Oblivious HTTP

When the server does not want to see your IP

Thibault Meunier
Cloudflare Research

CLOUDFLARE

Some sites don’t want to see your IP address

Services that don’t want to see your IP address

1. Telemetry services
2. Health tracking app
3. AI inference services
4. DNS resolution

Simple solution

Setup an intermediary that strips the IP address and proxies the request

Simple solution

Request

IP

Headers

Fingerprint

Request

~~IP~~

~~Headers~~

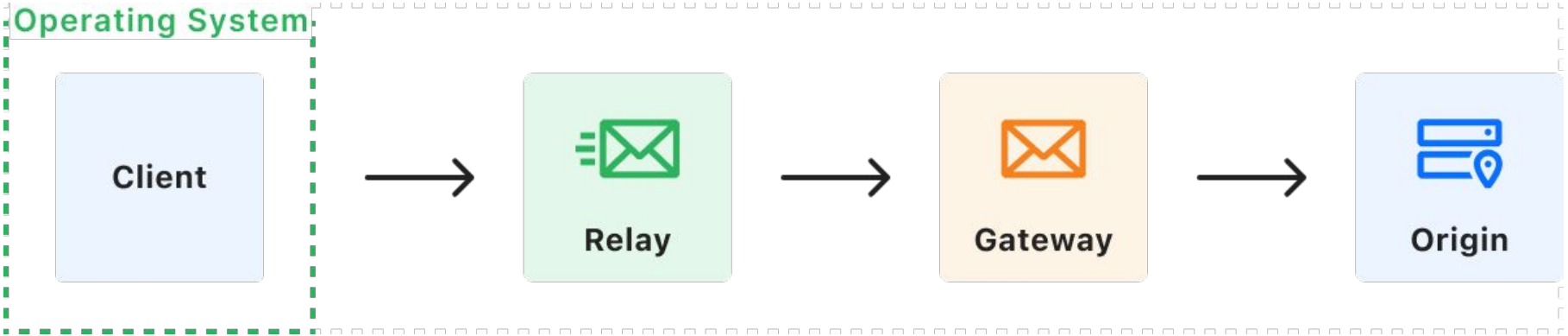
~~Fingerprint~~

CLOUDFLARE

Oblivious HTTP

Seeing less information is more work

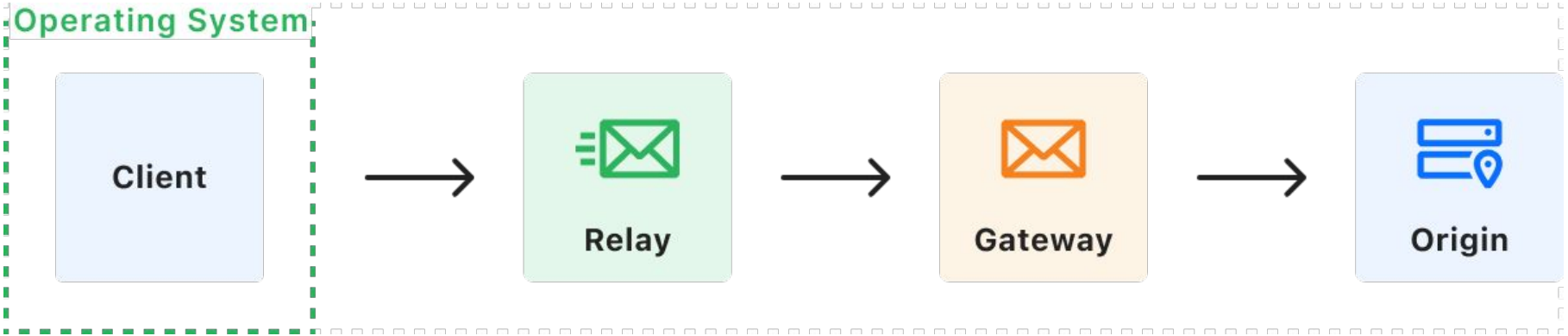
Oblivious HTTP - RFC 9458



1. The Client encrypts the request to the Gateway public key
2. The Client sends the encrypted request to the Relay
3. The Relay forwards it to the Gateway, stripping identifying information
4. The Gateway decapsulates the request

