúži
ECDHE skalár	svoj podiel	svoj podiel	svoj	výmena kľúčov
ECDHE tajomstvo Z	aditívny podiel	aditívny podiel	áno	iba vstup do 2PC
HS, MS, c_ap_0	**nikdy**	**nikdy**	áno	existujú iba vnútri garbled circuitu
c_hs, s_hs	áno	áno	áno	obaja overujú handshake (P5)
K_0, IV_0	**nikdy**	áno	áno	záznam s API kľúčom od N (P1)
c_ap_1	áno	**nikdy**	áno	telo požiadavky od C (P2)
s_ap_0	áno	**nikdy**	áno	odpoveď (P2)
h1 = SHA256(c_ap_1)	áno	áno	nie	verejná kotva pre π_C2S
h2 = SHA256(s_ap_0)	áno	áno	nie	verejná kotva pre π_C3S
hk = SHA256(K_0, IV_0)	áno	áno	nie	verejná kotva pre π_NK
API kľúč	**nikdy**	áno	áno	iba vnútri záznamu N v epoche 0

Tri **kotvy** sú hashe 32 náhodne vyzerajúcich bajtov (pri `hk` 28), takže ich zverejnenie nič neprezradí. Neskorší dôkaz vd'aka nim môže povedať „tajomstvo, ktoré používam, je to, ktoré mi dalo 2PC“ bez toho, aby vnútri dôkazu znova počítal odvodzovanie kľúčov. Každá strana si navyše svoje kotvy prepočíta z tajomstiev, ktoré dostala, a pri nezhode reláciu preruší.

4.2 Epochy na linke

Smer od klienta k serveru delí na epochy jeden KeyUpdate. Epocha 0 patrí uzlu, epocha 1 klientovi.



Obr. 4.1: Záznamy jednej relácie tak, ako ich vidí U. Uzol zapíše iba dva tmavomodré záznamy.

Epocha	Kľúč	Kto píše	Čo
handshake	c_hs / s_hs	C (U odpovedá)	ClientHello so spoločným key share; klientsky Finished, ktorý C vypočíta z c_hs
0	c_ap_0 (K_0, IV_0)	**N**	seq 0: záznam s API kľúčom (hlavička požiadavky s `Authorization`); seq 1: `KeyUpdate`
1	c_ap_1	C	JSON telo od seq 0; posledný záznam je verejný sufix, ktorý U dostane až po π_C2S
odpoveď	s_ap_0	U	streamovaná odpoveď; N preposiela šifrový text

V epoche 0 C nič nepíše. Jeho TLS stack (fork TLS 1.3 klienta s licenciou MIT, doplnený o „external“ hook) sám nikdy žiadny kľúč nevypočíta: po handshaku mu 2PC vloží `c_ap_1` ako odosielacie tajomstvo (od poradového čísla 0) a `s_ap_0` ako prijímacie. U vidí obyčajného klienta, ktorý poslal jeden záznam, vymenil kľúče a pokračoval.

Prečo záznam uzla a záznamy klienta potrebujú rôzne kľúče

Pri jednom spoločnom kľúči by ten, kto ho drží, vedel dešifrovať záznamy druhej strany (klient by čítal API kľúč, uzol prompt) a falšovať záznamy v jej mene: AES-GCM dvoch pisateľov s jedným kľúčom nijako neoddeľuje. KeyUpdate je jediný štandardný spôsob, ako v TLS 1.3 vymeniť odosielací kľúč klienta uprostred spojenia tak, aby si server nevšimol nič nezvyčajné, preto slúži ako bod odovzdania. Smer odpovede kľúč nikdy nemení a jeho kľúč má iba C.

5

Čo musí poskytovateľ spĺňať

Protokol funguje s každým poskytovateľom, ktorý spĺňa tento profil. Pri Venice body 1 až 7 naživo kontroluje `crypto/tools/provider-profile-check.py` (kontroly A1 až A11, k 10. 10. 2026 všetky prešli; rate limity a timeouty sa rátajú ako nepresvedčivý výsledok, nikdy nie ako úspech).

#	Požiadavka	Prečo
1	TLS 1.3, `TLS_AES_128_GCM_SHA256`, ECDHE na P-256; žiadny HelloRetryRequest, žiadne 0-RTT	2PC počíta presne toto odvodzovanie kľúčov; spoločný key share potrebuje sčítanie bodov, ktoré X25519 nemá
2	HTTP/1.1, `Connection: close`, `Accept-Encoding: identity`	jedna požiadavka na spojenie, záznamy sa mapujú na bajty, okolo údaja o spotrebe nie je kompresia
3	prijme klientsky KeyUpdate uprostred požiadavky	práve tam N odovzdáva štafetu C
4	nerobí nič, kým nepríde celé telo podľa `Content-Length`	N môže zadržať posledný záznam tela, kým sa neoverí π_C2S; dovtedy nič nebeží a nič nestojí
5	JSON: pri duplicitnom kľúči vyhráva posledný; neskorší objekt nahradí skorší; poškodený JSON a neznáme polia odmietne	verejný sufix prebije čokoľvek v súkromnej časti a súkromná časť nemôže nechať otvorený reťazec ani objekt, do ktorého by sufix dopadol
6	sufix vie nastaviť každý parameter, ktorý ovplyvňuje cenu (model, stropy vrátane reasoning tokenov, `n`, príznaky meniace cenu); ostatné polia meniace cenu sa odmietnu	závisí od nich rezervácia aj účet

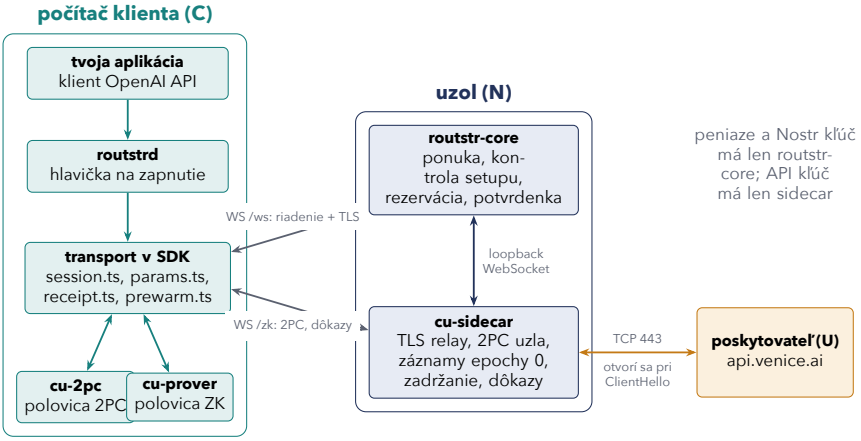
#	Požiadavka	Prečo
7	streamovaný údaj o spotrebe s počtom tokenov promptu a odpovede ešte pred koncom streamu	práve ten C odhalí a N podľa neho účtuje
8	nikdy neodzrkadlí hlavičku požiadavky (napríklad `Authorization`) do odpovede	odpoveď číta C; odzrkadlenie by mu kľúč priamo dalo (kontrola to netestuje)

Body 5 a 6 hovoria o každom možnom poli, čo žiadny konečný test nedokáže: kontrola ich iba vzorkuje a po zmenách u poskytovateľa ju treba spustiť znova. Zmena u poskytovateľa, ktorá rozbije fixovanie alebo hlásenie spotreby, spôsobí, že relácie zlyhajú. Nikdy kvôli nej neprejdú na neoverenú cestu.

Pri Venice zafixovaný sufix nastavuje aj `max_completion_tokens` (prebije `max_tokens`, toto obídenie sa našlo pri testovaní) a `venice_parameters` s vypnutým vkladaným systémovým promptom a vypnutým vyhľadávaním na webe (oboje mení účet).

6

Komponenty



Obr. 6.1: Procesy a kanály. routstr-core kopíruje bajty 2PC a dôkazov bez zmeny; zachytáva len niekoľko riadiacich rámcov.

Kanály sú WebSockety cez routstr-core, všetky pod `/v1/confidential/`:

Cesta	Čo ním ide
`ws`	riadiace rámce (JSON) a TLS záznamy (binárne); v query `session_id` (UUID) a `v=2`
`zk?proof=twopc`	bajty protokolu 2PC; `session_id` v query je `sid` protokolu
`zk?proof=zkchan`	predhriaty kanál pre dôkazy π_NK, π_C2S a π_C3S
`zk?proof=pi_nk`	záloha: jedno spojenie na každý dôkaz (rovnako `pi_c2s`, `pi_c3s`)

TCP spojenie na port 443 poskytovateľa U otvára sidecar až vo chvíli, keď príde ClientHello od C.

Ako sa to zapína: routstrd pošle požiadavku dôverne, ak nesie hlavičku `x-routstr-verify: confidential`. Funguje to iba pre `POST /v1/chat/completions`. Požiadavka, ktorá si dôverný režim vyžiadala, **zlyhá bezpečne**: nikdy ju potichu nepošle obyčajnou cestou. SDK ju smeruje len na uzly, ktoré majú pri danom modeli v `/v1/models` pole `confidential_upstream` a ktorých ponuka prejde overením.

7

Jedna relácia krok za krokom

V tejto kapitole sledujeme jednu požiadavku od ponuky až po potvrdenku. Šípky medzi C a sidecarom idú cez routstr-core, ktorý ich kopíruje bez zmeny; na diagramoch sú nakreslené priamo, aby bolo vidno, kto sa s kým naozaj rozpráva. Tyrkysovou je klient, tmavomodrou uzol, jantárovou poskytovateľ.

7.1 Krok 0: ponuka

Uzol zverejňuje podpísanú ponuku na `GET /v1/confidential/offer`. routstr-core ju skladá zo statického opisu sidecaru (`GET /offer` na sidecare: hostiteľ poskytovateľa, šablóna hlavičky, dĺžka a abeceda kľúča, kľúče sufixu, strop tokenov) a z vlastných cien pre jednotlivé modely. Podpisuje ju svojím Nostr kľúčom (BIP-340 Schnorr nad SHA-256 kanonického JSON-u: zoradené kľúče, žiadne biele znaky). Vymyslený, ale realistický príklad:

```
{
  "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}
```

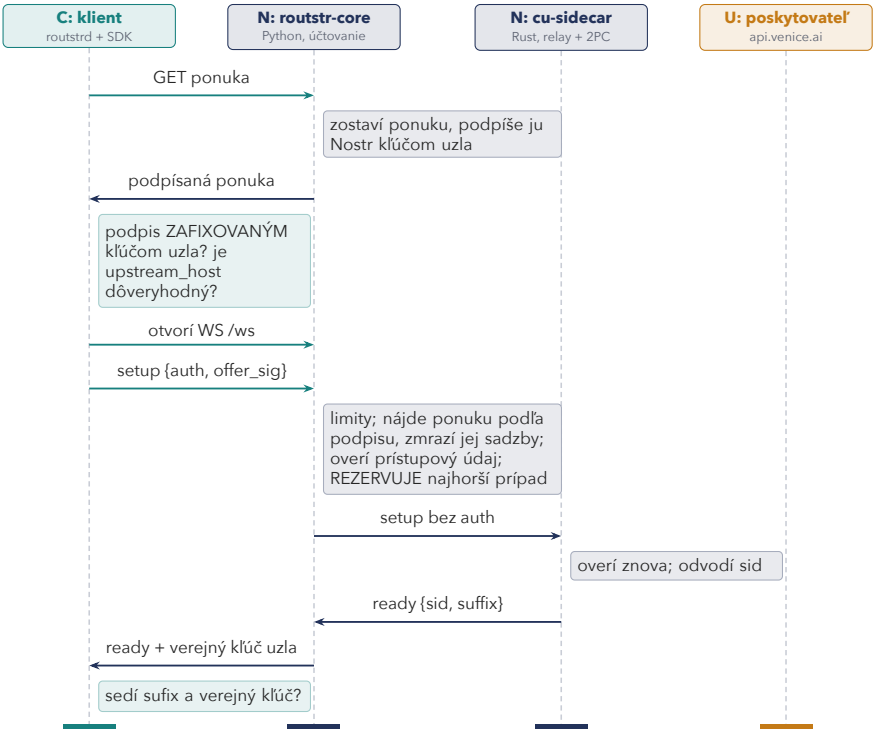
```
  },
  "mints": ["https://mint.example"],
  "abandon_fee_msat": 2000,
  "sig_payload": "{\"abandon_fee_msat\":2000,...}",
  "sig": "c03b...9f1e"
}
```

(Šablóna je jeden reťazec, zalomili sme ju len kvôli šírke strany.) Ceny sú v milisatoshi za 1 000 tokenov. `{KEY:42}` označuje 42 bajtov API kľúča, `{LEN}` dĺžku tela v desiatkovej sústave. Poplatky sa v ponuke objavia, len ak ich uzol účtuje. Klient:

1. berie všetky polia zo `sig_payload`, teda presne z podpísaných bajtov, nikdy z nepodpísanej obálky;
2. vyžaduje, aby podpis sedel **s kľúčom uzla, ktorý má zafixovaný** vo svojej konfigurácii (nestačí kľúč, ktorý uvádza samotná ponuka);
3. vyžaduje, aby `upstream_host` bol v jeho zozname dôveryhodných hostiteľov (prázdny zoznam odmietne každého);
4. vyžaduje podporovanú verziu a binárku dokazovateľa.

Pri každom modeli dostane výstup `/v1/models` navyše pole `confidential_upstream: {v, offer_url, ...}`, a to len pri modeloch, ktoré vie uzol oceniť.

7.2 Krok 1: setup a rezervácia



Obr. 7.1: Setup: kým neprejde kontrola prístupového údaja, nič sa nerezervuje, a sidecar prístupový údaj nikdy nevidí.

