Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Stealthily Hiding the SNI in TLS 1.3

Neimhin Robinson Gunning
Msc. Computer Science — Data Science

Abstract

The Server Name Indication (SNI) extension to TLS 1.3 is sent in cleartext and thus susceptible to Pervasive Monitoring (PM) and censorship. While the draft Encrypted ClientHello (ECH) extension provides a mechanism to hide the SNI and other information from surveillance and censorship tools, it falls short in one of its design goals: “Don’t Stick Out” (RFC 8744). In this work we propose, analyse, and implement a stealthy variant of the ECH extension, designed to prioritise the goal of not sticking out.

Summary of Work Completed

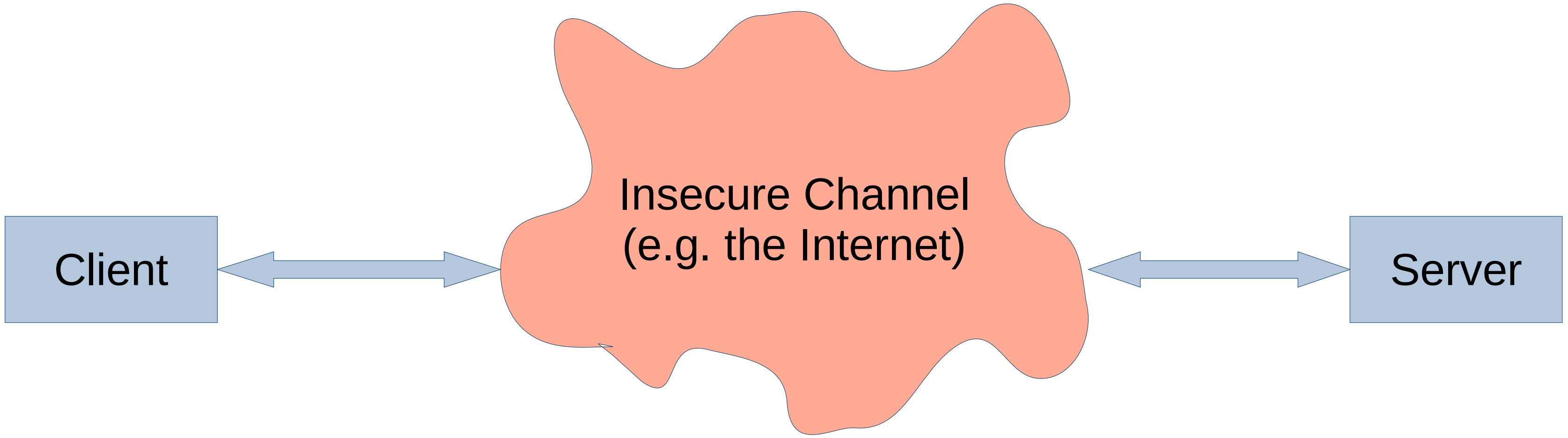
1) Stealthy SNI encryption with assumed OOB shared secret
2) Bootstrapping a shared secret for stealthy SNI encryption with the PSK mechanism

   - Designed a protocol to meet (most) requirements of RFC 8744
   - Implemented an API in OpenSSL
   - Tested the API with the OpenSSL unit test infrastructure