Same for the response

Signals seen by the Origin

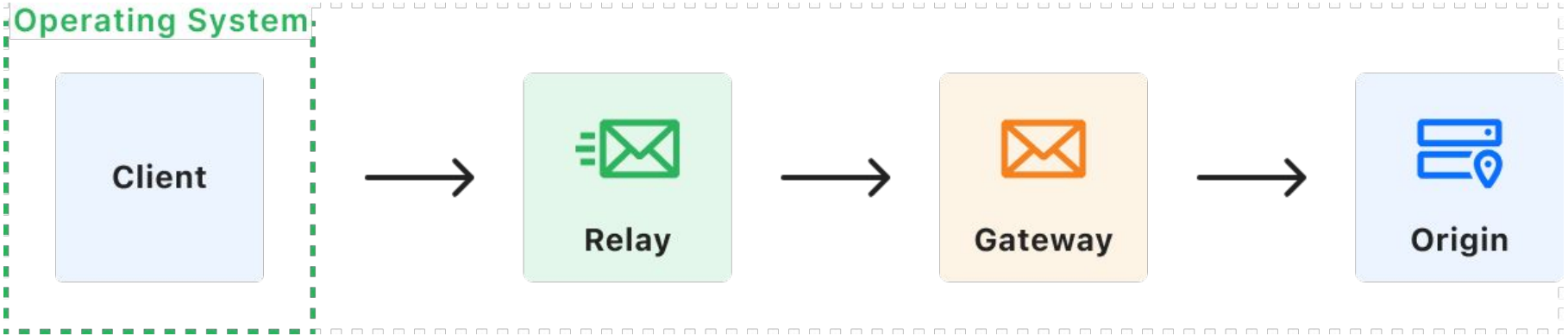


~~Request~~
IP
Headers
Fingerprint

Request
~~IP~~
~~Headers~~
~~Fingerprint~~

Request
~~IP~~
~~Headers~~
~~Fingerprint~~

Incentives



Origin wants

- Security
- Resource management
- Legal/T&S
- Responsiveness

OHTTP provides

- Requests unlinkability
- IP stripping
- Headers stripping
- Time decorrelation

In TypeScript with thibmeu/ohttp-ts

```ts
import { OHTTPClient } from "ohttp-ts";

// Setup -- we have keyConfig
const client = new ChunkedOHTTPClient(suite, keyConfig);

// Client: encapsulate request
const request = new Request(
  "https://target.example/upload", {
  method: "POST",
  body: "Hello World"
});
const { init, context } =
  await client.encapsulateRequest(request);

// Send to relay
const relayResponse = await fetch(
  "https://relay.example/ohttp",
  init
);
```

Client

```ts
import { OHTTPServer } from "ohttp-ts";

// Setup -- we have keyConfig
const server = new OHTTPServer([keyConfig]);

function handleHTTP(receivedRequest: Request): Response
{
  // Decapsulate the request
  const { request, context } =
        await gateway.decapsulateRequest(relayRequest);

  // Encapsulate the response
  const httpResponse = new Response(
    JSON.stringify({ result: "ok" }),
    { status: 200 },
  );
  const encapsulatedResponse =
        await context.encapsulateResponse(httpResponse);

  return encapsulatedResponse;
}
```

Server

What OHTTP does not solve

1. Need per user auth tied to IP/session
2. Latency sensitive flow
3. You cannot guarantee separation between relay/gateway
4. You need anonymity against a global observer

Demo

Take your phone

Go to https://ohttp.info

CLOUDFLARE

Thank you*

What we need to define

At minimum

1. Serialisation of the HTTP request
2. Encryption of that message

Nice to have

3. Abuse resistance
4. Streaming

CLOUDFLARE

Agenda

1. Oblivious HTTP
2. Encoding with Binary HTTP
3. Encrypting with HPKE
4. Abuse resistance with Privacy Pass
5. Streaming capabilities with Chunked OHTTP
6. Implementations

