OpenSSL CONFERENCE

TLS keyshare caching for faster handshakes

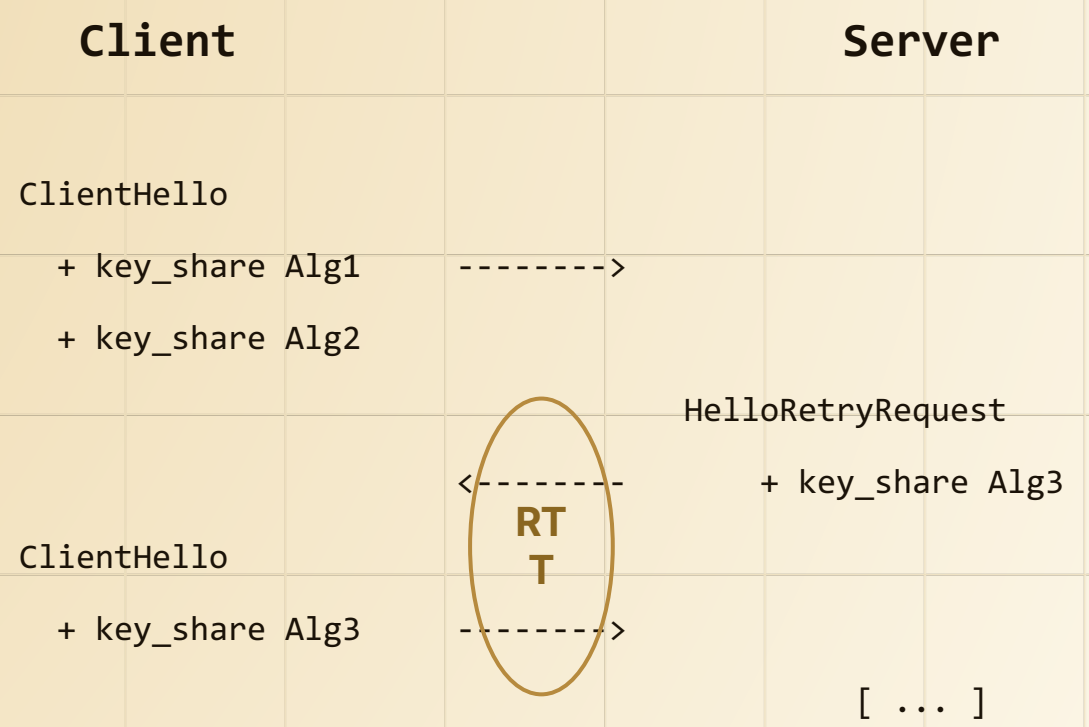
Panos Kampanakis

Principal Security Engineer

AWS

The Problem - TLS 1.3 HRR

- Round-trip delay
- CPU cycles wasted (for the rejected keyshare generation)
- Bandwidth wasted (for the rejected keyshares)



Candidate Use cases

OpenSSL CONFERENCE

Keyshare preferences that do not align between the client and servers.

- Any migration where the client/server preference changes before the peer's
  - ML-KEM-1024 on the CNSA 2.0-compliant server vs X25519MLKEM768 in commercial clients
  - HQC on the server vs X25519MLKEM768 in commercial clients
  - X25519 on the browser vs P256 in legacy server

- E.g. A few AWS regions see **>75% extra round-trips** due to compliance requirements on the server-side.

The Solution: Cache the server keyshare preference on the client

Note #1: Cloudflare's automatic-key-exchange-for-origins did something similar by scanning and "remembering" all origin server keyshare preferences.