Work Projected to be Completed

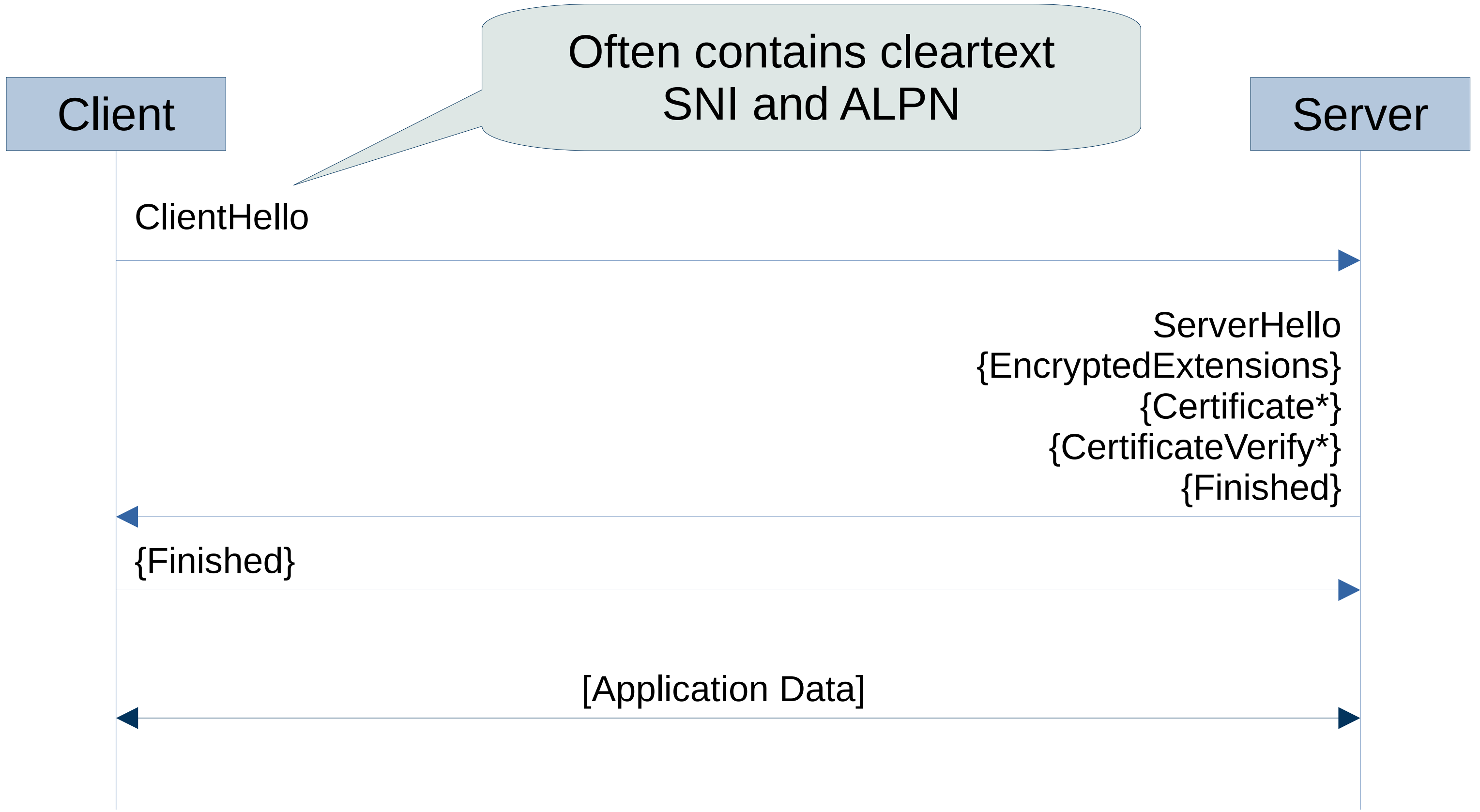
- Implement HPKE-based stealthy encryption
- Implement tests with OpenSSL’s s_server and s_client commands
- Cursory analysis of potential Bleichenbacher-style attacks