CLOUDFLARE

Binary HTTP

The best wrap we'll eat today

HTTP is a structured format

```
echo "GET / HTTP/1.1
  Host: example.com
  Connection: close
  " | nc example.com 80
```

You can even set encoding

```
echo "GET / HTTP/1.0
  Host: example.com
  Accept-encoding: gzip
  Connection: close
  " | nc example.com 80 \
  | awk 'BEGIN {RS="\r\n\r\n"; ORS=""} NR==2 {print}' \
  | gunzip
```

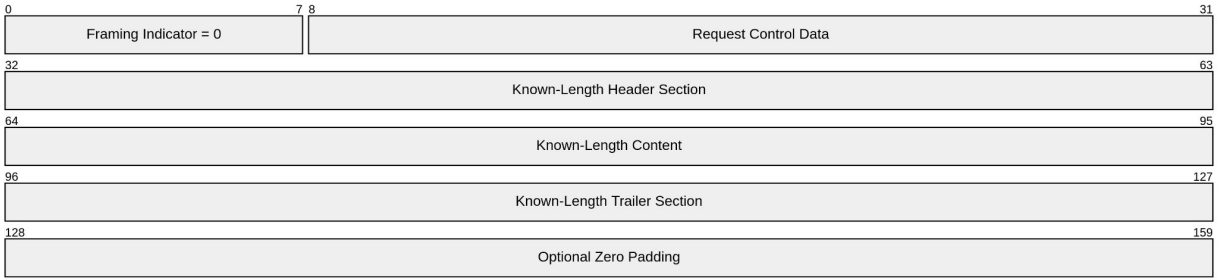
Binary HTTP - RFC 9292

Format to wrap HTTP

Defined in RFC 9292

Two modes:

1. known length
2. unknown length



RFC 9292 message/bhttp - Known-Length Request

In TypeScript with thibmeu/bhttp-ts

```ts
import { BHttpDecoder, BHttpEncoder } from "bhttp-ts"

const encoder = new BHttpEncoder()
const decoder = new BHttpDecoder()

const request = new Request(
  'https://example.com',
  {
    headers: { 'accept': 'text/html' }
  },
)
const encRequest = encoder.encodeRequest(request)

const encResponse = await fetch(
  'https://relay.example.com',
  {
    method: 'POST',
    headers : { 'content-type': 'message/bhttp' },
    body: encRequest,
  }
)
const response = decoder.decodeResponse(
  await encResponse.arrayBuffer()
)
```

Client

```ts
import { BHttpDecoder, BHttpEncoder } from "bhttp-ts"

const encoder = new BHttpEncoder()
const decoder = new BHttpDecoder()

function handleHTTP(receivedRequest: Request):
Response {
  const encRequest = await
receivedRequest.arrayBuffer()
  const request = decoder.decodeRequest(encRequest)

  const response = new Response('Hello World')
  const encResponse = encoder.encodeResponse(response)

  return new Response(
    encResponse,
    { headers: { 'content-type': 'bhttp/message' } }
  )
}
```