Klient otvorí WebSocket relácie a pošle:

```
{
  "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)"
}
```

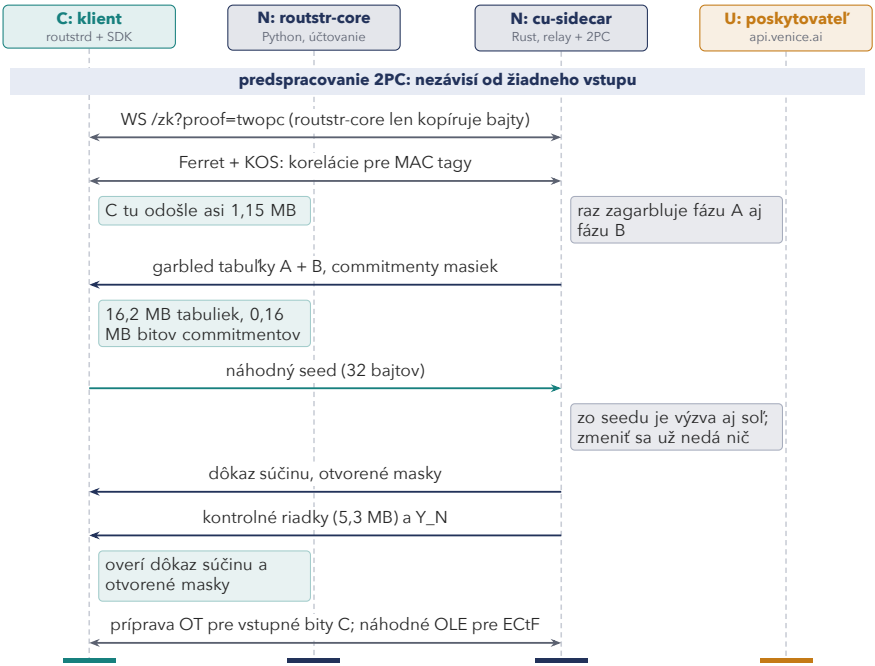
`len` je presná dĺžka JSON tela v bajtoch a odteraz sa nemení. routstr-core skontroluje tvar (`max_tokens` je celé číslo od 1 po strop, `len` od 1 po 8 MiB). Podľa `offer_sig` nájde ponuku medzi tými, ktoré podpísal za posledných desať minút, a zmrazí jej ceny pre túto reláciu (neznáma alebo expirovaná ponuka: HTTP 409 `offer_expired`, SDK si ju raz stiahne nanovo). Vyžaduje, aby klient zopakoval každý poplatok, ktorý ponuka účtuje (inak 409, takže klientovi, ktorý o poplatkoch nevie, sa nikdy žiadny nestrhne). Prístupový údaj overí bežnou funkciou uzla `validate_bearer_key` a cez `pay_for_request` **zarezervuje najhorší prípad** (kapitola 12). Až potom sa spojí so sidecarom a pošle mu setup **bez** `auth`, `offer_sig` a polí s poplatkami; pridá doň release id rezervácie ako `escrow`.

Sidecar setup znova overí a odvodí id relácie:

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

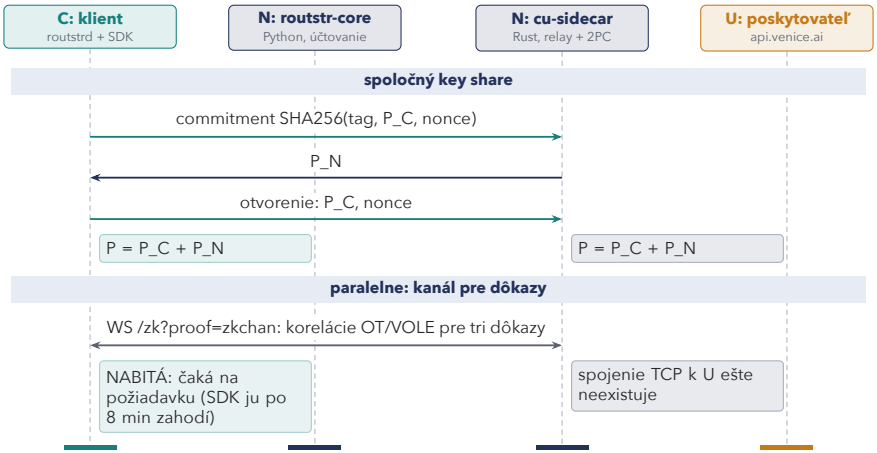
Späť príde `ready` so `sid`, `nonce_n`, `head_len`, kanonickým `suffix` a `v: 2`. routstr-core v ňom nahradí `pubkey` vlastným Nostr kľúčom. Klient skontroluje, že sufix sa zhoduje s tým, ktorý si vypočítal sám, a že `pubkey` je kľúč z podpísanej ponuky.

7.3 Krok 2: predspracovanie a spoločný key share (nabíjanie)



Obr. 7.2: Predspracovanie: drahá časť 2PC, ktorá nezávisí od vstupov. Každý jej bajt môže prebehnúť skôr, než nejaká požiadavka vôbec existuje.

Klient spustí `cu-2pc` a pripojí ho na `/zk?proof=twopc`. Uzol garbluje oba obvody, streamuje tabuľky, autentizuje klientovi masky svojich vodičov a spustí kontroly, ktoré nezávisia od žiadneho vstupu (jednotlivé správy vysvetľuje kapitola 8). Z N do C sa pritom prenesie asi 22 MB a z C do N 1,2 MB.



Obr. 7.3: Spoločný key share (najprv commitment, potom otvorenie) a kanál pre dôkazy.

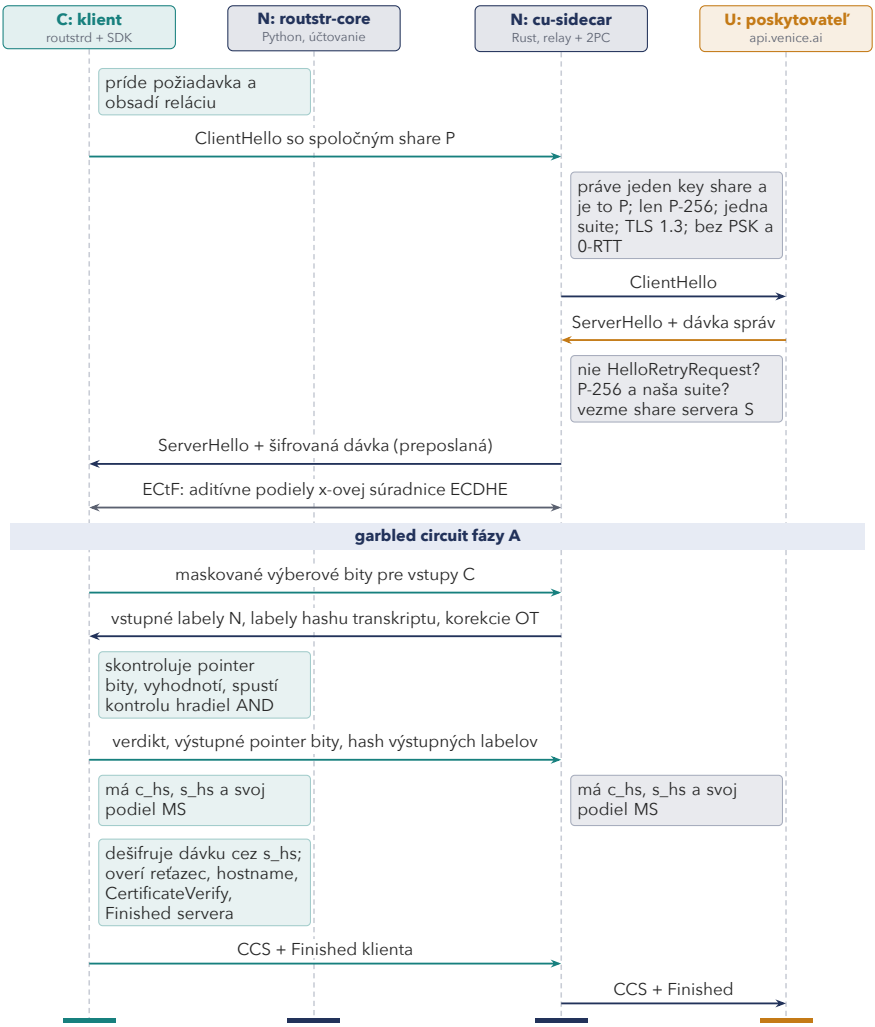
Spoločný key share je TLS key share, ktorý obe strany ponúknu spolu:

```
# 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)                 -->  overí commitment
P = P_C + P_N                         P = P_C + P_N
```

Commitment (záväzok) ide prvý, aby si C nemohol vybrať svoj bod až po tom, čo uvidí `P_N`. (Keby mohol, zvolil by `P_C` ako „bod, ktorého tajomstvo poznám, mínus `P_N`“ a spoločný bod by potom ovládal sám.) Klient zároveň otvorí `/zk?proof=zkchan`, dlho žijúci kanál pre dôkazy, ktorý pripraví náhodné korelácie pre všetky tri dôkazy relácie.

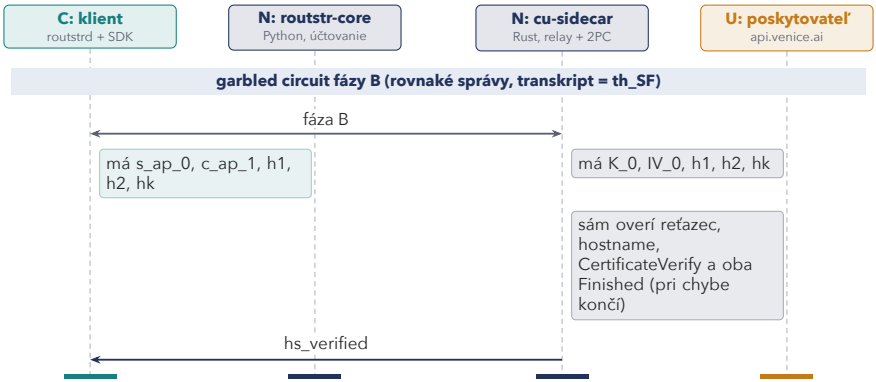
Relácia v tomto stave je **nabitá** (armed). Pri predhrievaní (kapitola 16) SDK nabíja relácie ešte predtým, než prídu požiadavky; studená požiadavka musí toto všetko urobiť priamo na ceste požiadavky.

7.4 Krok 3: TLS handshake s dvoma online fázami 2PC



Obr. 7.4: Handshake. Sidecar prepošle ClientHello, len ak jeho jediný key share je spoločný bod.

1. TLS fork na strane C zostaví ClientHello s jediným key share: spoločným bodom `P`.
2. **Sidecar ho rozparsuje a prepošle iba vtedy, keď** je jedinou cipher suite `TLS_AES_128_GCM_SHA256`, jedinou verziou TLS 1.3, `supported_groups` obsahuje len P-256, zoznam key shares má presne jednu položku a tá sa rovná `P`, kompresia je null, nie je tam PSK ani early data a žiadne rozšírenie sa neopakuje. Až teraz otvorí TCP spojenie na U.
3. U odpovie ServerHello a svojou zašifrovanou dávkou správ. Sidecar ServerHello skontroluje: nesmie to byť HelloRetryRequest, musí to byť TLS 1.3, ponúknutá suite a P-256 share. Vytiahne z neho share `S` poskytovateľa.
4. **ECtF** zmení `a_C` krát `S` a `a_N` krát `S` (dve polovice spoločného bodu) na aditívne podiely jeho x-ovej súradnice, čiže ECDHE tajomstva, bez toho, aby ho niektorá strana spoznala.
5. **Fáza A** garbled circuitu vypočíta handshake tajomstvá a master secret. Obe strany dostanú `c_hs` a `s_hs`; master secret z obvodu vyjde rozdelený na dva náhodne vyzerajúce podiely.
6. C dešifruje dávku od U pomocou `s_hs` a overí reťazec certifikátov voči koreňom WebPKI, meno hostiteľa, CertificateVerify a serverový Finished. Potom pošle svoj klientsky Finished vypočítaný z `c_hs`. Sidecar ho prepošle.



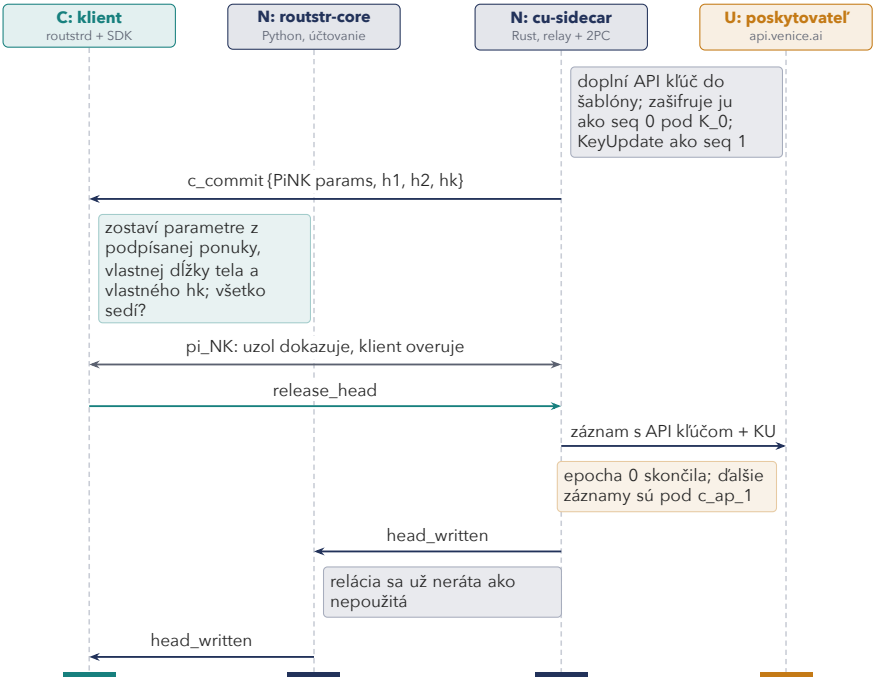
Obr. 7.5: Fáza B dá každej strane jej vlastné aplikačné kľúče; potom sidecar celý handshake overí sám.