TLS 1.3 Overview

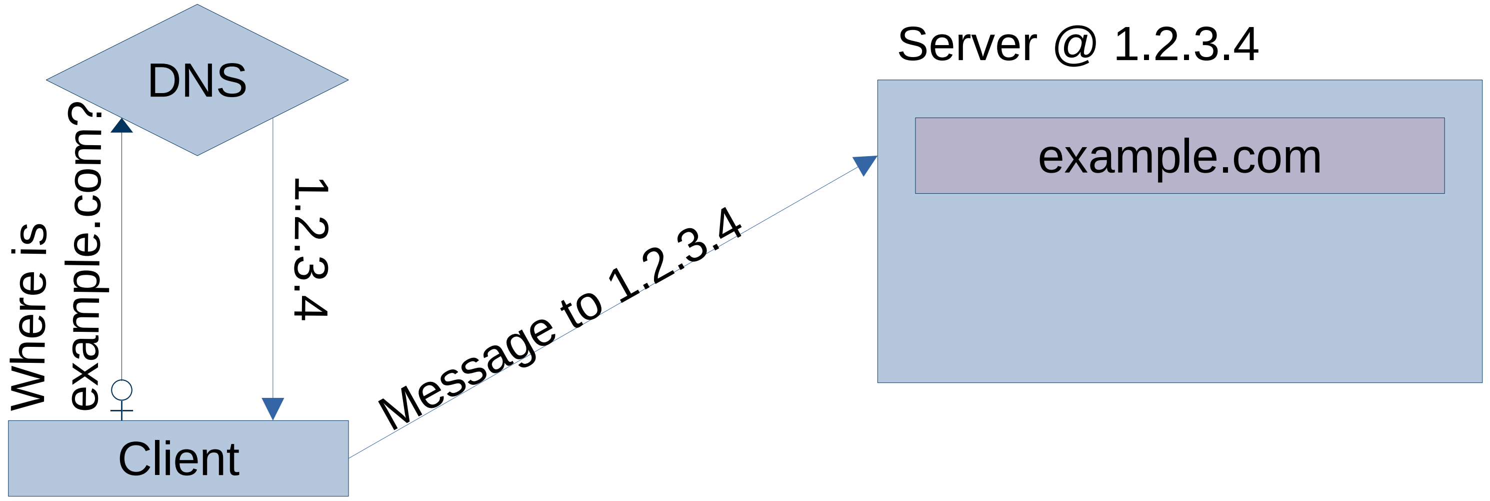


- Provide secret conversation over insecure channel.
    - Bidirectional message confidentiality, integrity, authentication.
- Mandatory server authentication, optional client authentication.
- Definition: the client is the initiating party.

TLS 1.3 Handshake (without HRR)

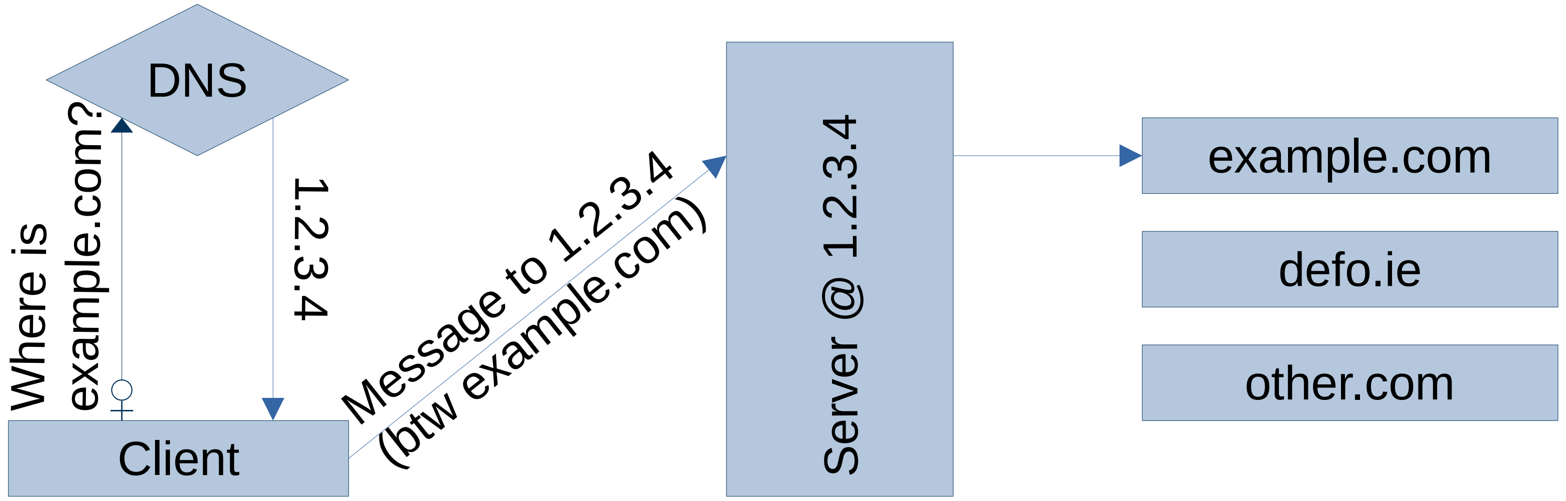


Domain Colocation and the SNI (1/4)