Server

CLOUDFLARE

Hybrid Public Key Encryption

Start cooking with this cryptography stand mixer

Hybrid Public Key Encryption - RFC 9180

Known as HPKE

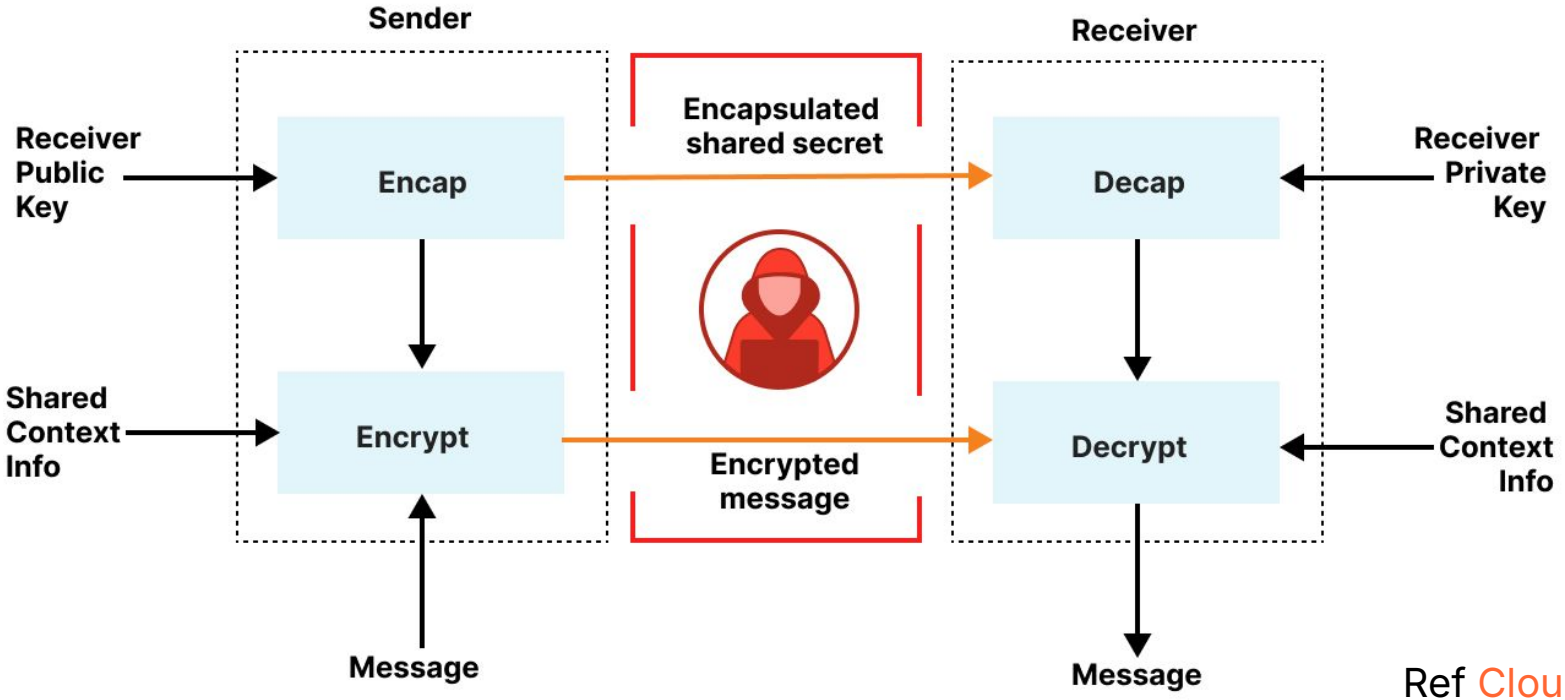
⇒ *Uses a KEM (asymetric) to derive a shared secret, then AEAD (symetric) to encrypt arbitrary-length data*

Provides

1. Agility on the algorithms
2. Different modes: authenticated and not

Allows for PQ-resistance for instance

HPKE



Ref Cloudflare blog

How does HPKE interface look like

```
def SealAuthPSK(pkR, info, aad, pt, psk, psk_id, skS):
  enc, ctx = SetupAuthPSKS(pkR, info, psk, psk_id, skS)
  ct = ctx.Seal(aad, pt)
  return enc, ct

def OpenAuthPSK(enc, skR, info, aad, ct, psk, psk_id, pkS):
  ctx = SetupAuthPSKR(enc, skR, info, psk, psk_id, pkS)
  return ctx.Open(aad, ct)
```