7. **Fáza B** (vstup: podiely master secretu a `th_SF`) dá C `s_ap_0` a `c_ap_1`, N dostane `K_0` a `IV_0` a obe strany kotvy `h1`, `h2`, `hk`.
8. Sidecar spustí vlastné overenie s `c_hs` a `s_hs`, ktoré pri chybe zlyhá bezpečne: dešifruje dávku, zrekonštruuje transkript, overí reťazec voči koreňom Mozilly, meno hostiteľa, CertificateVerify a **obe** správy Finished. Skon-

troluje tiež, že C zatiaľ neposlal nič okrem ClientHello, CCS a Finished. Potom pošle `hs_verified`.

U musí dostať klientsky Finished v rámci svojho handshake timeoutu (CDN zatvára nečinný handshake asi po 15 s). Fáza A trvá lokálne asi 33 ms, takže rezerva je veľká.

7.5 Krok 4: epocha 0, záznam s API kľúčom



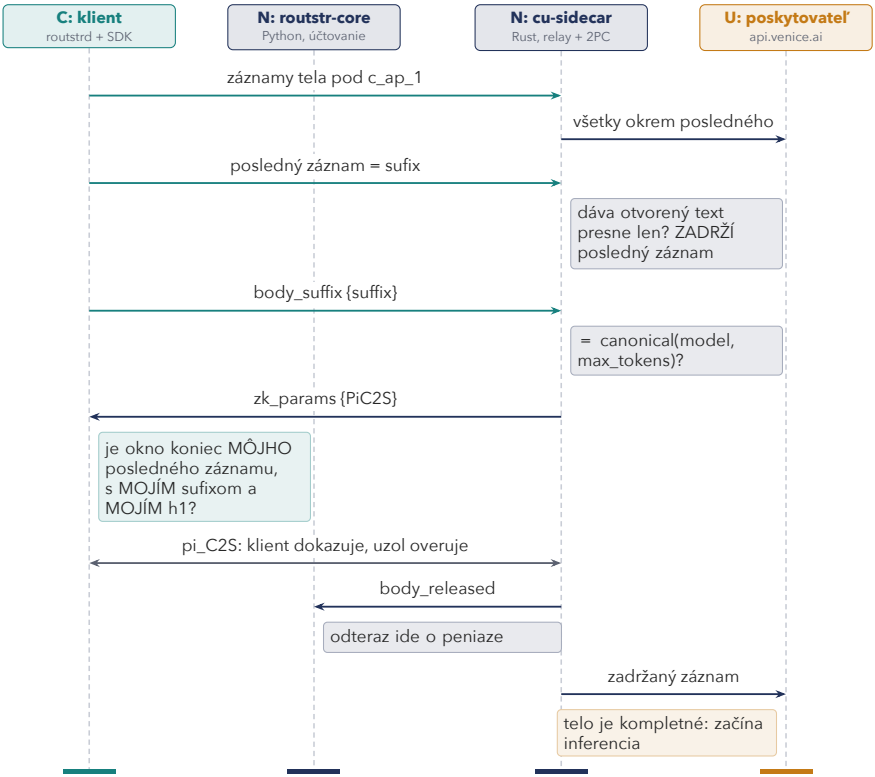
Obr. 7.6: Epocha 0. Uzol sa zaviaže k svojmu záznamu, dokáže, že zodpovedá šablóne, a zapíše ho, až keď klient dá súhlas.

Sidecar doplní do zverejnenej šablóny API kľúč a dĺžku tela a pod `K_0` zapečatí dva záznamy: sekvencia 0 je hlavička (vnútorný content type 0x17, application data), sekvencia 1 `KeyUpdate(update_not_requested)`. Klientovi pošle commitment k prvému záznamu a verejné parametre dôkazu 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]}}
}
```

Klient si každý z týchto parametrov znova zostaví z **podpísanej ponuky**, vlastnej dĺžky tela a vlastného `hk` a pri akomkoľvek rozdiele odmietne (pseudokód je v kapitole 9). Potom overí π_NK a pošle `release_head`. Sidecar zapíše oba záznamy na U jedným zápisom a odpovie `head_written`. Odteraz sa relácia pre limit na adresu už nepočíta ako „nepoužitá“.

7.6 Krok 5: telo a zadržanie záznamu



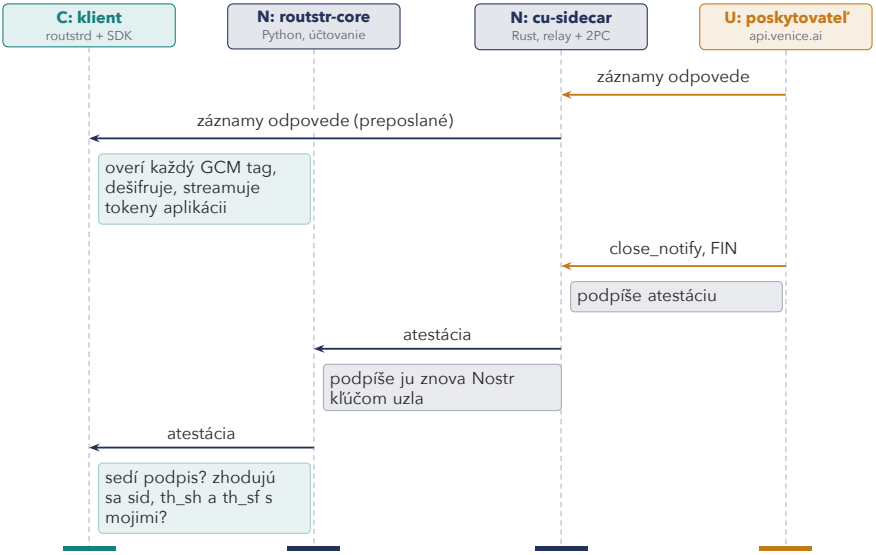
Obr. 7.7: Epocha 1. Záznam, ktorý dopĺňa telo, zostáva zadržaný, kým neprejde overenie dôkazu o sufixe.

C zapisuje telo pod `c_ap_1` od sekvencie 0: najprv súkromnú časť JSON-u (aj vo viacerých záznamoch), potom **zafixovaný sufix ako samostatný posledný záznam**. Sidecar počíta bajty otvoreného textu (dĺžka záznamu mínus 16 bajtov tagu mínus 1 bajt content type). Viac než `len` je chyba a záznam, ktorým súčet dosiahne presne `len`, sidecar **zadrží**. U má teraz neúplnú požiadavku a čaká (bod 4 profilu).

C pošle `body_suffix`. Sidecar vyžaduje, aby sa rovnal `canonical_suffix(model, max_tokens)` zo setupu, zo zadržaného záznamu zostaví parametre π_C2S, klient ich skontroluje a vytvorí dôkaz. Ak prejde, sidecar oznámi routstr-core `body_released` **ešte pred** zapísaním zadržaného záznamu, aby účtovanie uzla

vedelo, že poskytovateľ odteraz môže pracovať. Ak dôkaz neprejde, spojenie sa zavrie a žiadna inferencia neprebehla.

7.7 Krok 6: odpoveď



Obr. 7.8: Odpoveď prúdi ku klientovi; uzol atestuje, čo preposlal.

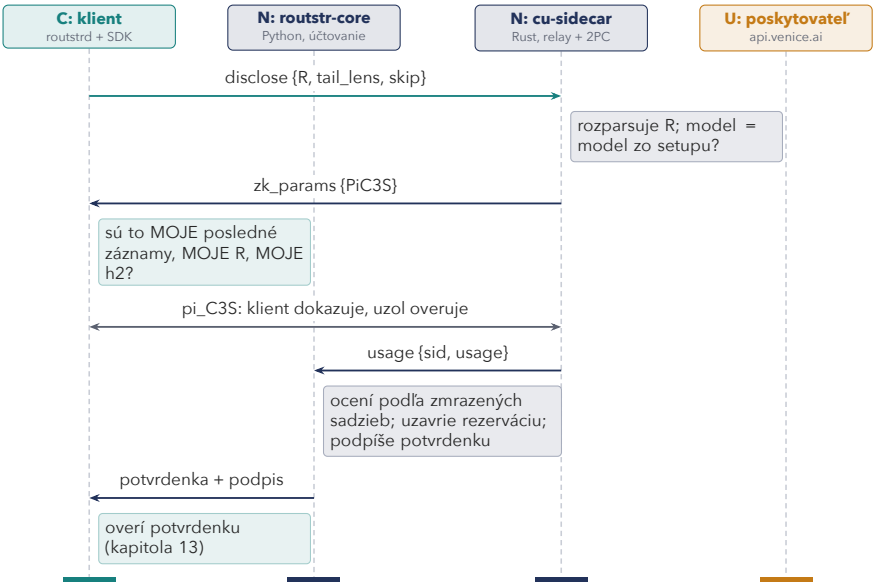
U streamuje odpoveď pod `s_ap_0`. Sidecar záznamy preposiela; C kontroluje každý GCM tag, dešifruje a tokeny posiela aplikácii hneď, ako prichádzajú. Keď U spojenie zavrie, sidecar podpíše atestáciu toho, čo preposlal, a routstr-core ju znova podpíše Nostr kľúčom uzla:

```
{
  "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 vyžaduje platný podpis uzla nad `a_json` a to, aby sa `sid`, `th_sh` a `th_sf` zhodovali s jeho vlastnými.

7.8 Krok 7: odhalenie a vyúčtovanie



Obr. 7.9: Odhalenie, dôkaz o spotrebe a vyúčtovanie.

C vyberie **R**: otvorený text od začiatku poslednej udalosti so spotrebou až po koniec streamu vrátane HTTP chunk framingu (pri HTTP chybe len hlavičku odpovede). Pošle `disclose {R, tail_lens, skip}`. Sidecar R rozparsuje (odstráni chunking; posledná udalosť `data:` pred `data: [DONE]` musí byť JSON s prázdnym `choices` a oboma počtami tokenov a jej `model` sa musí rovnať modelu zo setupu) a vráti parametre π_C3S: posledné záznamy servera, ktoré nájde od konca transkriptu. C overí, že sú to presne jeho posledné záznamy, a vytvorí dôkaz. Sidecar pošle overenú spotrebu do routstr-core, ten ju ocení zmrazenými cenami, uzavrie rezerváciu a vráti podpísanú potvrdenku (kapitola 13).

7.9 Kto čo vie v jednotlivých krokoch

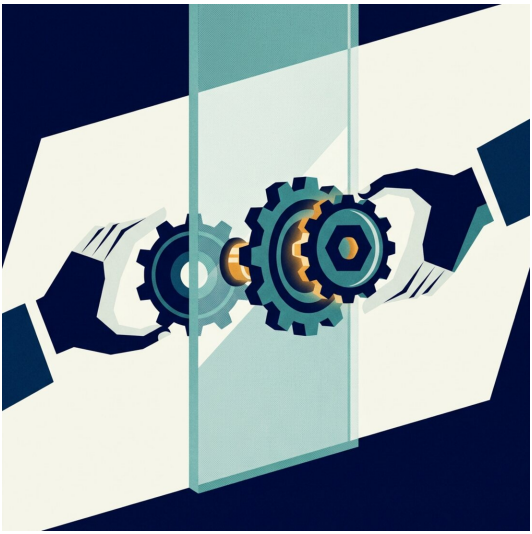
	C sa dozvie	N sa dozvie
setup	sid, nonce_n, zafixovaný sufix	model, max_tokens, dĺžku tela, prístupový údaj (na rezerváciu)
predspracovanie	garbled tabuľky, náhodne vyzerajúce labely a pointer bity	nič o vstupoch C (bity d sú zašifrované jednorazovým kľúčom)
key share	P_N, spoločný P	P_C, spoločný P
fáza A	c_hs, s_hs, svoj podiel MS r_C	c_hs, s_hs, svoj podiel MS
fáza B	s_ap_0, c_ap_1, h1, h2, hk	K_0, IV_0, h1, h2, hk
epocha 0	c_commit (hash šifrového textu), 1 bit: ⊓_NK platí	že C prijal ⊓_NK
telo	nič nové	dĺžku tela (známa), veľkosti záznamov, sufix, 1 bit: ⊓_C2S platí
odpoveď	celú odpoveď vrátane volaní nástrojov	len veľkosti záznamov a časovanie
vyúčtovanie	podpísanú potvrdenku	R (údaj o spotrebe alebo chybovú hlavičku), 1 bit: ⊓_C3S platí

Nikto sa nedozvie: ECDHE tajomstvo, HS, MS, c_ap_0. C sa nedozvie: K_0, IV_0, API kľúč. N sa nedozvie: c_ap_1, s_ap_0, prompt, odpoveď.

Obr. 7.10: Čo sa ktorá strana dozvie v ktorom kroku. U vidí obyčajnú TLS reláciu so všetkým, čo v nej je.

8

Dvojstranný výpočet



Cieľ: vypočítať odvodzovanie kľúčov TLS z obrázka 3.1 zo vstupov rozdelených medzi C a N a každý výstup doručiť len tej strane, ktorej patrí, aj keď druhá strana podvádza. 2PC pokrýva iba odvodzovanie kľúčov, spolu 31 kompresií SHA-256, nikdy nie samotný obsah. Preto má konštantnú cenu a dá sa zaplatiť ešte predtým, než nejaká požiadavka vôbec existuje.

8.1 Stavebné bloky ako API

Oblivious transfer (OT)

Vstup: odosielateľ má dve správy `m0`, `m1`, príjemca má výberový bit `b`. **Výstup:** príjemca dostane `m_b`. **Kto sa čo dozvie:** odosielateľ sa o `b` nedozvie nič, príjemca sa nedozvie nič o druhej správe. **Na čo slúži:** C dostane labely obvodu pre svoje vstupné bity bez toho, aby sa ich N dozvedel.

Korelovaný OT, KOS a Ferret: náhodnosť vyrobená vopred

Výstup: veľká dávka náhodných dvojíc s pevným skrytým vzťahom. Jedna strana drží tajný globálny posun (nazvime ho `Delta`) a ku každej položke náhodný kľúč `K`, druhá drží náhodný bit `x` a `M = K xor (x ? Delta : 0)`. **Cena:** pár drahých základných OT, z ktorých KOS a Ferret lacno natiahnu milióny korelácií. **Na čo slúži:** OT pre vstupné bity C a MAC tagy nižšie.