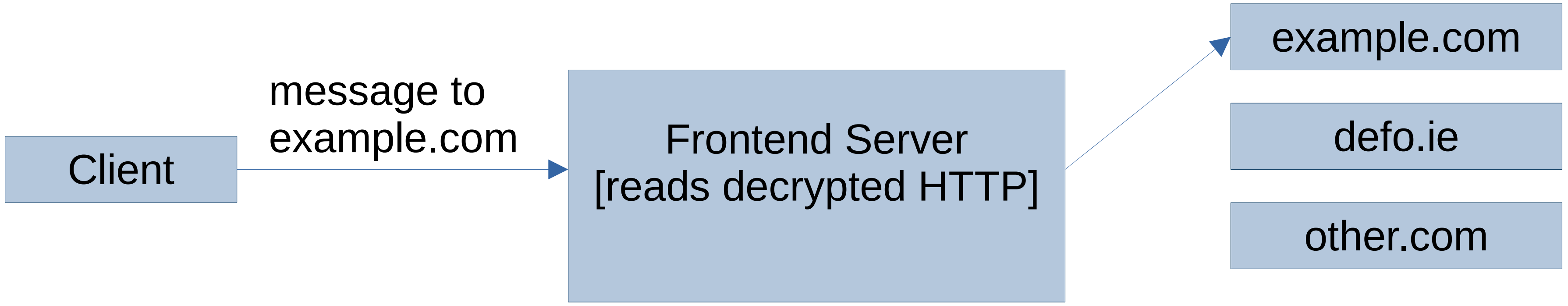
- In the old days a domain name in the Domain Name System (DNS) would map to an Internet Protocol (IP) address that uniquely identified the server.

Domain Colocation and the SNI (2/4)



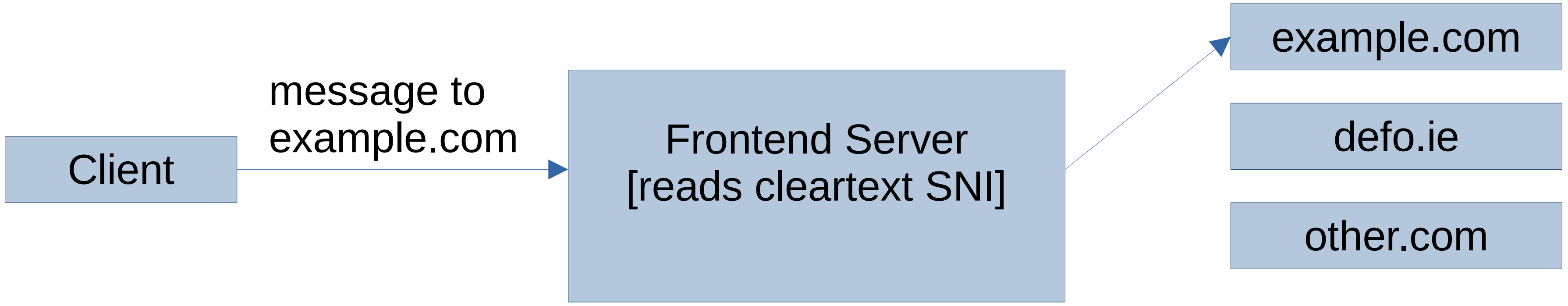
- Nowadays many servers, with different domain names, hosted at a shared IP address (colocation).
  - E.g. large Content Delivery Networks (CDNs) and cloud services host hundreds/thousands of servers on a single IP address.

Domain Colocation and the SNI (3/4)



- (Bad) solution: frontend server terminates TLS
  - Bad because frontend server
    - needs all private keys
    - can snoop on all connections
    - needs another secure connection to backend

Domain Colocation and the SNI (4/4)