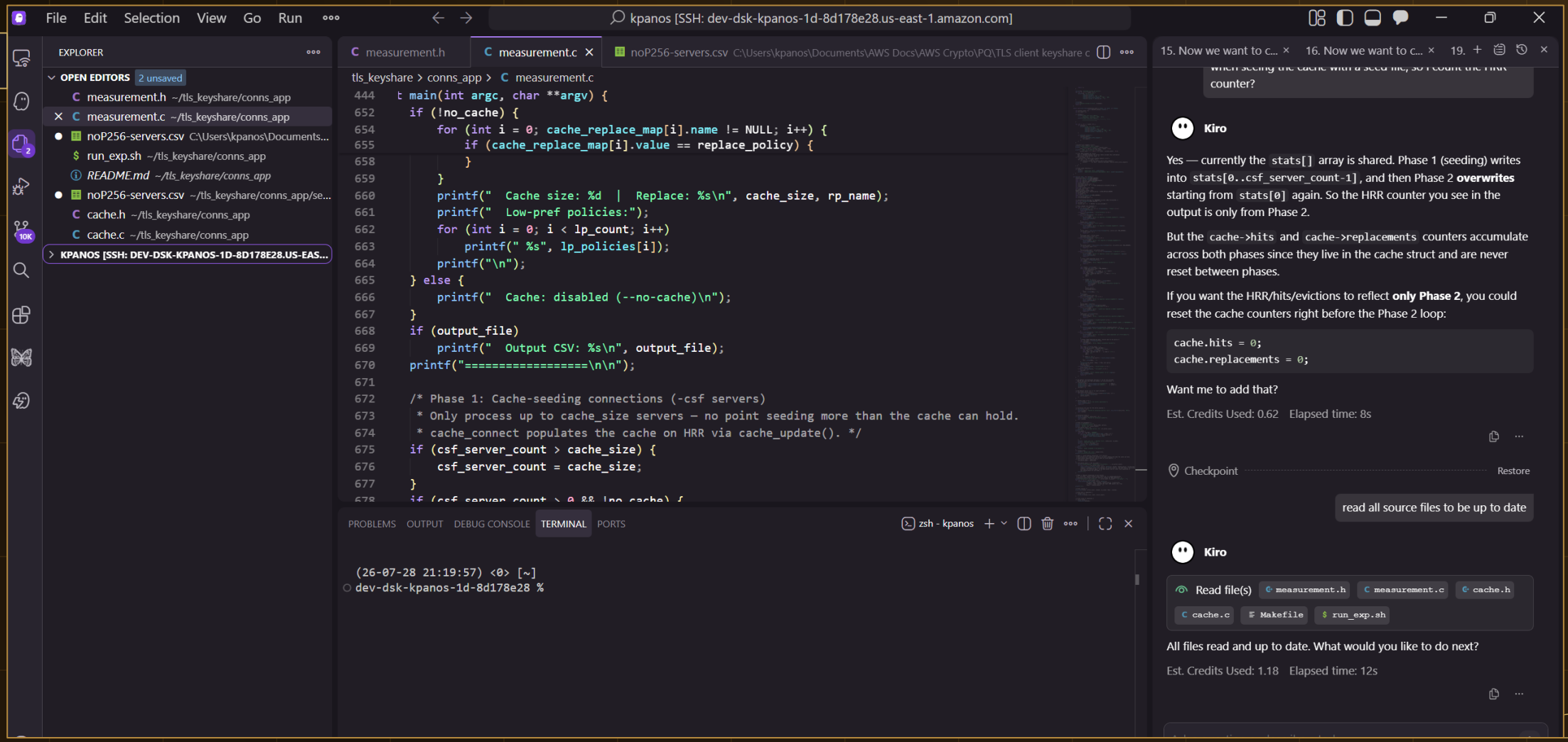
Note #2: IETF draft draft-ietf-tls-key-share-prediction can also address the issue, but requires DNS.

Side-note: But where is the AI in all this?

OpenSSL CONFERENCE

(Every presentation that respects itself must have AI in 😉 ...)

KIRO



Keyshare Caching Mechanism

- Cache `keyshare_cache_t`
  - Linked list of X entries `cache_entry_t`
    - server
    - TLS policy → keyshare

- For every conn to a server
  - Cache hit
    - Use the right server TLS keyshare
  - Cache miss
    - Use the default TLS keyshare
  - Connect. If the conn succeeds with HRR
    - Update cache with server preferred keyshare
      - Cache full
        - Evict a cache entry based
          - Random
          - Freshness
          - Least preferred keyshares
          - [more options]

```
typedef struct {
    cache_entry_t *head;
    int count;
    int csize;
    int hits
    int replacements;
} keyshare_cache_t;

typedef struct cache_entry {
    char hostname[CACHE_MAX_HOSTNAME];
    char s2n_tls_p[CACHE_MAX_POLICY];
    struct cache_entry *next;
} cache_entry_t;
```

Bonus: Could be used to cache
- ICAs per destination → Less bytes on the wire
- PQ Auth server support → Prevent