Example - KEM Identifier registry

7.1. Key Encapsulation Mechanisms (KEMs)

Value	KEM	Nsecret	Nenc	Npk	Nsk	Auth	Reference
0x0000	Reserved	N/A	N/A	N/A	N/A	yes	RFC 9180
0x0010	DHKEM(P-256, HKDF-SHA256)	32	65	65	32	yes	[NISTCurves], [RFC5869]
0x0011	DHKEM(P-384, HKDF-SHA384)	48	97	97	48	yes	[NISTCurves], [RFC5869]
0x0012	DHKEM(P-521, HKDF-SHA512)	64	133	133	66	yes	[NISTCurves], [RFC5869]
0x0020	DHKEM(X25519, HKDF-SHA256)	32	32	32	32	yes	[RFC5869], [RFC7748]
0x0021	DHKEM(X448, HKDF-SHA512)	64	56	56	56	yes	[RFC5869], [RFC7748]

HPKE IANA registries

In TypeScript with panva/hpke

```ts
import * as HPKE from 'hpke'

// 1. Choose a cipher suite
const suite = new HPKE.CipherSuite(
  HPKE.KEM_DHKEM_P256_HKDF_SHA256,
  HPKE.KDF_HKDF_SHA256,
  HPKE.AEAD_AES_128_GCM,
)

// 2. Generate recipient key pair
const recipient = await suite.GenerateKeyPair()

// 3. Encrypt a message
const plaintext = new TextEncoder().encode('Hello, World!')
const { encapsulatedSecret, ciphertext } = await suite.Seal(recipient.publicKey, plaintext)

// 4. Decrypt the message
const decrypted = await suite.Open(recipient.privateKey, encapsulatedSecret, ciphertext)
console.log(new TextDecoder().decode(decrypted)) // "Hello, World!"
```

How is OHTTP using HPKE

Steps

1. The gateway exposes a configuration
2. BHTTP body is encrypted towards that configuration

This protects against tampering in transit

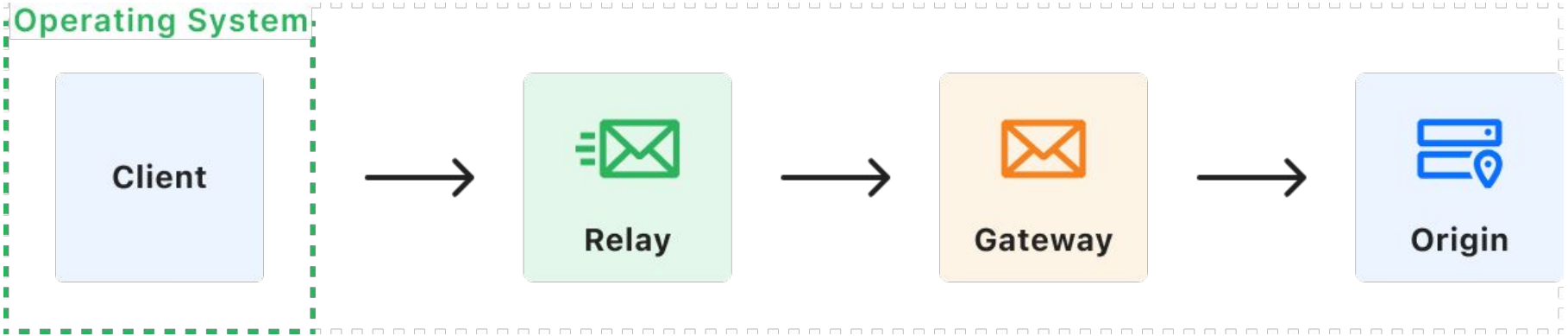
Things this does not solve: discovery. DO NOT trust the relay to provide the gateway configuration