IT-MAC: tag, na ktorom sa klamúci uzol prezradí

Setup: C drží tajný 128-bitový `Delta_B` (127 náhodných bitov) a jeden kľúč `K` na každý otagovaný bit. N drží bit `x` a jeho tag `M`, pričom `M` sa rovná `K`, a ak je `x` 1, tak ešte XOR s `Delta_B`. **Záruka:** N môže neskôr `x` „otvoriť“ (poslať `x` aj `M`) a C tag skontroluje. Na otvorenie falošnej hodnoty by N musel poznať `Delta_B`, ktorý nikdy neuvidí. **Bonus:** tagy sú lineárne, XOR otagovaných bitov má za tag XOR ich tagov, a to zadarmo.

Commitment: zapečatená obálka

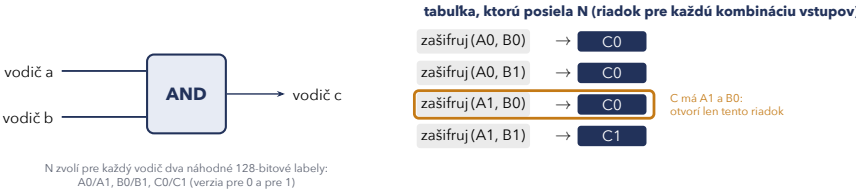
Vstup: hodnota a náhodný nonce. **Výstup:** `SHA256(tag | value | nonce)`. Po odoslaní hashu už hodnotu nezmeníš a hash o nej nič neprezradí, kým ho neotvoríš (nepošleš hodnotu aj nonce).

OLE a konverzia podielov (ECtF)

Vstup OLE: jedna strana má `a`, druhá `b`. **Výstup:** dve náhodne vyzerajúce čísla, jedno pre každú stranu, ktorých súčet je `a` krát `b`. **A2M / M2A** pomocou OLE prevádzajú „súčet rozdelený medzi dve strany“ na „súčin rozdelený medzi dve strany“ a späť. **ECtF** (EC to field) reťazí dve A2M a jednu M2A nad základným poľom P-256. C drží `a_C` krát `S`, N drží `a_N` krát `S` (dva body, ktorých súčet je ECDHE bod), a na konci majú čísla `x_C` a `x_N`, ktoré sa (modulo prvočíslo poľa) sčítajú na x-ovú súradnicu súčtu, čiže na ECDHE tajomstvo. Ani jeden ho nepozná. Rovnakú konverziu používa TLSNotary. Je semi-honest: podvodník vie len pokaziť výsledok, a to zachytí kontrola TLS Finished.

Garbled circuit: zašifrovaný program z vyhľadávacích tabuliek

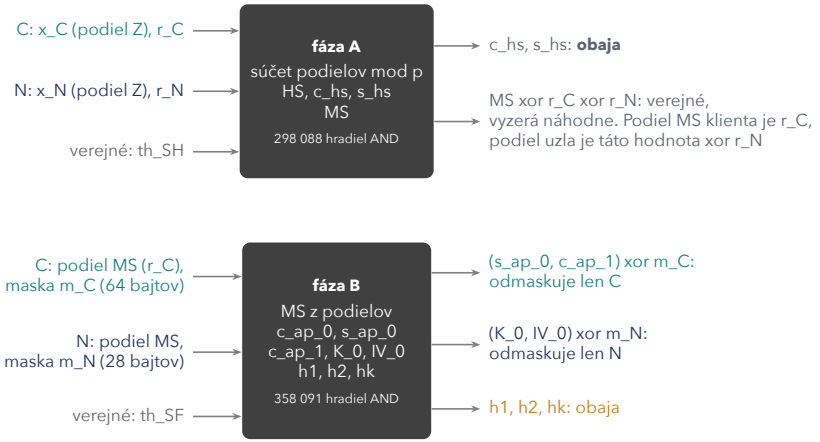
Vstup garblera (N): booleovský obvod (verejný) a tajný globálny posun `Delta_A`. **Výstup garblera:** pre každý vodič dva náhodné 128-bitové labely (jeden znamená 0, druhý 1, líšia sa o `Delta_A`) a pre každé hradlo AND malú zašifrovanú tabuľku. **Vstup vyhodnocovateľa (C):** jeden label na každý vstupný vodič. **Výstup vyhodnocovateľa:** jeden label na každý výstupný vodič plus dekódovacie bity, ktoré z výstupných labelov urobia hodnoty. **Kto sa čo dozvie:** C sa dozvie výstupy, ktoré smie dekódovať, a o medzivýsledkoch nič. **Háčik:** obyčajné garblovanie verí, že garbler zagarbloval správny obvod.



C dostane jeden label pre každý vstupný vodič, v každom hradle otvorí práve jeden riadok a získa jeden label výstupného vodiča. Label vyzerá náhodne, C nevie povedať, či znamená 0 alebo 1. Jeho posledný bit (**pointer bit**) je skutočná hodnota XOR tajná **maska** vodiča, ktorú pozná len N, a určuje, ktorý riadok má C otvoriť. Skutočné tabuľky sú komprimované (three-halves garbling: asi 197 bitov na hradlo AND) a hradlá XOR tabuľku vôbec nepotrebujú.

Obr. 8.1: Jedno garblované hradlo AND.

8.2 Čo obvody počítajú



Obr. 8.2: Dva obvody. Tyrkysové vstupy pochádzajú od C, tmavomodré od N, sivé sú verejné; výstupy sú zafarbené podľa toho, kto ich môže čítať.

Fáza A beží, keď je známy ServerHello od U. Sčíta dva podiely ECDHE tajomstva modulo prvočíslo P-256 (asi 770 hradiel AND) a potom vypočíta HS, kľúčované stavy HMAC pre HS, `c_hs`, `s_hs`, tajomstvo „derived“ a MS. `c_hs` a `s_hs` vychádzajú v čistom tvare, lebo ich potrebujú obe strany. MS vychádza XORované s náhodnou maskou od každej strany, `r_C` a `r_N`. Podiel C na MS je potom `r_C` a podiel N je „verejný výstup XOR `r_N`“. Každý podiel sám osebe je náhodný šum.

Fáza B beží po serverovom Finished. Z dvoch podielov znova poskladá MS a vypočíta `c_ap_0`, `s_ap_0`, `c_ap_1`, `K_0`, `IV_0` a tri kotvy. Súkromné výstupy opúšťajú obvod XORované s maskou, ktorú doň vložil ich vlastník: `s_ap_0` a `c_ap_1` pod 64-bajtovou maskou `m_C` od C, `K_0` a `IV_0` pod 28-bajtovou maskou `m_N` od N. Zamaskovaný výstup druhej strany môže vidieť každá strana, je to one-time pad. Kotvy vychádzajú v čistom tvare.

Každé HKDF-Expand-Label sa zmestí do jedného bloku SHA-256 a kľúčované stavy HMAC sa počítajú pre každé tajomstvo raz, takže fáza A má 14 kompresií (298 088 hradiel AND) a fáza B 17 (358 091 hradiel AND). Na reláciu to je 656 179 hradiel AND.

8.3 Ako sa C uistí, že uzol garbloval poctivo

Zlomyseľný garbler by mohol zagarblovať iný obvod, napríklad taký, ktorý skopíruje `c_ap_1` do výstupu N. Nasadená konštrukcia je **autentifikované garblovanie len na strane uzla**: garbluje iba N, C vyhodnocuje a kontroluje každé hradlo AND pomocou tagov na tajných maskách N.

Maska a pointer bit. Každý vodič má tajný bit masky, ktorý zvolil N (posledný bit 0-labelu od N). Label, ktorý C pre vodič drží, končí **pointer bitom**: skutočnou hodnotou vodiča XOR táto maska. C vidí pointer bity, nikdy nie masky, takže pointer bity mu pripadajú náhodné.

Predspracovanie, v poradí. Kód je `preprocess_n / preprocess_c` v súbore `crypto/twopc/src/lib.rs`.

1. Korelovaný OT s **C ako odosielateľom** (Ferret nad KOS; C drží `Delta_B`). N cez neho otaguje (IT-MAC) masku každého vstupu obvodu a každého výstupu hradla AND („základné vodiče“) a pre každé hradlo AND aj súčin jeho dvoch vstupných masiek, spolu asi 1,3 milióna otagovaných bitov. Masky ostatných vodičov vyplývajú z linearity. Za každú otagovanú hodnotu pošle N jeden korekčný bit.
2. N zagarbluje fázu A aj fázu B garblovaním three-halves a streamuje tabuľky (16,2 MB).
3. **Až keď' toto všetko dorazí**, pošle C 32-bajtový náhodný seed. Obe strany z neho odvodia náhodnú výzvu pre kontroly a soľ pre reláciu. N už nemôže nič zmeniť.
4. N dokáže všetky súčiny naraz (dávková kontrola v štýle QuickSilver: dva prvky poľa).
5. N otvorí spolu s tagmi masky vstupných vodičov C, vodičov s hashom transkriptu a všetkých výstupných vodičov. C ich potrebuje na kontrolu svojich vstupov a na dekódovanie výstupov.
6. N pošle pre každý základný vodič jeden **kontrolný riadok** (orezaný na 64 bitov) a hodnotu `Y_N`.
7. KOS OT s N ako odosielateľom pripraví prenos 1 280 vstupných bitov C; náhodné OLE pre ECtF.

Online, v každej fáze (asi 1,5 round tripu, 35 až 45 ms CPU):

```
# C, jedna fáza (zjednodušené z gc::run_check a phase_c)
send(choice_bits xor ot_randomness)        # N sa nedozvie nič
labels_N, labels_th, corrections = recv()  # vstupy N, 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)
# kontrola hradiel AND: jedna rovnica pre všetky hradlá
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))
```

Prečo kontrola funguje, intuitívne. Pri správne vyhodnotenom hradle AND spĺňajú pointer bity, ktoré C videl, commitnuté masky a commitnutý súčin masiek pevný XOR vzťah. Masky C nevidí, ale keďže tagy sú lineárne, vie len zo svojich kľúčov vypočítať, aký musí byť tag náhodnej kombinácie všetkých týchto vzťahov. Zodpovedajúcu hodnotu musí dodať N, lenže nevie, ktoré pointer bity C videl, takže ju nemôže vypočítať priamo. Každý kontrolný riadok je preto „vypchaný“ hashmi oboch labelov svojho vodiča, a to tak, že správny príspevok odomkne iba label, ktorý C naozaj drží. Ak čo i len jedno hradlo vyhodnotil C proti niečomu inému, než N commitol, výsledná hodnota sa líši o veličinu prenásobenú tajným `Delta_B` od C. N všetko zafixoval skôr, než poznal výzvu, a o `Delta_B` nevie nič, takže musí naslepo trafiť 64-bitovú hodnotu: šanca je asi jedna ku 9 triliónom na fázu.

Prečo soľ. Riadky putujú ako 64 bitov, čím sa ušetrí 5,3 MB. Uzol, ktorý by labely používal v ďalších reláciách znova, by si mohol jednorazovým výpočtom rádovo 2 na 64 hashov pripraviť falošný label, pri ktorom orezaný rozdiel hashov zmizne, a cez neho vyzradiť, ktorou cestou išlo vyhodnocovanie u C, bez toho, aby spôsobil prerušenie. Hash riadku je kľúčovaný soľou odvodenou zo seedu od C, ktorý N dostane až potom, čo commitne svoje tabuľky, posun aj kľúče, takže takýto predvýpočet nie je možný. Poctivý uzol labely znova nepoužíva tak či tak; soľ robí kontrolu korektnou aj voči uzlu, ktorý to robí.

Doručenie výstupov. C dekóduje svoje výstupy pomocou otvorených otagovaných masiek. Až potom pošle N výstupné pointer bity a hash svojich výstupných labelov. N si labely prepočíta zo svojich kľúčov, skontroluje hash a dekóduje svoje výstupy. C nemôže výstupy N zmeniť: na to by potreboval „neaktívny“ label (ten, ktorý nedostal), a na ten treba `Delta_A`. **Žiadny výstup sa nepošle U, nepoužije na šifrovanie ani neodhalí, kým tieto kontroly neprejdú.**

Prečo je to bezpečné pre klienta

Každá maska a každý súčin masiek sú zafixované pod tajným kľúčom C skôr, než výzva vôbec existuje. Nekonzistentné hradlo prejde kontrolou v každej fáze s pravdepodobnosťou zhruba jedna ku 2 na 63. Vstupné labely C aj labely hashu transkriptu sú pripnuté kontrolou pointer bitov proti otvoreným otagovaným maskám, takže N nemôže ani vymeniť vstupy C, ani podstrčiť iný transkript. Keď kontrola prejde, výstupy C sú správnou funkciou jeho vstupov, (efektívnych) vstupov N a skutočného transkriptu.

Prečo je to bezpečné pre uzol

C je vyhodnocovateľ. Vidí tabuľky, svoje aktívne labely, náhodne vyzerajúce pointer bity, masky hodnôt, ktoré už pozná alebo aj tak dekóduje, kontrolné riadky vypchané hashmi labelov, ktoré nedrží, a `Y_N`, ktorý by si vedel vypočítať sám. Súkromné výstupy N sa k C dostanú len XORované s `m_N`. API kľúč do 2PC vôbec nevstupuje.

8.4 Predspracovanie a online fáza

Krok	Kedy	C posiela	C prijíma	Čas (lokálne)
predspracovanie (tabuľky, tagy, kontroly, setup OT, OLE)	pred ClientHello; dá sa predhriať	1,21 MB	22,07 MB	0,55 s v jednom procese, 0,70 až 0,84 s naživo
spoločný key share	tiež predhriaty	41 B	74 B	2 ms
ECtF	po ServerHello	0,2 kB	12 kB	5 ms
fáza A	po ServerHello	73 B	8 kB	asi 33 ms
fáza B	po serverovom Finished	238 B	24 kB	asi 40 ms

Z 22 MB, ktoré C stiahne, je 16,2 MB garbled tabuliek a 5,3 MB kontrolných riadkov. Z 1,21 MB, ktoré odošle, pripadá 1,15 MB na jeho časť Ferretu (C je pri Ferrete odosielateľ). Pri round-trip čase 80 ms trvá online časť asi pol sekundy: key share 84 ms, ECtF 170 ms, fáza A 121 ms, fáza B 126 ms. Garbled circuit sa dá použiť len raz, takže predspracovanie sa medzi požiadavkami nikdy nedá zdieľať. Predhrievanie ho len presunie mimo cesty požiadavky.