Experimental Results

Side-note: An Inadvertent Discovery – (Rare) Unnecessary HRRs

OpenSSL CONFERENCE

```
dev-dsk-kpanos-1d-8d178e28 % ./measurement -sf servers/noP256-servers.csv -p P256 --print-servers --cache-size 2000
[...]
Connecting to 382 servers...  [ with only P256 keyshare, advertising support for both P25[illegible]nd X25519 ]
```

Hostname	Success	HRR	KE
www.google.com	YES	YES	x25519
www.facebook.com	YES	YES	x25519
www.googleapis.com	YES	YES	x25519
www.cloudflare.com	YES	YES	x25519
www.youtube.com	YES	YES	x25519
www.instagram.com	YES	YES	x25519
www.linkedin.com	YES	YES	x25519
www.chatgpt.com	YES	YES	x25519

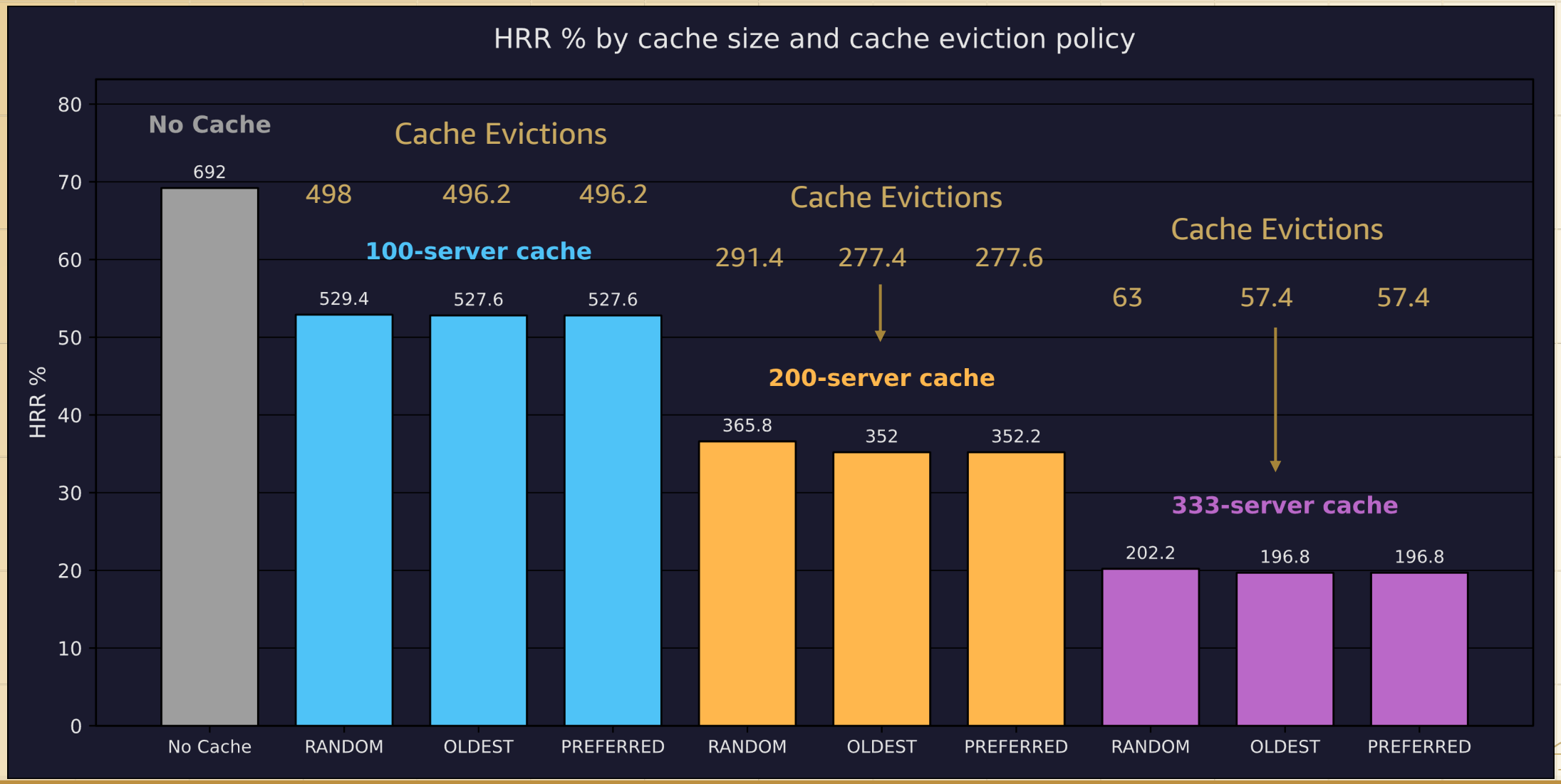
```
[...]
Total servers: 382
  Successful: 382 (100.0%)
  HRRs: 382 (100.0%)
```

When X25519 was not advertised in the TLS `ClientHello`, all these servers completed the handshake fine with P256 without HRRs.