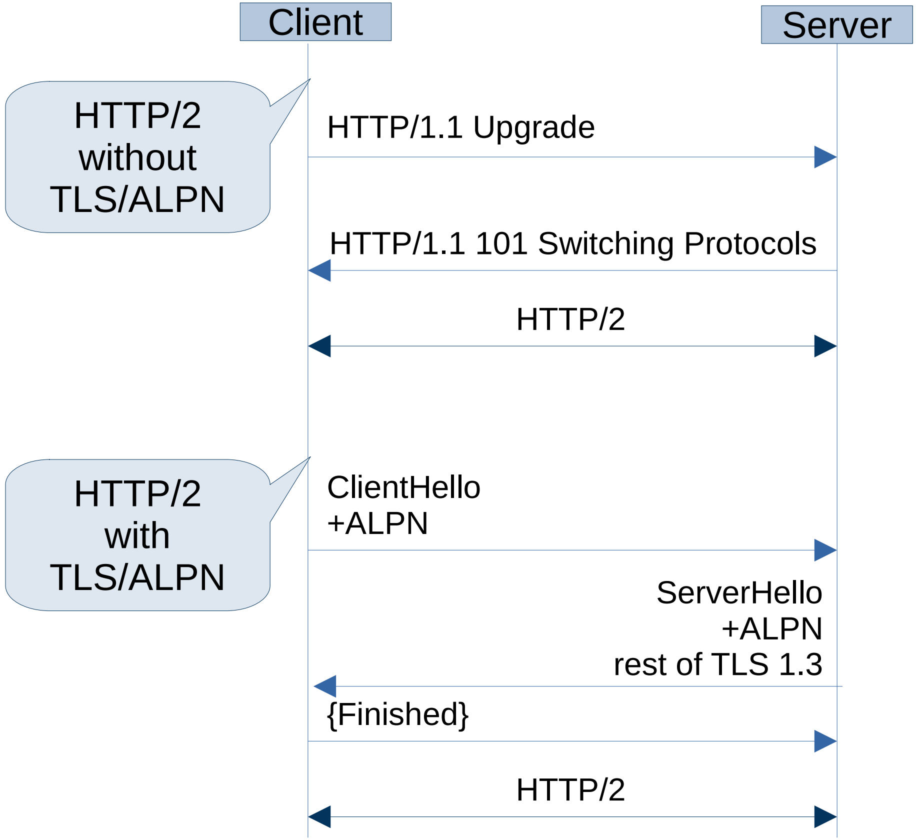
- (Less bad) solution: include cleartext SNI in the TLS 1.3 handshake
  - Still bad because censor/surveiller can read SNI
  - SNI is ubiquitous, but also used for censorship
  - Some security products/practitioners rely on the SNI

Application-Layer Protocol Negotiation

TLS is a transport-layer protocol that protects application-layer protocols.
Examples of Application-Layer protocols are:

- HTTP/1.1
- HTTP/2
- SMTP
- FTP

The ALPN extension lets the client and server negotiate an ALP before any application data is sent.
For HTTP/2 this saves a round-trip.



Application-Layer Protocol Negotiation

Two essential purposes:

- **1:** Facilitate fewer round-trips for application-layer protocols
  - => An ALPN encryption solution that adds round-trips defeats this point
- **2:** Allow server to choose a different authentication certificate for different apps
  - => ALPN also potentially a discriminator for censorship

Servername Encryption Design Goals (RFC 8744)

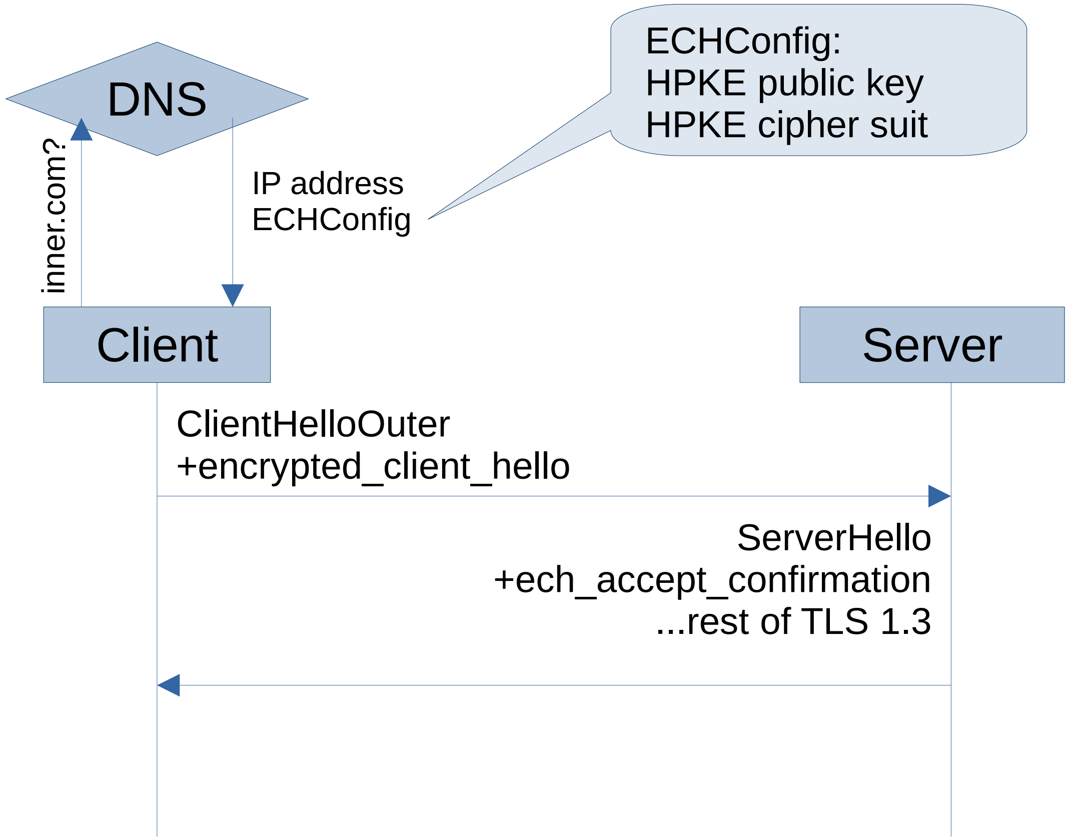
- Mitigate Cut-and-Paste Attacks
- Avoid Widely Shared Secrets
- Prevent SNI-Based Denial-of-Service Attacks
- Do Not Stick Out
- Maintain Forward Secrecy
- Enable Multi-Party Security Contexts
- Support Multiple Protocols
  - Hiding the ALPN
  - Support other transport protocols (TCP, UDP)