CLOUDFLARE

Privacy Pass

Authentication, but anonymously

OHTTP setup



Origin wants

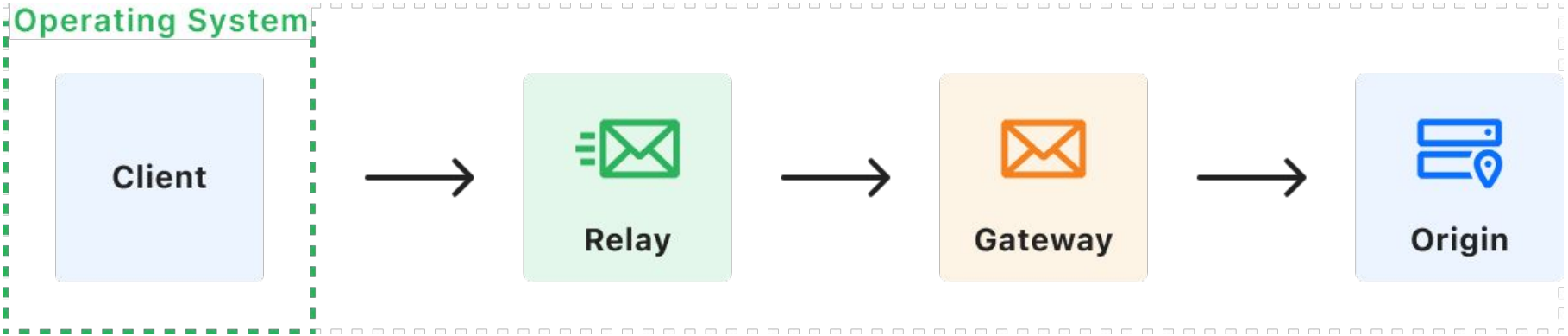
- Security
- Resource management
- Legal/T&S
- Responsiveness

OHTTP provides

- Requests unlinkability
- IP stripping
- Headers stripping
- Time decorrelation

What if the client is malicious?