Experimental Results

OpenSSL CONFERENCE

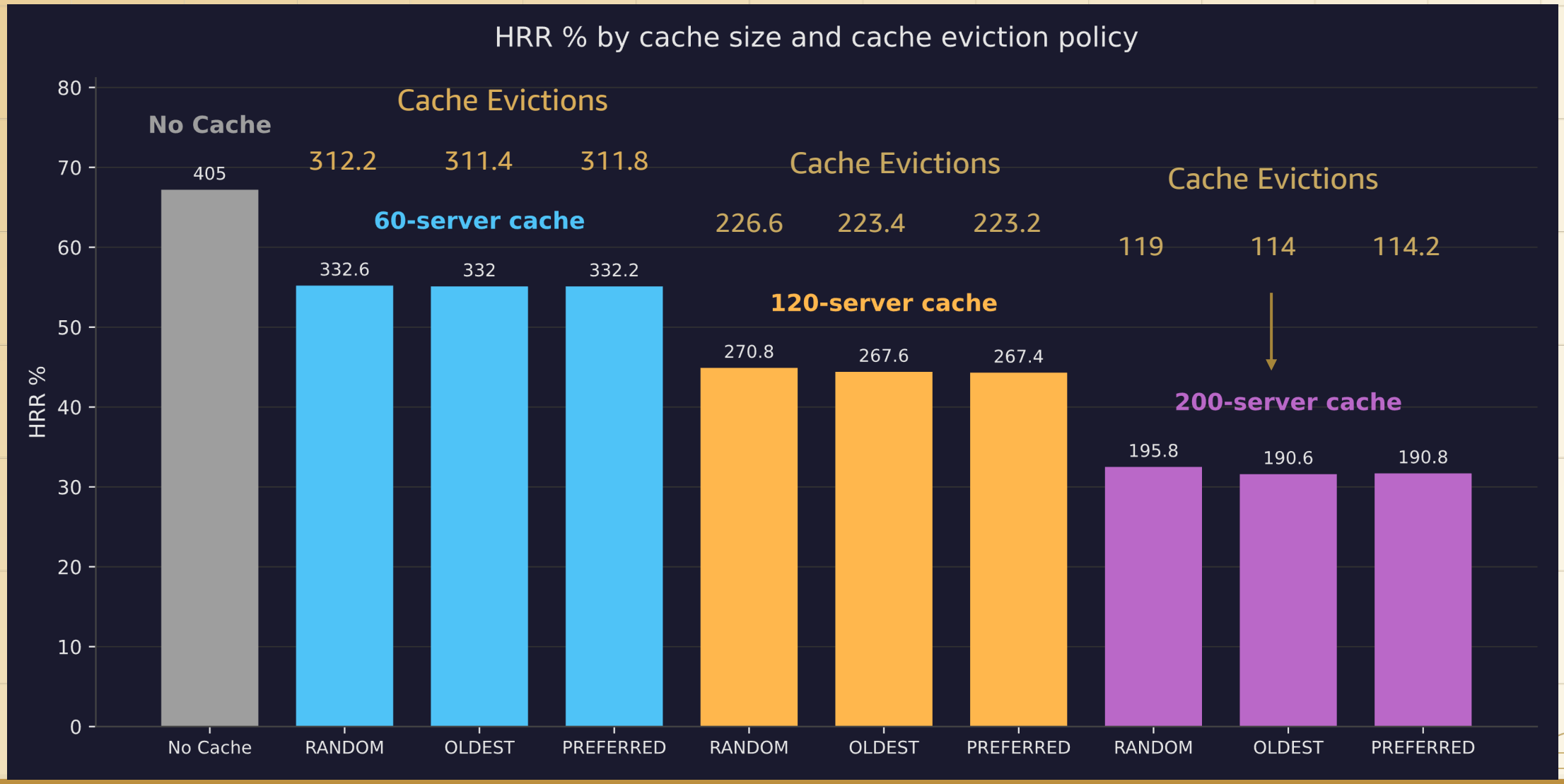
- 1000 servers - 310 no HRR, 690 HRR (310 repetitions)