8.5 Únik: najviac 2,3 bitu na reláciu

Podvádzajúci uzol môže urobiť jednu vec, ktorej kontroly nebránia: zariadiť, aby prerušenie **záviselo od tajných hodnôt C**. Napríklad pokazí jeden riadok tabuľky, ktorý C otvorí, len ak je istý tajný bit 1. Ak C preruší, N sa ten bit dozvie. Volá sa to selective failure.

Koľko môže uniknúť? V každej fáze N pozoruje jeden z troch výsledkov: C prerušil pri kontrole; hlásenie výstupov od C bolo odmietnuté (taký výsledok si N vie zariadiť); úspech. Relácia sa končí pri prvom zlyhaní, takže celá relácia má najviac päť rozlíšiteľných výsledkov:

```
fáza A: C preruší                        -> výsledok 1
fáza A: hlásenie odmietnuté              -> výsledok 2
fáza A ok, fáza B: C preruší             -> výsledok 3
fáza A ok, fáza B: hlásenie odmietnuté   -> výsledok 4
obe fázy ok                              -> výsledok 5
```

Päť výsledkov nesie najviac log2(5), teda asi 2,3 bitu.

Prečo sa to na nič nenazbiera. Tie bity sa týkajú vstupov C **do danej relácie**: jeho ECDHE podielu, jeho podielu na master secret, jeho masiek. Všetko sú to čerstvé náhodné hodnoty vytiahnuté pre jednu reláciu a nikdy nepoužité znova, a prerušená relácia sa zahodí. Predstav si hru na dvadsať otázok, kde v jednej hre dostaneš najviac dve či tri odpovede áno/nie a každá hra má nové tajomstvo: k ničomu sa nepriblížiš. V najhoršom prípade (všetky kontroly prejdú) môže N poznať asi dva bity 512-bitovej masky, ktorá skrýva dva 32-bajtové traffic secrets od C. AES-128 kľúče z nich odvodené si zachovajú plnú silu. API kľúč nikdy nie je vstupom 2PC a uzol, ktorý naschvál prerušuje relácie, si jeho klienti všimnú.

Vstupy sa nikdy neotvárajú. Každý súkromný vstup vstupuje len ako garblovaný vstup alebo vstup OLE a potom sa nič neodhaľuje. Návrhy ako DEAP z TLSNotary alebo Garble-then-Prove získajú bezpečnosť voči zlomyseľnému protivníkovi lacno tak, že jedna strana po relácii otvorí svoje podiely. Tu by to dalo C do rúk master secret, z neho `K_0` a z neho API kľúč.

> **Čo je dokázané a čo len zdôvodnené**
>
> Kontrola súčinov v štýle QuickSilver a otvárania IT-MAC sú štandardné. Garblovanie three-halves má vlastný dôkaz (Rosulek a Roy), no táto implementácia bola znova postavená zo štruktúry schémy a jej matice vyhodnocovateľa musí s článkom porovnať recenzent. Kontrola hradiel AND s maskami, ktoré drží garbler, orezanie na 64 bitov a soľ sú zdôvodnené v dokumentácii projektu, ale **nemajú publikovaný dôkaz** a konštrukcia zatiaľ neprešla externou revíziou. Pri Ferrete a KOS sa predpokladá bezpečnosť voči zlomyseľnému príjemcovi (tu N). ECtF je semi-honest.

9

Dôkazy s nulovou znalosťou

9.1 Nástroj

ZK dôkazy nad VOLE (QuickSilver z mpz2)

Vstup: booleovský obvod (tu poskladaný z kompresií SHA-256, blokov AES-128, XOR-ov a gadgetu na porovnanie bajtov), verejné vstupy, ktoré poznajú obe strany, a tajný svedok, ktorého pozná len dokazovateľ. **Výstup pre overovateľa:** iba hodnoty, ktoré sa tvrdenie rozhodne dekódovať. V našich tvrdeniach je to niekoľko 32-bajtových hashov, ktoré sa musia rovnať verejným kotvám, a jeden bit rovnosti za každé porovnanie s plaintextom. **Záruka:** dokazovateľ nedokáže presadiť nepravdivé tvrdenie (okrem zanedbateľnej pravdepodobnosti) a overovateľ sa o svedkovi nedozvie nič nad rámec dekódovaných hodnôt. **Forma:** interaktívny dôkaz s určeným overovateľom: presvedčí druhú stranu, teraz, a nikoho iného. Beží na vopred pripravených OT/VOLE koreláciách (základný OT Chou-Orlandi, KOS, Ferret).

Každé porovnanie s plaintextom prebieha **vnútri** obvodu. Keď klient dokazuje, že „tento ciphertext sa dešifruje na sufix“, uzol dostane jeden bit, nikdy nie dešifrované bajty, a to ani vtedy, keď dôkaz zlyhá.

9.2 π_NK: záznam uzla zodpovedá zverejnenej šablóne

dokazovateľ	uzol (sidecar), overovateľ: klient
tajné	`K_0`, `IV_0` (28 bajtov) a bajty API kľúča
verejné	`hk`; šablóna s nulovými bajtmi na pozíciách kľúča a vyplneným `Content-Length`; offsety a dĺžka kľúča; abeceda `[A-Za-z0-9_-]`; 5-bajtová hlavička záznamu; `c_commit`
dekódované	`hk`, `c_commit` (oba sa porovnajú s verejnými hodnotami), jeden bit pre abecedu

Čo obvod kontroluje, zapísané ako kód:

```
def pi_nk(K0_IV0, key):                    # tajné
    assert sha256(K0_IV0) == hk
    head = template.copy()
    for i, off in enumerate(key_offsets):
        head[off] = key[i]                 # vloženie kľúča
    plain = head + b"\x17"                 # vnútorný content type TLS
    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)
```

Čo z toho má klient. Záznam, ku ktorému sa uzol zaviazal, je presne zverejnená hlavička požiadavky (rovnaká cesta, rovnaké hlavičky, deklarovaný `Content-Length`) so 42 znakmi kľúča v slote pre kľúč, zašifrovaná pod `K_0` z 2PC (viaže ju `hk`, ktoré C dostal z vlastného behu 2PC). V zázname, ku ktorému sa zaviazal, teda uzol nemôže presmerovať požiadavku na iný endpoint ani pridať hlavičku.

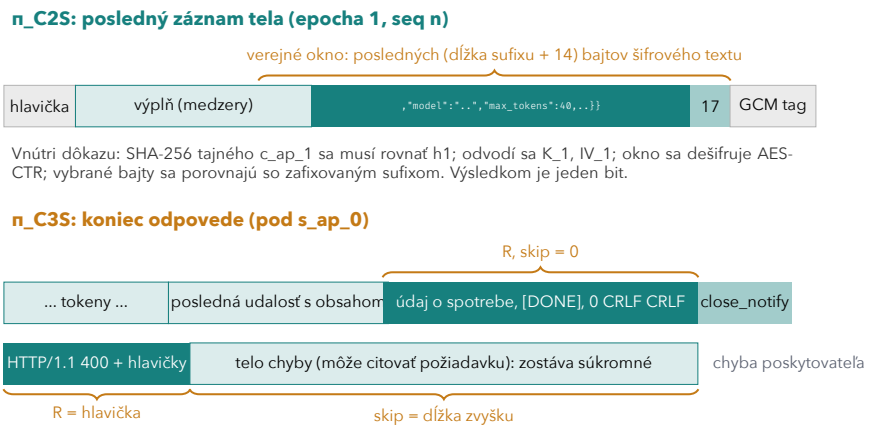
Čo z toho nemá. Bajty, ktoré uzol naozaj zapíše smerom k U, klient nikdy nevidí, a uzol drží `K_0`: mohol by zapísať iný záznam alebo ich zapísať viac. Prečo to klientove dáta ohroziť nemôže, rozoberá sekcia 11.4.

9.3 π_C2S: telo končí zafixovaným sufixom

dokazovateľ	klient, overovateľ: uzol
tajné	`c_ap_1` (32 bajtov)
verejné	`h1`; poradové číslo posledného záznamu tela; okno ciphertextu na jeho konci a offsety okna; očakávaný sufix
dekódované	`h1` (porovná sa), jeden bit rovnosti

```
def pi_c2s(c_ap_1):                          # tajné
    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
```

Okno tvorí posledných `len(suffix) + 14` bajtov ciphertextu záznamu (bez tagu) a vybrané bajty končia tesne pred bajtom vnútorného content type, takže dôkaz potvrdzuje, že sufix je **koniec** tela. Keďže `h1` je verejný výstup toho istého 2PC, z ktorého vzišli kľúče, klient nemôže dokazovať voči tajomstvu podľa vlastného výberu. Dôkaz stojí 7 kompresií SHA-256 a zhruba 15 blokov AES: lokálne 0,16 s, cez predhriaty kanál pre dôkazy 0,04 s.



Obr. 9.1: Na čo sa pozerajú π_C2S a π_C3S.

9.4 π_C3S: odhalená spotreba pochádza priamo od poskytovateľa

dokazovateľ	klient, overovateľ: uzol
tajné	`s_ap_0` (32 bajtov)
verejné	`h2`; posledné záznamy servera (ciphertext bez tagov, s poradovými číslami), vzaté z vlastnej kópie transkriptu, ktorú má uzol; R; `skip`
dekódované	`h2` (porovná sa), jeden bit rovnosti

```
def pi_c3s(s_ap_0):                          # tajné
    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
```

Pri odhalení spotreby je `skip` nula: R končí presne tam, kde končí odpoveď. Pri chybe poskytovateľa je R len hlavička odpovede a `skip` je dĺžka zvyšku, takže telo chyby zostane súkromné, no hlavička je ukotvená v známej vzdialenosti od konca.

Prečo sa R nedá podvrhnúť výstupom modelu. Údaj o spotrebe je surová SSE udalosť, pred ktorou je HTTP chunk framing (`<hex size> CRLF`). Výstup modelu

sa vždy objaví len vnútri JSON reťazcov, kde sú CR a LF escapované, takže žiadny výstup modelu nevyrobí hranicu udalosti ani chunk framing. Projekt to otestoval tak, že model žiadal o vypísanie falošných údajov o spotrebe: každá imitácia zostala escapovaná vnútri reťazca. Rovnaký argument platí pre hlavičku chyby: striktný stavový riadok a za ním riadky hlavičiek v tvare `Name: value` nemôžu vzniknúť z výstupu modelu ani z veľkostí chunkov.

9.5 Prečo uzol nemôže dôkazy zneužiť ako dešifrovacie orákulum

Verejné parametre π_C2S a π_C3S volí uzol. Podvádzajúci uzol by sa mohol pokúsiť namieriť dôkaz na iný ciphertext a z jedného bitu zistiť, či nejaký odhad plaintextu sedí. Klient preto **pred dokazovaním** skontroluje každý parameter voči vlastným záznamom (`client/confidential/params.ts`):

```
def check_pi_c2s(p, my_last_record, my_suffix, my_h1, my_seq):
    assert p.h1 == my_h1                        # kotva z MÔJHO 2PC
    payload = my_last_record[5:]                # bez hlavičky
    ct_len  = len(payload) - 16                 # bez tagu
    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:                     # jeden close_notify
        recs = recs[:-1]                        # môže byť na konci
    mine = recs[-len(p.records):]
    assert seqs_contiguous(p.records)
    for theirs, ours in zip(p.records, mine):
        assert theirs.ct == ours[:-16]          # presne moje záznamy
    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]
```

Sidecar to robí symetricky: každý dôkaz spustí len voči **vlastným** uloženým parametrom pre daný názov dôkazu a na výsledok reaguje, iba ak sú parametre, s ktorými dôkaz bežal, bajt po bajte zhodné s uloženými.

9.6 Kanál pre dôkazy

Každý dôkaz potrebuje čerstvé OT/VOLE korelácie. Ak sa generujú pre každý dôkaz zvlášť, stojí to asi dvanásť round tripov a až 1,5 MB: cez domáci uplink 5,4 s pre π_NK a 3,0 s pre π_C2S. Každá relácia si preto, kým je nabitá (armed), otvorí jeden dlhodobý kanál (`/zk?proof=zkchan`) a korelácie pre oba smery vygeneruje vopred: reťazec „nk“ (dokazuje N, tajnú `Delta` drží C) a reťazec „cp“ (C dokazuje π_C2S a π_C3S, `Delta` drží N). Pravidlá: pre každú reláciu čerstvá `Delta`, každá korelácia sa použije raz, každý dôkaz sa obslúži najviac raz, neúspešný dôkaz kanál otrávi, takže podvodník nemôže skúšať dvakrát, a v rámci relácie beží naraz iba jeden dôkaz. Bez kanála sa klient vráti k jednému WebSocketu na každý dôkaz.