OHTTP setup



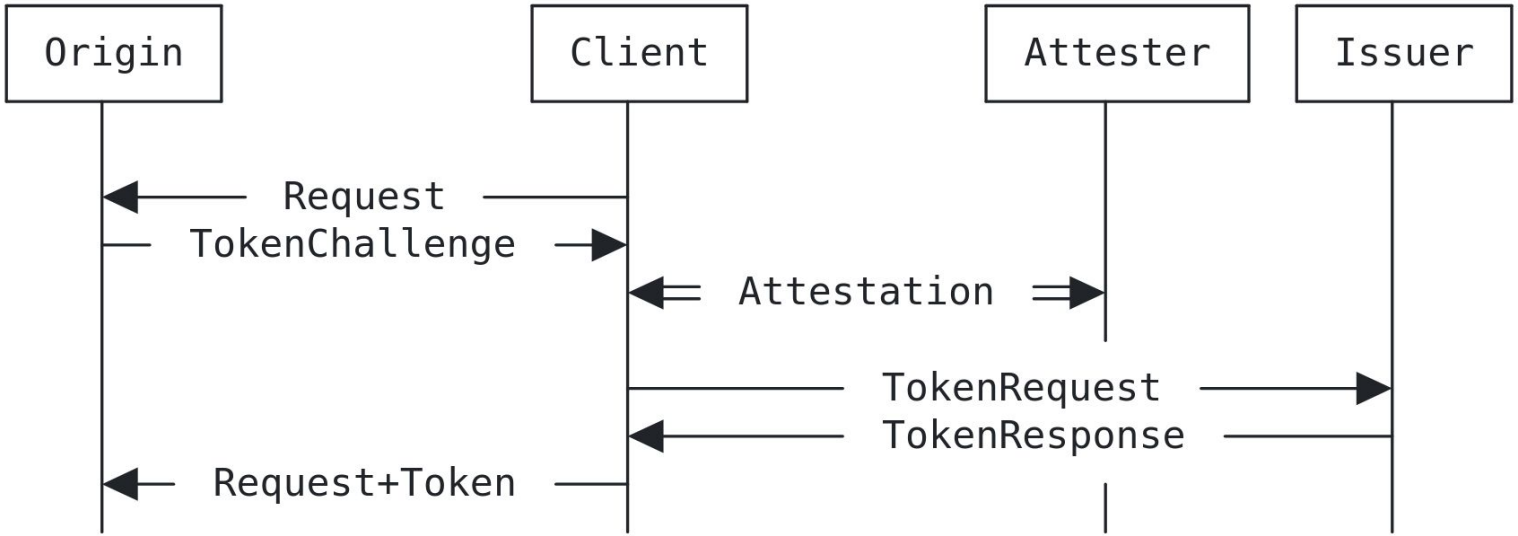
Origin wants

- ~~Security~~
- ~~Resource management~~
- ~~Legal/T&S~~
- ~~Responsiveness~~

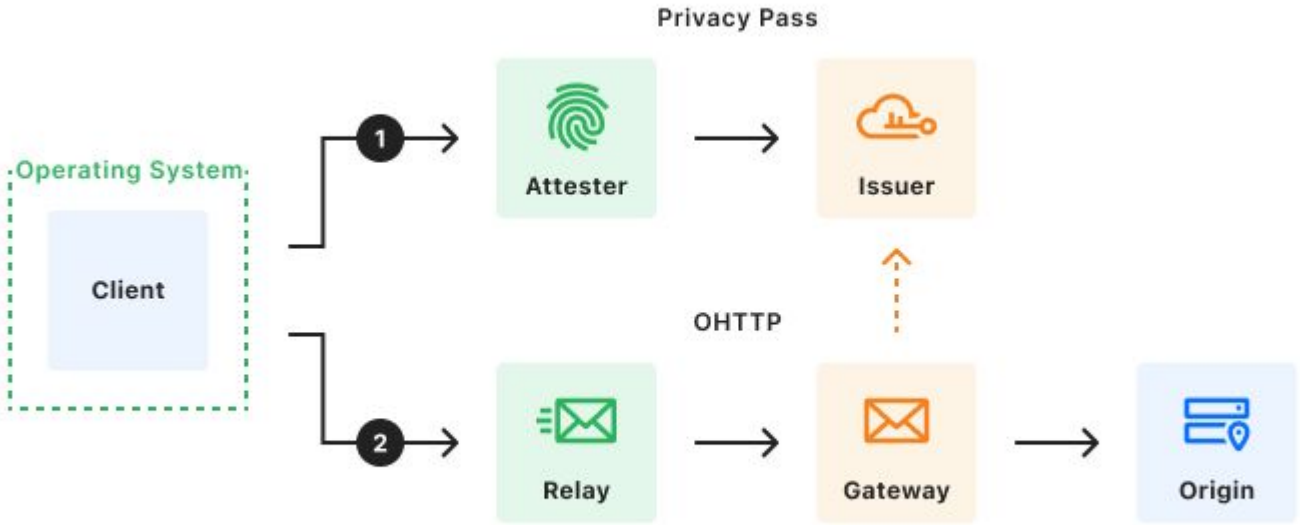
OHTTP provides

- Requests unlinkability
- IP stripping
- Headers stripping
- Time decorrelation

Privacy Pass - RFC 9576



Abuse


Privacy Pass
Operating System
Client
1
2
Attester
Issuer
OHTTP
Relay
Gateway
Origin