Experimental Results

OpenSSL CONFERENCE

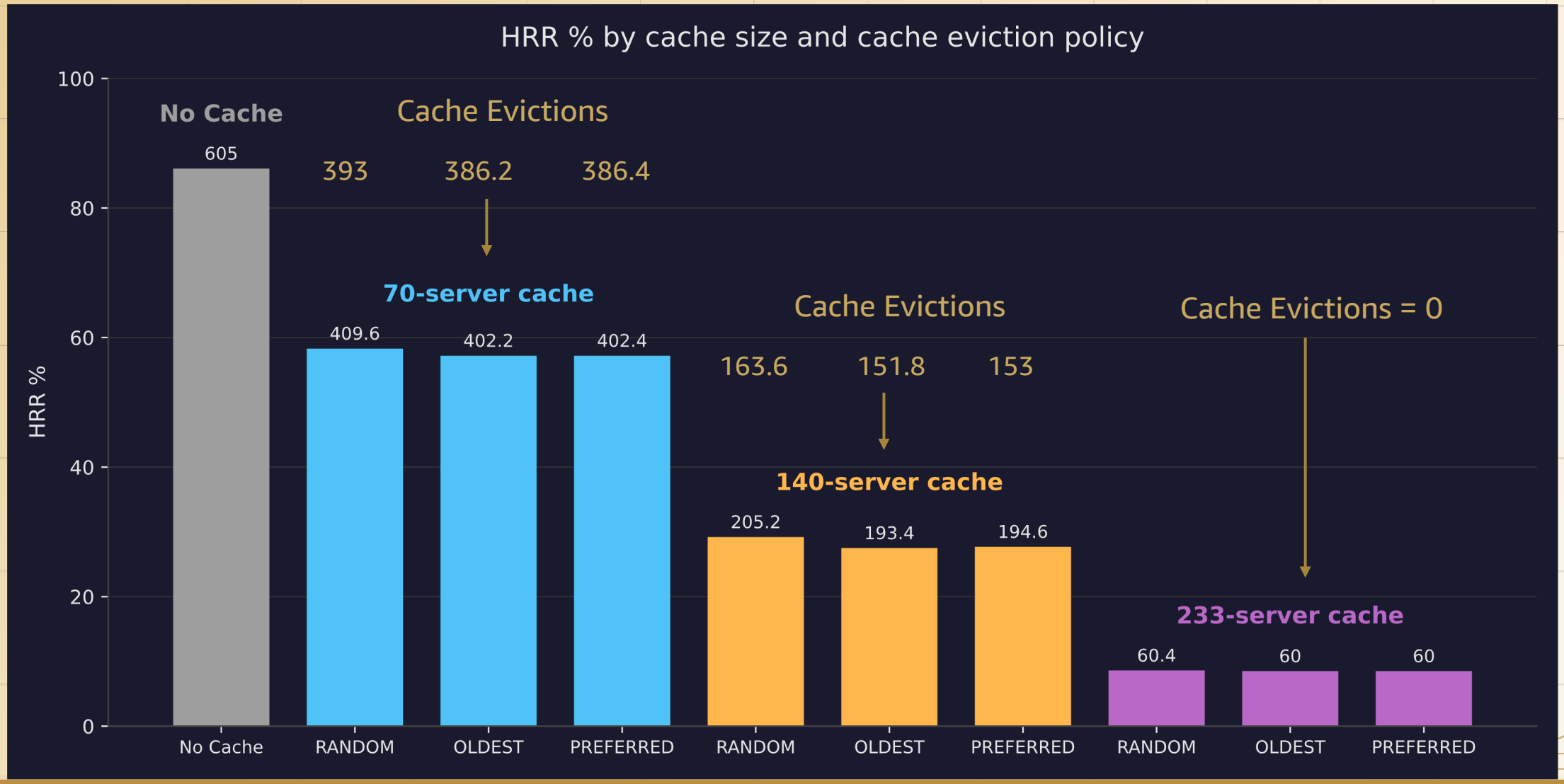
- 600 servers - 200 no HRR, 400 HRR (100 repetitions)



Experimental Results

OpenSSL CONFERENCE

- 700 servers - 100 no HRR, 600 HRR (400 repetitions)