10

Zafixovanie požiadavky: sufix a zadržanie

10.1 Sufix

Sidecar aj klient počítajú z modelu a limitu tokenov zo setupu ten istý sufix, bajt po bajte (`canonical_suffix` v `relay/src/main.rs`, `canonicalSuffix` v `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"}}
```

(Na linke ide ako jeden riadok; pre tento model a limit má 217 bajtov.) Začína čiarkou a končí zatvorením objektu najvyššej úrovne. SDK skladá telo požiadavky takto:

```
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]       # zahodí koncovú }
    suffix = canonical_suffix(model, max_tokens)
    if prefix == "{":                       # ziadne súkromné kľúče
        return "{" + suffix[1:]
    return prefix + suffix
```

`max_tokens` je hodnota volajúceho orezaná na limit z ponuky (a ak ju volajúci nezadá, berie sa priamo limit). Predhriate relácie majú pevne dané `len`, takže

kratšie telo sa medzi súkromnou časťou a sufixom doplní medzerami. JSON biele znaky pred čiarkou ignoruje.

10.2 Prečo rozhoduje sufix

Pri sémantike poskytovateľa (bod 5 profilu) vyhráva **posledný** výskyt kľúča na najvyššej úrovni a neskorší objekt nahradí skorší. Nech teda súkromná časť o `model` alebo `max_tokens` tvrdí čokoľvek, platia hodnoty zo sufixu. Parsovanie je striktné (poškodený JSON a neznáme polia poskytovateľ odmietne), takže súkromná časť nemôže nechať otvorený reťazec ani objekt, do ktorého by sufix spadol ako obyčajné dáta.

10.3 Zadržanie

Sidecar vynucuje, aby plaintext epochy 1 mal presne `len` bajtov. Vďaka tomu vie, ktorý záznam telo dokončuje, a zadrží ho, kým neprejde π_C2S. Poskytovateľ nezačne nič robiť, kým nedostane celé telo (bod 4 profilu). Keď teda dôkaz zlyhá, poskytovateľ má v rukách neúplnú požiadavku, spojenie sa zavrie, žiadna inferencia nebežala a okrem prípadného poplatku za zrušenie sa nič neúčtuje.

Útok: klient uvedie v setupe lacnejší model a v tele drahší

Prečo to nevyjde: sufix s modelom zo setupu je dokázateľne koncom tela a posledný kľúč vyhráva, takže poskytovateľ spustí model zo setupu. Telo, ktoré sufix vynechá, nikdy nebude úplné, a poskytovateľ teda vôbec nezačne. Adversariálny test (`crypto/tools/adversarial-model-swap.sh`) skúša presne toto. K tomu ešte návnadový model v súkromnej časti (vyúčtuje sa ako model zo setupu) a telo, ktoré končí sufixom iného modelu, no tvári sa, že nesie kanonický (π_C2S zlyhá).

Útok: pripojiť za telo druhú požiadavku

Prečo to nevyjde: sidecar odmietne každý bajt za deklarovanou dĺžkou aj každý záznam po tom zadržanom. `Connection: close` a pevný `Content-Length` v šablóne nenechávajú na prepašovanú požiadavku žiadne miesto.

Útok: rozdeliť sufix do dvoch záznamov

Prečo to nevyjde: π_C2S pokrýva len okno v poslednom zázname. Rozdelený sufix sa dokázať nedá, zadržaný záznam sa nikdy neuvoľní a relácia skončí bez inferencie.

Útok: zvýšiť v súkromnej časti limit tokenov alebo zapnúť vyhľadávanie na webe

Prečo to nevyjde: `max_tokens`, `max_completion_tokens` aj `venice_parameters` sú v sufixe, a ten vyhrá. Ďalšie polia, ktoré menia cenu (`service_tier`, `web_search_options`, `prediction`), poskytovateľ odmieta (overuje to vzorka v teste profilu). Obrázkové časti odmieta každý ponúkaný model (`crypto/tools/regression-modality.py`) a rezervácia aj tak pokrýva celé kontextové okno.

11

Vlastnosť po vlastnosti

11.1 P1: klient nemôže ukradnúť API kľúč

V protokole existuje kľúč iba ako plaintext záznamu uzla v epoche 0, zašifrovaný pod `K_0`. C nikdy nezíska `K_0`, `IV_0` ani žiadne tajomstvo, z ktorého sa odvodzujú (Z, HS, MS, `c_ap_0`): 2PC ich nedá nikomu, prípadne len N pod maskou N, a neskôr sa nič neotvára. Tajomstvá, ktoré C dostane (`c_hs`, `s_hs`, `s_ap_0`, `c_ap_1`), sú jednosmerné výstupy HKDF a ani jedno z nich nie je predkom `K_0`.

Útok: nahrať odchádzajúcu prevádzku uzla a dešifrovať záznam s API kľúčom

Prečo to nevyjde: šifrový text je pod `K_0` a ten C nemá. Práve toto pridáva protokol v2 oproti v1: vo v1 klient poznal celý kľúč epochy 1, takže ktokoľvek, kto videl odchádzajúcu prevádzku uzla a spolčil sa s klientom, mohol API kľúč prečítať.

Útok: dať do ClientHello vlastný key share klienta

Prečo to nevyjde: sidecar prepošle ClientHello, len ak nesie presne jeden key share a ten sa rovná spoločnému bodu. S druhým, vlastným key share by C dokončil handshake sám a poznal by všetky tajomstvá. Adversariálny test tento prípad skúša a sidecar ho odmietne skôr, než čokoľvek dorazí k poskytovateľovi.

Útok: zvoliť bod klienta až po tom, čo uvidí bod uzla

Prečo to nevyjde: C sa k svojmu bodu zaviaže skôr, než uvidí bod N. Inak by si mohol bod vybrať tak, aby spoločný bod bol taký, ku ktorému pozná tajomstvo.

Útok: sfalšovať 2PC výstupy uzla alebo ich vyčítať z obvodu

Prečo to nevyjde: zmeniť to, čo N dekóduje, si vyžaduje label, ktorý C nikdy nedostal, a na ten treba `Delta_A`. Výstupy N idú cez C iba XORované s náhodnou maskou N. API kľúč do 2PC ako vstup vôbec nevstupuje.

Útok: uhádnuť kľúč hrubou silou z c_commit alebo hk

Prečo to nevyjde: `c_commit` je hash šifrového textu, takže na overenie odhadu treba `K_0`. `hk` je hash 28 rovnomerne náhodných bajtov. π_NK odhalí len tieto hashe a jeden bit o abecede.

Zostávajúce predpoklady: poskytovateľ nikdy nevracia hlavičku požiadavky späť (bod 8 profilu) a ani proces uzla, ani poskytovateľ nie sú kompromitované.

11.2 P2: uzol nemôže čítať prompt ani odpoveď

Telo požiadavky ide pod `c_ap_1` a odpoveď pod `s_ap_0`. Oba kľúče dá 2PC iba C, zamaskované maskou C, a výstupy N nie sú predkom ani jedného z nich. Uzol, ktorý garbluje nesprávny obvod (napríklad taký, čo vypustí `c_ap_1` do jeho vlastného výstupu), odhalí kontrola AND hradiel na strane C skôr, než C použije akýkoľvek kľúč. Uniknúť to môže s pravdepodobnosťou zhruba 2 na mínus 62 na reláciu a cez prerušenia sa uzol o čerstvých hodnotách C dozvie najviac asi 2,3 bitu. Dôkazy, ktoré dešifrujú pod kľúčmi C, vracajú jeden bit nad parametrami, ktoré si C overil. V plaintexte sa N dozvie len to, čo je zámerne verejné: sufix, údaj o spotrebe, prípadne pri chybe hlavičku odpovede.

11.3 P3: uzol nemôže zmeniť požiadavku ani odpoveď

Prečo sa odpoveď nedá zmeniť, ani volania nástrojov

Každý záznam odpovede je zapečatený cez AES-GCM pod `s_ap_0`, ktorý má len C (a U). TLS stack na strane C kontroluje každý tag. Prevrátený bajt, vložený záznam, prehodené poradie záznamov (sekvenčné číslo je súčasťou nonce) aj záznam prehraný z inej relácie (každá relácia má čerstvé kľúče) neprejdú kontrolou tagu a požiadavka skončí chybou. Volania nástrojov sú len JSON v tom istom streame a chránené sú rovnako. Uzol môže stream nanajvýš prerušiť. C potom nemá úplnú odpoveď ani údaj o spotrebe a požiadavka viditeľne zlyhá.

Telo požiadavky je zapečatené pod `c_ap_1`, ktorý N nemá, takže ho nevie prečítať ani zmeniť. O sufixe je dokázané, že tvorí koniec tela, takže platia dohodnuté parametre. Po prijatí odpovede C ešte skontroluje, či sa model uvedený v odpovedi zhoduje s požadovaným.

11.4 Zvyškové riziko: epochu 0 ovláda uzol

N má `K_0`, takže môže zapísať ľubovoľné záznamy epochy 0, nielen dokázanú hlavičku. π_NK viaže obsah záznamu, ku ktorému sa N zaviazal, nie bajty, ktoré naozaj zapíše (C ich nikdy nevidí). N by mohol použiť iný účet alebo endpoint, prípadne pridať hlavičku, napríklad takú, čo zapne uchovávanie dát u poskytovateľa. C to odhaliť nevie. Škoda je však ohraničená: epocha 0 nenesie žiadne dáta C, vlastné záznamy C zostávajú autentifikované pod `c_ap_1`, odpoveď prečíta iba C a pokazená požiadavka sa prejaví ako zlyhaná odpoveď. Svoj vlastný účet u poskytovateľa môže uzol zneužiť vždy. Protokol zaručuje to, že obsah C nemôže prečítať ani zmeniť.

11.5 P4: účet je číslo od samotného poskytovateľa

Účtovaná spotreba je podreťazec autentifikovanej odpovede, dokázaný pod serverovým tajomstvom relácie (π_C3S) a ukotvený na konci transkriptu, ktorý uzol sám preposielal. `h2` je výstup toho istého 2PC, ktoré vyrobilo kľúče, a N ho prijme len cez autentifikované hlásenie výstupov od C. C teda nemôže dokazovať voči tajomstvu podľa vlastného výberu bez toho, aby prelomil SHA-256. N odvodí sufix z hodnôt zo setupu, prijme iba tento sufix a celú požiadavku uvoľní až po π_C2S, takže U spustí presne rezervovaný model s rezervovaným limitom.

Útok: klient odhalí skorší, menší údaj o spotrebe

Prečo to nevyjde: pri odhalení spotreby musí byť `skip` rovné 0. R musí končiť presne na konci odpovede (nasledovať smie už len koncový close_notify) a sidecar pri parsovaní R vyžaduje, aby končilo `data: [DONE]` a posledným chunkom.

Útok: klient odhalenie zatají

Prečo to nevyjde: keď už celá požiadavka dorazila k U, chýbajúce odhalenie stojí celú rezerváciu. Odhaliť spotrebu je teda vždy v záujme klienta.

Útok: klient tvrdí, že nastala chyba poskytovateľa

Prečo to nevyjde: odhalenie chyby sa musí dať naparsovať ako striktný stavový riadok HTTP (400 až 599), za ktorým nasledujú korektné riadky hlavičiek, a π_C3S dokazuje, kde presne v skutočnej odpovedi leží. Výstup modelu nedokáže vyrobiť surové CR/LF, takže falošná hlavička sa v odpovedi 200 schovať nemôže.

11.6 P5: poskytovateľa overujú obe strany

C vo svojom TLS stacku overí reťazec certifikátov až po koreňové certifikáty WebPKI, hostname (RFC 6125), CertificateVerify a serverový Finished, a to voči hostiteľovi poskytovateľa, ktorého si zafixoval ako dôveryhodného. Sidecar overí to isté z `c_hs` a `s_hs` a navyše obe správy Finished. Robí to fail-closed a skôr, než pod `K_0` čokoľvek zapíše. Klient validuje cestu certifikátov podľa RFC 5280, ale bez kontroly revokácie.

11.7 P6: čerstvosť

Identifikátor relácie je hash čerstvých nonce od oboch strán a jednorazového identifikátora rezervácie uzla. TLS randomy, skaláre ECDHE, masky, garbled tabuľky, oba globálne offsety, MAC kľúče, salt aj korelácie kanála pre dôkazy sa pre každú reláciu generujú nanovo.

12

Účtovanie

V peniazoch nie je nič špecifické pre protokol. Dôverné relácie bežia cez obyčajné účty uzla (rovnaká cesta rezervácie a finalizácie ako pri jeho EHBP toku s nepriehľadným telom). Prístupový údaj klienta, dobíjanie, zostatok aj refundácie (`/v1/wallet/refund`) sú tie, ktoré už existujú.

12.1 Rezervácia: prečo celé kontextové okno

Uzol prompt nevidí, takže ho nevie vopred nacenit'. Rezervuje preto najhorší prípad pri zmrazených sadzbách:

```
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)
```

Ak dĺžku kontextu modelu nepoznáme, použije sa namiesto toho maximálna nezľavnená cena uzla pre daný model (aj s prirážkou). Príklad z článku: `deepseek-v4-flash` s limitom promptu 104 858 tokenov a `max_tokens` 40 rezervuje 2 282 msat, pričom relácia v skutočnosti stála 63 msat. Model s kontextom 1M tokenov rezervuje pri sadzbách z prípadovej štúdie asi 157 satov. Pri vyúčtovaní sa rezervácia uvoľní až na skutočnú cenu, no na začatie relácie musí mať prístupový údaj taký veľký zostatok. Limit tokenov vynucuje π_C2S spolu so zadržaním a veľkosť promptu samotné kontextové okno, takže rezervácia vždy pokryje skutočnú cenu.

12.2 Vyúčtovanie podľa údaja o spotrebe od poskytovateľa