In TypeScript with cloudflare/privacypass-ts

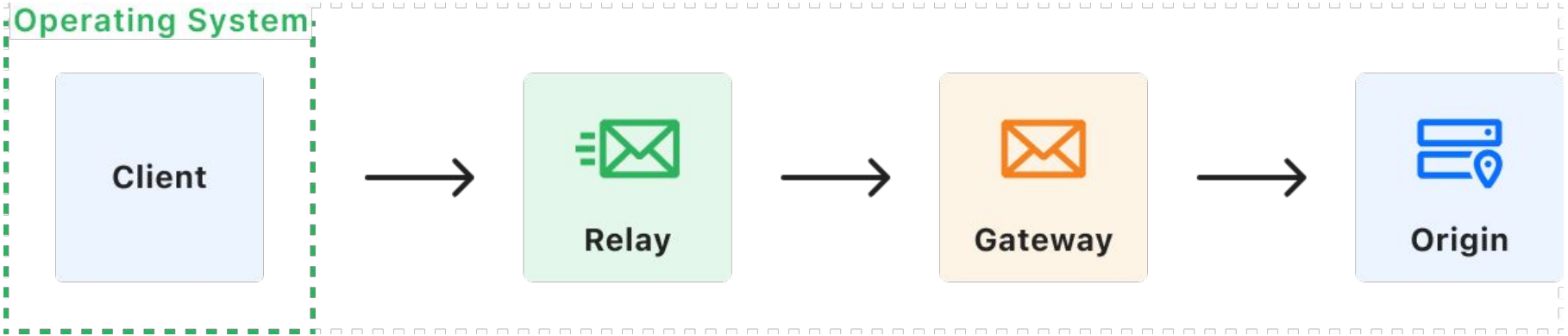
```ts
import { publicVerif } from "@cloudflare/privacypass-ts"

// Protocol Setup
const { issuer, client, origin, pkIssuer } = await setup(mode);

const redemptionContext = crypto.getRandomValues(new Uint8Array(32));
const tokChl = origin.createTokenChallenge(issuer.name, redemptionContext);

// Attestation + issuance
const tokReq = await client.createTokenRequest(tokChl, pkIssuer);
const tokRes = await issuer.issue(tokReq);
const token = await client.finalize(tokRes);

// Presentation + validation
const isValid = await origin.verify(token, issuer.publicKey);
```

CLOUDFLARE

Chunked OHTTP

Everything is better with chunks

Back to OHTTP



What if the Client does not know the full content ahead of time?

What if the server generates responses on the fly?

CLOUDFLARE

What if we had this new application type that needs to stream small tokens that take a while to generate and that might be sensitive?

Chunked OHTTP - draft-ietf-ohai-chunked-ohttp-08

Preserve OHTTP key establishment

Split OHTTP message into chunks

Consider out-of-order delivery

Revise the privacy and security considerations

In TypeScript with thibmeu/ohttp-ts

```ts
import { OHTTPClient } from "ohttp-ts";

// Setup -- we have keyConfig
const client = new ChunkedOHTTPClient(suite, keyConfig);

// Client: encapsulate request
const request = new Request(
  "https://target.example/upload", {
  method: "POST",
  body: "Hello World"
});
const { init, context } =
  await client.encapsulateRequest(request);

// Send to relay
const relayResponse = await fetch(
  "https://relay.example/ohttp",
  init
);
```

Client

```ts
import { OHTTPServer } from "ohttp-ts";

// Setup -- we have keyConfig
const server = new OHTTPServer([keyConfig]);

function handleHTTP(receivedRequest: Request): Response
{
  // Decapsulate the request
  const { request, context } =
        await gateway.decapsulateRequest(relayRequest);

  // Encapsulate the response
  const httpResponse = new Response(
    JSON.stringify({ result: "ok" }),
    { status: 200 },
  );
  const encapsulatedResponse =
        await context.encapsulateResponse(httpResponse);

  return encapsulatedResponse;
}
```

Server

CLOUDFLARE

Implementations

Use it today from your machine

Demo

Take your phone

Go to https://ohttp.info

ohttp.info

thibmeu/ohttp-ts - TypeScript

chris-wood/ohttp-go - Go

openpcc/ohttp - Go

martinthomson/ohttp - Rust

apple/swift-nio-oblivious-http - Swift

This list is curated by myself: don’t trust me. It is:

1. Not exhaustive
2. Not a blank cheque for the quality or production readiness

Rough overhead numbers for TypeScript

Step	Time	Wire
Encapsulate Request	8.2ms	55B
Decapsulate Request	5.7ms	-
Encapsulate Response	5.9ms	32B
Decapsulate Response	4.7ms	-

Configuration:

- *1MB request/response body*
- *KEM: DHKEM(X25519, HKDF-SHA256) - KDF: HKDF-SHA256 - AEAD: AES-128-GCM*
- *On my local machine - to re-run yourself*

Conclusion

Where to go next

A few learnings

Seeing less information is more work

From a simple idea, coordination requires clear definition

A lot of standard components, and nascent open source ecosystem

PQ is coming

Make things composable with a clear interface

Questions?