  Forerunning design proposal called ECH
  The requirement “Do Not Stick Out” is deprioritised by ECH

ESNI, ECH

- Encrypted Server Name Indication (ESNI) proposed to protect the SNI
  - Evidence of ESNI blocking by the Great Firewall in 2020
- Since evolved to Encrypted ClientHello:
  - Protect SNI, ALPN and more
  - No extra round-trip
  - Use Hybrid Public Key Encryption (HPKE)
  - ECH **sticks out!** The extension identifier is cleartext

How does ECH work?



- Client acquires an ECHConfig out-of-band
  - e.g using DNS-over-HTTPS
- Client encapsulate inner ClientHello with server's public key
- If server accepts respond with cryptographic verification
- This is a depiction of **shared-mode**, but ECH also works in **split-mode**

“Don’t Stick Out”

- i.e. be stealthy
- Make the SNI-protected handshake ‘look’ identical to a normal TLS 1.3 handshake
- Avoid sticking out via traffic patterns, timing, error types

GREASE

- Generate Random Extensions And Sustain Extensibility (GREASE)
- Add extensions with random values to discourage ossification

ECH GREASE

- Run TLS 1.3 but make it look like ECH
  - Use random value for payload
- If ECH GREASE can be widely deployed then normal ECH won’t stick out
- Censors can detect and block ECH GREASE

Our approach:

- Encrypted data looks random
- The ClientHello contains random bytes
- => replace random bytes with encrypted data

We call existing random bytes in the ClientHello ‘cover’