```
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)
```

Tokeny promptu z cache sa účtujú sadzbou pre cache, nikdy nie vyššou než vstupná sadzba. Ak sa model v údaji o spotrebe líši od modelu zo setupu, uzol strhne celú rezerváciu (sufix model zafixuje, takže ide o porušenie protokolu). Platba ide cez bežnú funkciu uzla `finalize_ehbp_actual_cost_payment`.

12.3 Poplatky: cena za prácu uzla

Relácia stojí uzol asi 22 MB uploadu a 0,5 s CPU ešte predtým, než poskytovateľ uvidí jediný bajt, a predhriata relácia toľko stojí ešte skôr, než vôbec existuje požiadavka. Uzol môže v podpísanej ponuke zverejniť tri voliteľné nastavenia (všetky sú štandardne 0):

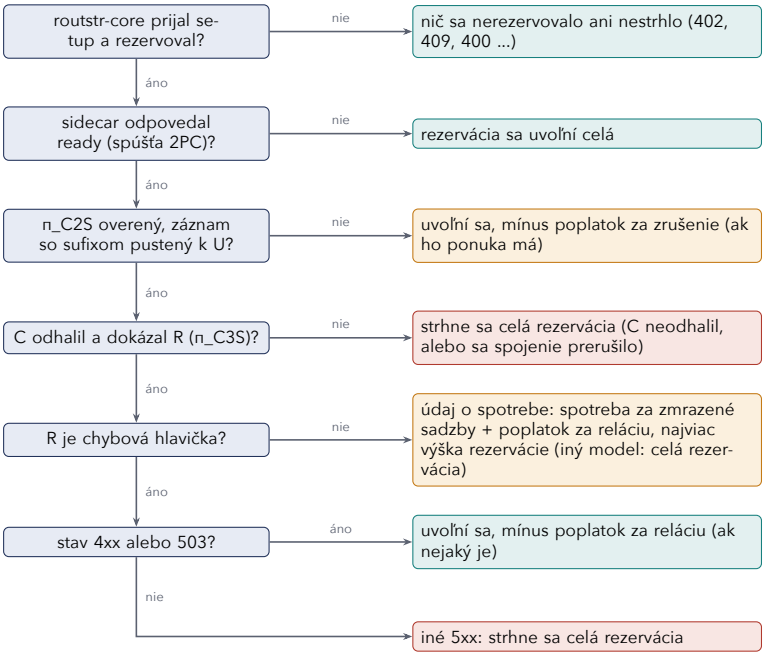
- **Prirážka k sadzbám** (`CONFIDENTIAL_RATE_SURCHARGE_PCT`) zvyšuje sadzby za token. Je už započítaná v podpísanom cenníku, takže kontrola sadzieb na strane klienta sa nemení.
- **Poplatok za reláciu** (`CONFIDENTIAL_SESSION_FEE_MSAT`) sa pripočíta k dokončenej požiadavke a strhne sa samostatne pri odmietnutí poskytovateľom (4xx, 503), keďže aj vtedy požiadavka k poskytovateľovi došla.
- **Poplatok za zrušenie** (`CONFIDENTIAL_ABANDON_FEE_MSAT`) sa strhne namiesto nuly, keď relácia, ktorú sidecar prijal, skončí bez toho, aby k U došla požiadavka. Nikdy nie je nižší než poplatok za reláciu.

Poplatok za zrušenie funguje ako záloha: rezervuje sa pri setupe a keď sa relácia použije, všetko nad poplatok za reláciu sa v praxi vráti. Platiaci používateľ ho teda nikdy nezaplatí, kým nabitá relácia, ktorá sa nikdy nepoužije, stojí presne toľko. Klient poplatky prijme tak, že ich zopakuje vo svojom setupe. Setup, ktorý ich nezopakuje, uzol odmietne, takže starému klientovi sa nikdy nestrhne poplatok, ktorý nekontroluje. Nasadený uzol z prípadovej štúdie si neúčtuje poplatok za reláciu ani prirážku a poplatok za zrušenie má 2 saty. Dokončený krátky chat stál 25 msat, zrušená relácia 2 000 msat.

12.4 Chyby poskytovateľa

HTTP chyba od U nenesie údaj o spotrebe. C odhalí len hlavičku odpovede (stavový riadok, hlavičky, prázdny riadok) a zvyšok odpovede preskočí cez `skip`. Telo chyby môže citovať požiadavku (Venice cituje názov neznámeho poľa), preto zostáva súkromné. Pri 4xx a 503 poskytovateľ nič neurobil, takže uzol rezerváciu uvoľní a ponechá si len poplatok za reláciu. Ostatné 5xx mohli uzol u poskytovateľa stáť peniaze a účtujú sa rovnako ako chýbajúce odhalenie.

12.5 Čo sa stane pri prerušení v jednotlivých bodoch



Obr. 12.1: Ako končí rezervácia relácie. Tyrkysová: zadarmo, jantárová: poplatok alebo spotreba, červená: celá rezervácia.

Kde sa to zastaví	Výsledok
setup odmietnutý (zlý prístupový údaj, chýbajú prostriedky, neznámy model, ponuka vypršala, poplatky neprijaté)	nič sa nerezervuje; explicitný chybový rámec so stavom (402, 409, 400 …)
sidecar odmietne setup (žiadne `ready`)	rezervácia sa uvoľní celá
zlyhá kontrola 2PC, ClientHello je odmietnutý, zlyhá handshake, zlyhá π_NK, klient odíde, nabitá relácia sa nikdy nepoužije, zlyhá π_C2S	U nikdy nedostal celú požiadavku: uvoľní sa, mínus prípadný poplatok za zrušenie
spotreba odhalená a dokázaná	spotreba pri zmrazených sadzbách plus poplatok za reláciu, najviac výška rezervácie

Kde sa to zastaví	Výsledok
odhalené 4xx alebo 503	uvoľní sa, mínus prípadný poplatok za reláciu
iné 5xx alebo žiadne odhalenie po uvoľnení tela	strhne sa celá rezervácia

routstr-core sa rozhoduje podľa dvoch príznakov: `engaged` (sidecar odpovedal `ready`) a `body_released` (sidecar oznámil, že sa chystá uvoľniť záznam so sufixom). Sidecar pošle `body_released` ešte pred zápisom a uzol po odchode klienta ešte až 10 s dočerpáva rámce, ktoré sú na ceste, takže rozhodnutie o peniazoch vidí skutočný konečný stav sidecaru. Vyúčtovanie beží chránené pred odpojením klienta a finalizuje sa práve raz.

> **Dôvera, ktorá zostáva**
>
> Ak uzol preruší stream pred údajom o spotrebe, strhne rezerváciu a klient nevie dokázať, kto spojenie prerušil. Takú dôveru už routstr uzlu dáva. Suma, o ktorej uzol rozhoduje, je rezervácia pre požadovaný model.

13

Potvrdenka a atestácia

Po vyúčtovaní routstr-core podpíše potvrdenku (BIP-340 Schnorr nad SHA-256 kanonického JSON-u) a pošle ju aj s presnými podpísanými bajtmi:

```
{
  "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
}
```

(Čísla sú vymyslené tak, aby zodpovedali príkladu z článku: rezervuje sa 2 282 msat za kontext plus 2 000 msat poplatku za zrušenie, strhne sa 63 msat.) Kontroly na strane klienta (`client/confidential/receipt.ts`, `session.ts`):

```
def verify_receipt(msg, my):
    r = parse(msg.receipt_json)              # len podpísané bajty
    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                # z môjho vlastného
↪   plaintextu
        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]     # PODPÍSANÉ sadzby
        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)
```

Kontrola ceny počíta každý token promptu za plnú vstupnú sadzbu. Uzol, ktorý tokeny z cache účtuje lacnejšie, ňou prejde, a uzol, ktorý si účtuje viac, neprejde. Už predtým klient overil podpis uzla na atestácii a to, že jej `sid`, `th_sh` a `th_sf` sú jeho vlastné: platne podpísaná atestácia inej relácie o tejto nič nehovorí. Potvrdenka je podpísané vyhlásenie uzla, ktoré sa klientovi hodí, no pre tretie strany nie je dôkazom (klient má `s_ap_0` a mohol by sfalšovať serverové záznamy voči komukoľvek okrem uzla).

14

Limity proti DoS

Relácia je pre uzol drahá ešte predtým, než poskytovateľ uvidí jediný bajt (22 MB uploadu, 0,5 s CPU, držaná rezervácia), a vďaka predhrievaniu sa nepoužitá relácia vytvára lacno. Limity, od bezplatných a vždy zapnutých po spoplatnené a voliteľné:

Limit	Kde	Hodnota
k U nič nedôjde pred požiadavkou	sidecar	TCP spojenie k U sa otvorí až po príchode ClientHello od C, po setupe, predspracovaní a key share
validácia a rezervácia pred prácou	core, potom sidecar	klient bez prostriedkov nedostane žiadny garbled circuit
súbežné relácie	core aj sidecar	po 32 (`MAX_SESSIONS`, `--max-sessions`)
čas do rámca setup	core aj sidecar	15 s
životnosť relácie	core	600 s od prijatia WebSocketu (sidecar má vlastné 600 s časovače)
predspracovanie 2PC plus key share	sidecar	180 s
jeden dôkaz	sidecar	30 s; v jednej relácii vždy len jeden dôkaz naraz
dĺžka tela	core aj sidecar	1 bajt až 8 MiB
`max_tokens`	core aj sidecar	1 až strop z ponuky (4096)
nepoužité relácie na adresu klienta	core	12 (`CONFIDENTIAL_MAX_UNUSED_PER_IP`), HTTP 429; relácia sa počíta, kým sa nezapíše záznam s API kľúčom
bajty klienta v bufferi pred preposlaním	sidecar	`len + len/256 + 4` TLS záznamov maximálnej veľkosti (ClientHello, CCS, Finished, celé telo)

Limit	Kde	Hodnota
prealokácia kanála pre dôkazy	obe strany	každá strana navrhne veľkosť a vyhrá menšia (najviac 2 na 25.), takže C nemôže prinútiť N prealokovať viac, než si N sám vypýtal
veľkosť WebSocket správy	core	64 MiB

Adresa klienta je TCP protistrana, alebo, len pri protistranách nastavených ako dôveryhodné proxy, posledná položka vpravo v `X-Forwarded-For` (tá, ktorú pridala proxy). Klient si teda vlastnú adresu určiť nemôže. Všetci používatelia Toru jednej onion služby zdieľajú adresu, a teda aj limit. Nad tým všetkým naceňuje každú nepoužitú reláciu poplatok za zrušenie (kapitola 12). Z mnohých adries sa relácie otvárať stále dá, každá však zaplatí poplatok, a to už je spoplatnená služba, nie útok.

15

Náklady a časy

Všetky údaje o komponentoch sú z Apple M2 Max s release buildmi; „lokálne“ znamená bez sieťového oneskorenia.

2PC, studený štart, kým obe strany neskončia (medián z 3 meraní; „up/down“ je linka klienta):

Linka	Dual execution (staré)	Len uzol (aktuálne)
lokálne	0,17 s	0,63 s
40 ms jedným smerom, 2 / 50 Mbit/s	85,2 s	7,38 s
40 ms jedným smerom, 7 / 50 Mbit/s	24,9 s	5,41 s
25 ms jedným smerom, 100 Mbit/s symetricky	2,27 s	3,05 s
40 ms jedným smerom, 2 / 10 Mbit/s	85,2 s	20,0 s
klient pošle / prijme	21,1 / 21,1 MB	1,21 / 22,1 MB

Dôkazy:

Dôkaz	Lokálne	50 ms RTT, setup pre každý dôkaz	Predhriaty kanál, domáca linka	Dáta
π_NK	0,16 s	1,23 s	0,21 až 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

Celý ret’azec proti živému poskytovateľovi (`deepseek-v4-flash`, 40 tokenov):

- jeden stroj, studený štart: od začiatku požiadavky po odoslanie tela asi 1,2 s; dôkaz zadržaného sufixu 0,10 s;
- jeden stroj, predhriaty: od prevzatia relácie po odoslanie tela 5 ms;

- dobrá domáca linka (asi 100 Mbit/s dole, 60 až 80 ms k uzlu), studený štart: predspracovanie 5,1 až 11,7 s, prvý token po 13,8 až 16,2 s; plne nabitá relácia: 73 ms od prevzatia po odoslanie tela a prvý token po 1,05 s, pričom väčšinu tvorí latencia samotného poskytovateľa;
- slabá linka s obmedzeným downloadom: studená požiadavka čaká asi minútu.

Na jednu požiadavku pripadá asi 23,3 MB prevádzky 2PC a 4,4 MB dôkazov, spolu asi 28 MB. Uzol nahrá asi 24 MB (100 požiadaviek za minútu je asi 0,3 Gbit/s) a na 2PC minie asi 0,5 s CPU. Nič z toho nezávisí od veľkosti promptu ani odpovede: po handshake ide payload natívnou rýchlosťou TLS.

16

Predhrievanie

Garbled tabuľky sa dajú použiť iba raz, takže protokol ich cenu nezdieľa, ale skrýva. Čo má nabitá relácia hotové a čo ju ešte čaká:

Predpočítané počas nabitia	Spraví sa, keď si reláciu vezme požiadavka
setup, kontrola prístupového údaja, rezervácia	ClientHello, ECtF, fázy A a B
celé predspracovanie 2PC (22 MB)	overenie handshaku na oboch stranách
spoločný key share	π_NK, záznam s API kľúčom a KeyUpdate
korelácie kanála pre dôkazy pre π_NK, π_C2S, π_C3S	telo, π_C2S, odpoveď, π_C3S, vyúčtovanie

Nabitá relácia sa zastaví **pred ClientHello**. Poskytovateľ totiž spojenie, ktoré v handshaku nič nerobí, do pár sekúnd zavrie, a nabitá relácia preto k poskytovateľovi nedrží žiadne spojenie. (Režim `--arm` referenčného klienta ide pre benchmarky d'alej, cez handshake až po π_NK.)

Pool v SDK (`client/confidential/prewarm.ts`, predvolené hodnoty tak, ako sa dodáva):

- **Kľúč** poolu: uzol, model, `max_tokens` (nesie ho sufix) a prístupový údaj.
- **Len pri prevádzke.** Pri štarte sa nenabíja nič a rovnako ani pre kľúč, na ktorý za posledných 10 minút neprišla požiadavka; nečinný pool nemá spustený žiadny časovač.
- **Koľko.** Jedna relácia na aktívny kľúč, po nápore (3 požiadavky za 5 minút) dve, alebo najvyšší počet súbežne bežiacich požiadaviek; najviac 6 na kľúč a 6 celkovo, čo je pod limitom uzla 12 na adresu. Naraz sa nabíja len jedna (zaberá veľa prenosového pásma) a nikdy nie vtedy, keď studená požiadavka ešte beží vo vlastnom predspracovaní.
- **Dĺžka tela.** `len` sa určí pri setupe, preto pool vyberá bucket s veľkosťou mocniny dvoch: najväčšie z posledných 20 tiel krát 1,25, od 4 KiB po 8

MiB; kratšie telá sa doplnia medzerami. Väčšia požiadavka pôjde studenou cestou a bucket zväčší.

- **Vyradenie** 480 s po setupe (uzol končí relácie po 600 s); keď kľúč prestane byť aktívny, jeho relácie sa hneď zatvoria.
- **Peniaze.** Nabíja sa až potom, čo sa pre daný kľúč vyúčtovala nejaká požiadavka (z potvrdenky je známa veľkosť rezervácie), a len kým je voľný zostatok aspoň dvojnásobný. Ďalej len vtedy, keď je poplatok za zrušenie najviac 5 000 msat, a pri uzloch, ktoré ho účtujú, najviac 6 relácií celkovo. Poplatky zaplatené za nevyužité relácie idú do logu.
- **Zlyhania.** Po neúspešnom nabití kľúč čaká (30 s, zdvojnásobuje sa až po 10 min). Ak si reláciu vezme požiadavka a tá zlyhá pred `release_head`, prejde sa na studenú reláciu; po `release_head` sa už nikdy neopakuje (požiadavka sa mohla dostať k poskytovateľovi).

17

Obrázky

Tá istá relácia môže namiesto chatu niesť požiadavku na obrázok. V epochách TLS, v 2PC ani v troch tvrdeniach sa nič nemení. Mení sa HTTP požiadavka, cena a spôsob, akým sa relácia vyúčtuje. Návrh a záväzné rozhodnutia sú v `docs/CONFIDENTIAL-IMAGES.md` (sekcia 0).

Ponuka. Uzol, ktorý ponúka obrázky, pridá do podpísanej ponuky `capabilities: ["chat", "images"]` a blok `images`:

```
"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}]}
}
```

Cenník zostaví uzol zo svojich cien za obrázok. Model ponúkne, len ak jeho cena závisí iba od pripnutých parametrov; modely účtované podľa tokenov a poskytovatelia, ktorí hlásia vlastnú cenu, v ponuke nie sú. Blok pochádza z `endpoints.images` v súbore ponuky sidecaru: bez neho uzol obrázky neponúka.

Sufix. Pre model M, počet k a variant V je sufix bajt po bajte `,"model":M,"variants":k,`, potom pevné piny, potom piny variantu V a nakoniec `}`, v kompaktnom JSON. Uzol ho vyrenderuje a pošle sidecaru, ktorý ho skontroluje; klient si vyrenderuje vlastný z podpísanej ponuky a vyžaduje, aby sa `ready.suffix` zhodoval. π_C2S ho dokazuje presne tak ako sufix chatu. Prompt a každá voľba, ktorá nemení cenu (`aspect_ratio`, `seed`, `negative_prompt`, bezpečnostný režim), zostávajú v súkromnej časti.

Setup. Setup klienta uvádza `kind: "images"`, `model`, id variantu (`variant`) a `count`, bez `max_tokens`, a nikdy nie je `armed`. Uzol zarezervuje `msat_per_image × count + max(f_s, f_a)`.

Vyúčtovanie: `accept`**.** Cena je známa už pri setupe, takže netreba dokazovať žiadnu spotrebu. Po atestácii pošle klient pri 2xx `{"type": "accept"}`; sidecar ohlási paušálnu spotrebu a uzol zaúčtuje `msat_per_image × count + f_s`. Pri chybovom stave klient odhalí hlavičku odpovede ako pri chate (4xx/503: uvoľnené mínus `f_s`). Klientovi, ktorý nepošle ani jedno, sa zaúčtuje rezervácia, ktorá nikdy nie je nižšia ako paušálna cena. Potvrdenka navyše nesie `kind`, `variant`, `count` a `msat_per_image` a klient kontroluje `cost_msats == ceil(msat_per_image × count + f_s)`.

Čo sa uzol dozvie. Model, variant a počet (tak ako pri chate model a limit tokenov), dĺžku tela, veľkosť a časovanie odpovede a úspech alebo chybový stav. Nedozvie sa prompt, súkromné voľby, obrázok ani hlavičky úspešnej odpovede (z ktorých by vyčítal, či bol obrázok rozmazaný).

API. V prehliadači `client.images.generate({model, prompt, count, resolution, aspectRatio})` vráti obrázky ako `Blob`y spolu s overenou reláciou; uzol bez tejto schopnosti hodí `offer_no_images`, chyba poskytovateľa `UpstreamImageError`. V SDK a routstrd spustí `POST /v1/images/generations` (tvar OpenAI) a `POST /v1/image/generate` (tvar Venice) s dôverným opt-inom obrázkovú reláciu a odpovedia `{created, data: [{b64_json}]}`.

Bez predhrievania. Obrázkové relácie sa nikdy nenabíjajú a nesiahajú do chatového poolu: každá zaplatí svoje studené 2PC, asi 26 MB protokolovej prevádzky pred samotným obrázkom. Na živom uzle trvala požiadavka na jeden obrázok z referenčného klienta 10 s; v prehliadači trvala studená požiadavka 16,7 s, z toho 7,2 s generoval obrázok poskytovateľ.

18

Čo neskrýva a ďalšie obmedzenia

- **Poskytovateľ vidí všetko.** Prompt, odpoveď aj API kľúč. Protokol odstraňuje dôveru v uzol, nie v poskytovateľa.
- **Analýza prevádzky.** N vidí veľkosť každého záznamu a kedy prišiel. Dĺžka tela je viditeľná a poskytovatelia pri streamovaní posielajú zhruba jeden záznam na token alebo skupinu tokenov, takže uniká postupnosť dĺžok tokenov v odpovedi. V publikovaných útokoch sa cez takéto bočné kanály podarilo zrekonštruovať veľké časti odpovedí LLM. Prompty sa dajú doplniť (padding) vnútri JSON tela, na strane odpovede by padding musel robiť poskytovateľ.
- **Metadáta chýb.** Pri chybe poskytovateľa sa N dozvie stavový riadok, hlavičky a dĺžku tela, telo nie.
- **Epochu 0 má pod kontrolou uzol** (sekcia 11.4).
- **2PC je nová varianta** bez publikovaného dôkazu bezpečnosti a bez externého review (kapitola 8) a o čerstvých hodnotách prezradí až 2,3 bitu na reláciu.
- **P-256.** Spoločný key share potrebuje sčítanie bodov, takže X25519 sa použiť nedá.
- **Žiadny dôkaz pre tretiu stranu.** Všetko presvedčí iba dve strany danej relácie.
- **Zmeny u poskytovateľa.** Zmena na strane poskytovateľa môže rozbiť fixovanie alebo hlásenie spotreby; profilovú kontrolu treba spustiť znova a pokazené relácie zlyhajú, neprejdú na náhradný režim.
- **Koncové body.** Kľúče žijú v procese klienta a API kľúč v pamäti uzla. Klient pri overovaní certifikátov nekontroluje revokáciu.

19

Predchodcovia: v1 a dual execution

v1, jednostranná varianta (vetva `v1-poc`). Bez 2PC: C robí TLS handshake sám s X25519 share a odhalí iba `c_hs` a `s_hs`. V epoche 0 pošle KeyUpdate a dá N **jednorazový materiál pre záznam** pre sekvenciu 0 epochy 1 (keystream skrátený na dĺžku hlavičky, GHASH kľúč a masku tagu). Dôkaz π_C1 ukáže N, že materiál pochádza z odvodzovania kľúčov tejto relácie; N si záznam vyrobí cez XOR a jeho obsah dokáže cez π_N. C potom kľúč znova otočí a telo píše v epoche 2. π_C2 a π_C3 sú ako π_C2S a π_C3S, len odvodzovanie kľúčov počítajú znova od HS. Slabina v1 je v tom, že C drží celý kľúč epochy 1, takže API kľúč sa rovná ciphertextu záznamu XOR keystream, ktorý C pozná. P1 platí len dovtedy, kým C ten ciphertext nikdy neuvidí, a to je prevádzkový predpoklad (ktokoľvek na odchádzajúcej ceste uzla, kto by spolupracoval s C, ho poruší). v1 potrebuje na požiadavku asi 5,9 MB namiesto asi 28 MB a hodí sa pre uzly, ktoré majú cestu k poskytovateľovi pod kontrolou.

Dual execution (tag `v2-symmetric`) bola prvá 2PC vo v2: každá fáza sa garblovala dvakrát, raz každou stranou, a výstupy sa porovnali commit-then-open hashom výstupných labelov. Stojí na jednoduchších a lepšie preskúmaných konštrukciách a prezradí menej (asi 1,6 bitu na reláciu), ale každá strana garblovala 656 179 AND hradiel a posielala asi 21 MB: za studena 85 s pri 2 Mbit/s uplinku oproti dnešným 7,4 s.

A

Slovník: názvy a symboly z článku

Článok	Táto kniha	Význam
C, N, U	klient, uzol, poskytovateľ	klientsky démon, smerovací uzol (core a sidecar), API poskytovateľa
P1 až P6	ciele	API kľúč, obsah, integrita, účtovanie, autenticita, čerstvosť
Z	ECDHE tajomstvo	x-ová súradnica spoločného bodu; rozdelená na aditívne podiely
a_C, a_N; P_C, P_N; P; S	skaláry; body; spoločný share; share servera	príspevky do ECDHE; P je jediný key share, ktorý U vidí
HS, MS	handshake secret, master secret	vnútorné uzly odvodzovania kľúčov; nikto ich nemá
c_hs, s_hs	handshake traffic secrets	dostanú ich obe strany, aby overili handshake
c_ap,0; K_0, IV_0	kľúč epochy uzla	secret a kľúč/IV záznamov epochy 0
c_ap,1; K_1, IV_1	klientov kľúč pre telo	secret a kľúč/IV záznamov epochy 1
s_ap,0	kľúč odpovede	aplikačný secret servera
th_SH, th_SF	hashe transkriptu	po ServerHello; po Finished servera
h_1, h_2, h_k	kotvy h1, h2, hk	verejné hashe c_ap,1, s_ap,0, K_0 a IV_0
r_C, r_N; m_C, m_N	masky	masky MS (fáza A); výstupné masky (fáza B)
ECtF, A2M, M2A, OLE	konverzia podielov	z dvoch bodov spravia aditívne podiely x-ovej súradnice
Delta_A	garblovací offset uzla	rozdiel medzi dvoma labelmi každého drôtu
Delta_B	MAC kľúč klienta	globálny kľúč IT-MAC tagov; 127 náhodných bitov
lambda_w	maska drôtu	tajný bit N; posledný bit 0-labelu N
Lambda_w	pointer bit	skutočná hodnota XOR maska; to, čo vidí C

Článok	Táto kniha	Význam
t_k	súčin masiek	súčin dvoch vstupných masiek AND hradla, s tagom
chi	výzva	z C-čkového seedu, až keď N spravil commitment
e_k, E	chyba hradla, kombinovaná chyba	nulová pre každé konzistentne vyhodnotené hradlo
Y_N, check rows, pi(.)	kontrolný materiál, 64-bitová projekcia	čo N posiela na kontrolu AND hradiel; skrátenie na 64 bitov
KOS, Ferret, COT, RCOT	korelovaný OT	vopred pripravené náhodné korelácie
QuickSilver, VOLE-ZK	dôkazový systém	interaktívne ZK s určeným overovateľom z mpz2
π_NK, π_C2S, π_C3S	tri dôkazy	šablóna záznamu; sufix; odhalenie spotreby
π_C1, π_N, π_C2, π_C3	dôkazy vo v1	materiál záznamu; šablóna; sufix; spotreba
R, skip	odhalenie	posledný plaintext odpovede; neodhalené bajty za ním
L, T, m	limit promptu, strop tokenov, minimum	vstupy rezervácie
r_in, r_out, r_cached	sadzby	msat za 1 000 tokenov zo zmrazenej ponuky
f_s, f_a	poplatok za reláciu, poplatok za zrušenie	voliteľné poplatky v podpísanej ponuke
sid	id relácie	SHA-256 oboch nonce a id rezervácie
DEAP, Garble-then-Prove	návrhy s neskorším odkrytím	2PC-TLS návrhy, ktoré podiely potom odhalia; tu nepoužiteľné
WRK17, KRRW18, DILO22, CWYY23	autentizované garblovanie	literatúra, z ktorej vychádza kontrola len na strane uzla

B

Kde čo nájdeš v kóde

Oblasť	Súbor
stavový automat sidecaru, kontrola setupu, buffre, zadržanie záznamu, parsovanie R	`crypto/relay/src/main.rs` (`handle_setup`, `handle_control`, `relay_body`, `v2_pump_client`, `v2_progress`, `validate_r`, `validate_error_head`)
pravidlá pre ClientHello a ServerHello	`crypto/relay/src/tls_v2.rs` (`check_client_hello`, `server_hello_share`)
overenie handshaku	`crypto/relay/src/tls.rs` (`verify_handshake`)
2PC	`crypto/twopc/src/lib.rs` (predspracovanie, fázy), `gc.rs` (three-halves garblovanie, check rows), `circuits.rs`, `ectf.rs`; binárka `bin/cu-2pc.rs`
dôkazové tvrdenia	`crypto/zkbench/src/statements_v2.rs` (`build_pi_nk`, `build_pi_c2s`, `build_pi_c3s`), `gadgets.rs`, `channel.rs` (kanál pre dôkazy)
ponuka, rezervácia, vyúčtovanie, poplatky	`routstr-core/routstr/confidential.py`
WebSocket relay uzla, limit na adresu, spracovanie prerušení, potvrdenka	`routstr-core/routstr/confidential_ws.py`
relácia klienta a kontroly	`routstr-sdk/client/confidential/session.ts`, `params.ts`, `receipt.ts`, `offer.ts`, `suffix.ts`, `transport.ts`
predhrievanie	`routstr-sdk/client/confidential/prewarm.ts`, `zkchan.ts`, `twopc.ts`
kontroly poskytovateľa a adversariálne testy	`crypto/tools/provider-profile-check.py`, `adversarial-model-swap.sh`, `regression-modality.py`