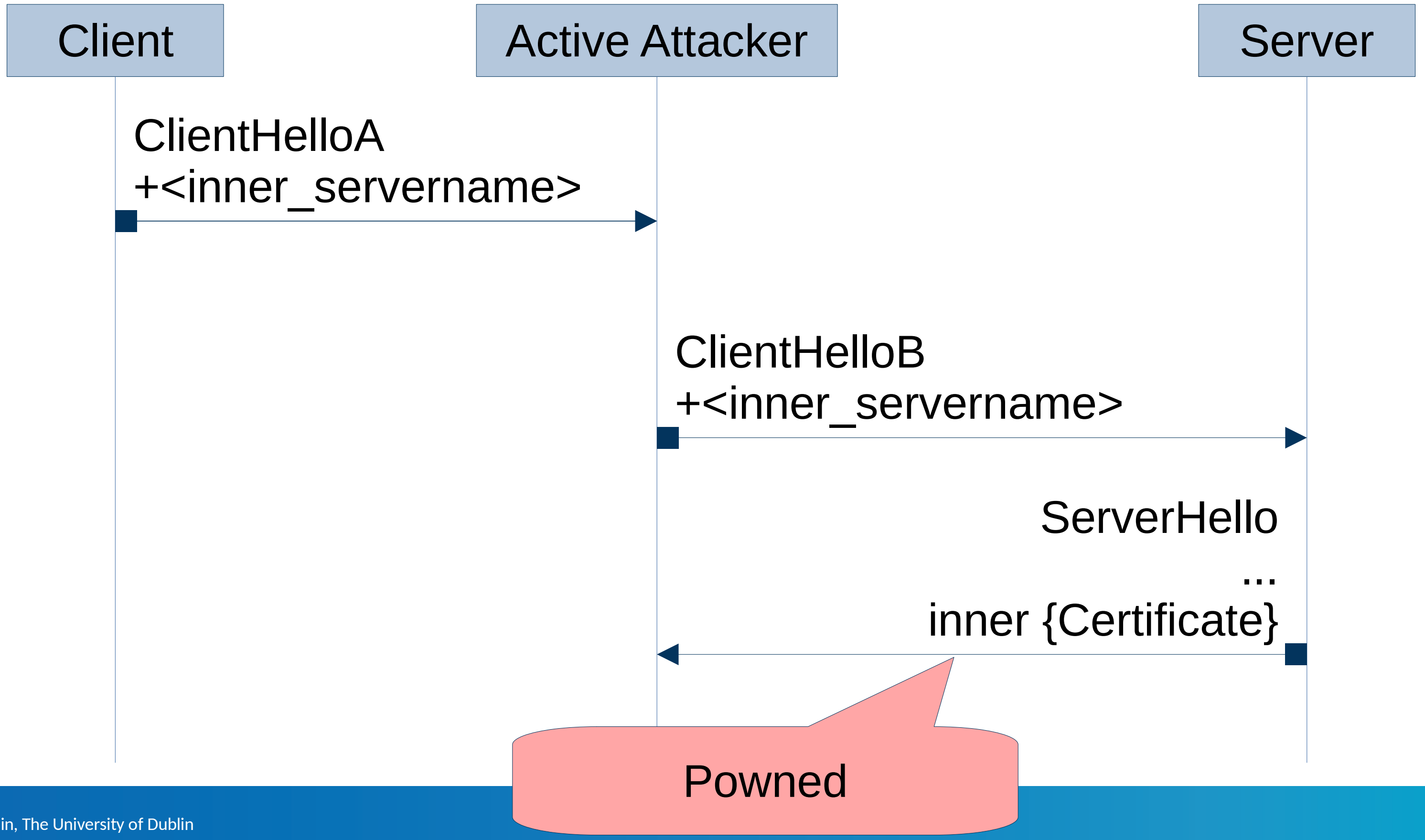
- ClientHello.random => 32 random bytes
- ClientHello.legacy_session_id => 32 random bytes
- padding extension => a handful of bits
- pre_shared_key extension => near arbitrary random bytes
  - used differently by different clients/servers
  - not always present
  - harder to gaurantee ‘not sticking out’ if using pre_shared_key as cover

Naïve stealthy encrypted SNI

- Assume shared secret between client and server.
- Encapsulate inner_servername with AEAD using shared secret.
- Plaintext is inner_servername padded to 36 bytes.
- Use 64 random-looking octets of ClientHello.random and ClientHello.legacy_session_id as cover.
- Hide AEAD nonce (12 bytes), ciphertext (64-12-16=36 bytes), and tag (16 bytes) in cover.

Any problems?

Cut-and-paste against naïve stealthy encrypted SNI



Mitigating cut-and-paste:

- Bind inner servername ciphertext to key_share
  - Use ClientHelloOuter as AAD for encryption
  - Derive encryption key from ClientHelloOuter

SECH 2 Basic Flow

- Assume shared out-of-band long-term secret between client and client-facing server
- Client construct ClientHelloOuterContext
  - Clone of ClientHello but with cover set to all 0s and PSK binders set to all 0s
- Derive a session key from the long-term secret and the ClientHelloOuterContext
- Encapsulate inner SNI and ALPN with AEAD, using ClientHelloOuterContext as AAD
- Replace cover with cipher-text and tag
- If necessary, now compute PSK binders
- Server decapsulates inner SNI and ALPN
  - On decap failure continue with normal TLS 1.3
- On decap success forward ClientHelloInner to backend server
  - ClientHelloInner is the ClientHello but with
    - the cover/ciphertext replaced with the plaintext (no longer looks random)
    - the outer SNI set to all 0s (signals SECH 2 to backend server)
- If necessary exchange HRR and ClientHello2 as normal
- When server sends ServerHello include an 8 octet accept-confirmation signal in the random
  - accept-confirmation is a function of
    - inner transcript
    - session key
- Client verifies accept-confirmation before switching handshake transcript context

Widely shared secret?

Servername Encryption Design Goals (RFC 8744)

Session key is bound to ClientHelloOuterContext

Looks identical to TLS 1.3

+1 Symmetric decryption for **all** server handshaske

- Mitigate Cut-and-Paste Attacks
- Avoid Widely Shared Secrets
- Prevent SNI-Based Denial-of-Service Attacks
- Do Not Stick Out
- Maintain Forward Secrecy
- Enable Multi-Party Security Contexts
- Support Multiple Protocols
  - Hiding the ALPN
  - Support other transport protocols (TCP, UDP)

Nope!

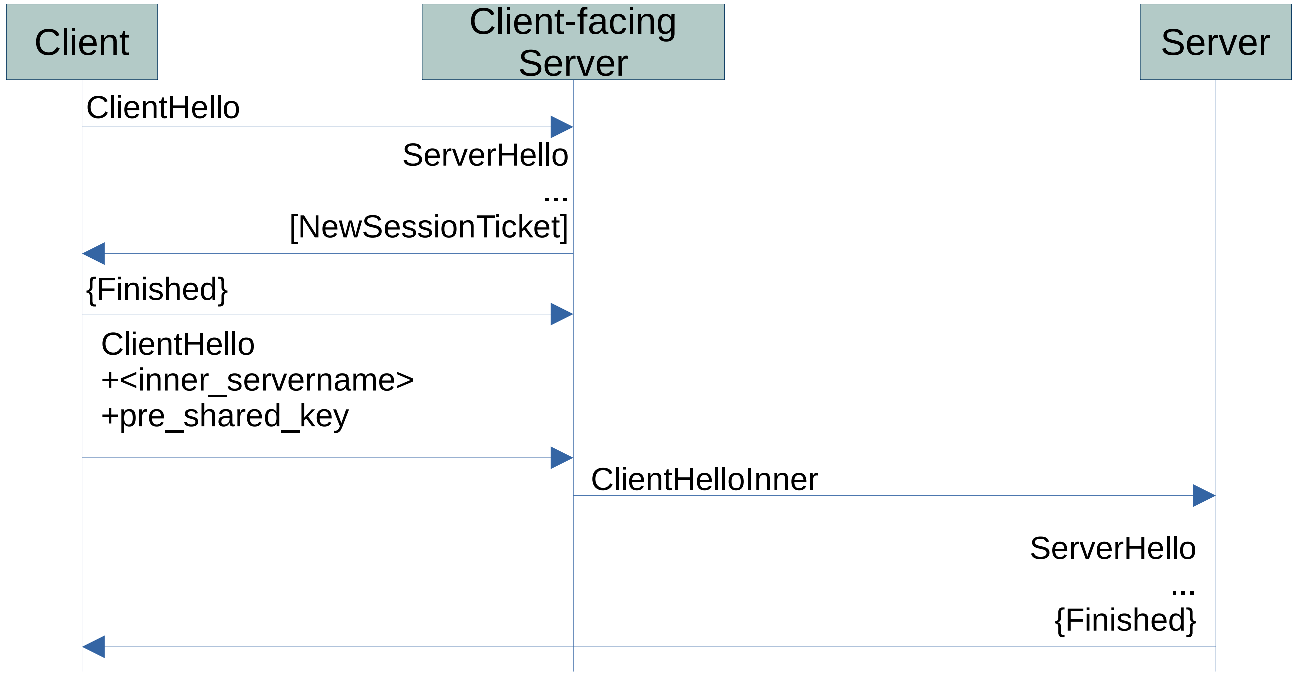
Accept-confirmation authenticates client-facing server

Extensible inner payload (but limited size)

Nope!

SECH 2 PSK Flow

- Client completes normal handshake with outer SNI and acquires a PSK
- Use this PSK as the SECH shared secret to complete the SECH 2 Basic Flow
- Problem: client sends PSK **and** server returns a secret, which might stick out



Work to be Completed

Implement HPKE-based SECH

- Avoid additional bootstrap handshake
- Server publishes an SECHConfig with public key
- Client encapsulates inner SNI and ALPN with public key
- HPKE encapsulation is byte-hungry
  - Do we need more cover?

Implement tests using OpenSSL’s s_server and s_client

Analyse implementations to find Bleichenbacher-style attacks that destroy stealth