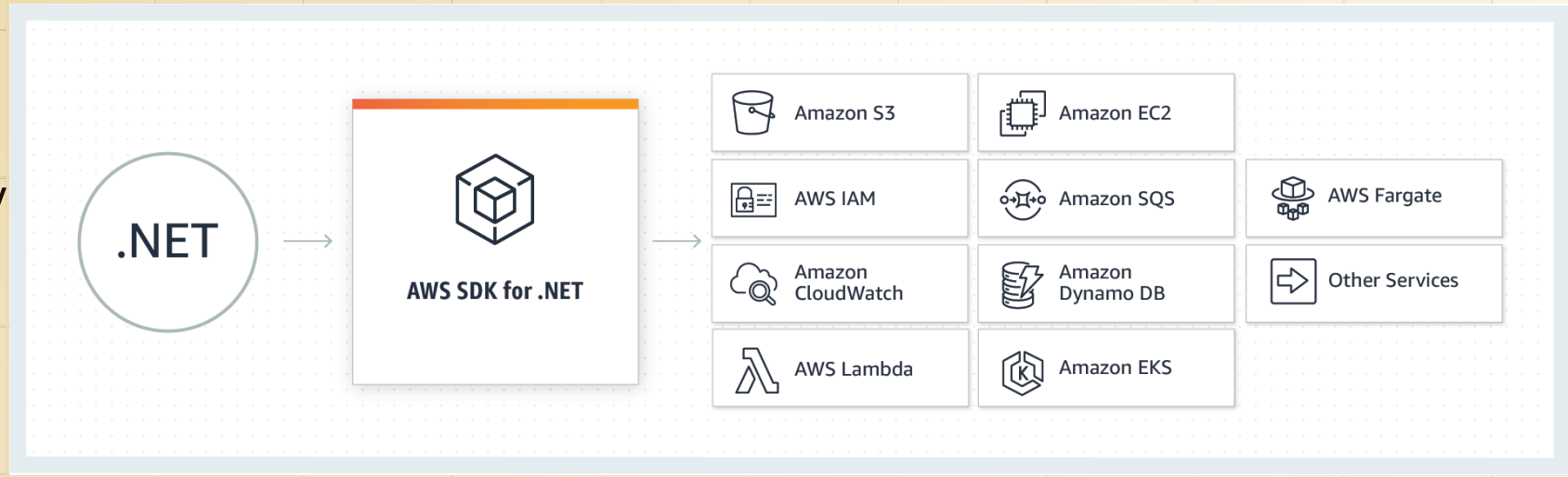
Keyshare Caching Optimization Factors

OpenSSL CONFERENCE

- Cache size
- Cache eviction scheme (Random, Freshness, Preference, etc)
- Server order

Beneficial for applications connecting deterministically to a set of servers.

i.e. AWS SDK applications



Call to Action (for TLS stacks)

OpenSSL CONFERENCE

- OpenSSL
- JSSE / Bouncy Castle
- Go
- Python
- rustls ☑
- .NET

Easy for TLS stacks (e.g. s2n-tls, libssl)

OpenSSL CONFERENCE

Implementation

Files modified (8 files, +120/-17)

File	
`tls/s2n_connection.h`	
`tls/extensions/s2n_client_key_share.c`	
`tls/s2n_server_hello_retry.c`	Update learned ID on HRR (ECC and KEM)
`tls/s2n_resume.c`	Append/read learned ID in client-side session blob; `S2N_STATE_WITH_KEYSHARE_ONLY` format
`tls/s2n_resume.h`	`S2N_STATE_WITH_KEYSHARE_ONLY` enum value; `S2N_TLS12_SESSION_SIZE` +2
`tls/s2n_server_new_session_ticket.c`	Use `session_len` for allocation instead of hardcoded size
`bin/s2nc.c`	TLS 1.3 reconnect path: fall back to `get_session` when no ticket received
`tests/unit/s2n_resume_test.c` + `s2n_session_ticket_test.c`	Updated for new session blob size

The learned keyshare IANA ID is appended to the client-side session blob returned by `s2n_connection_get_session`. This blob is already managed by the application for session resumption.

Similarly easy for libssl type of libraries.
Learned group lives on SSL_SESSION. No changes on client already doing standard client-side resumption. No new public API.
~40 lines across 6 files in aws-lc's ssl/

Rustls already does it

Sharing X25519 setup costs

Background

In TLS1.3, the client starts the key exchange in its first message (the `ClientHe` shares".

The server then evaluates which algorithms it is willing to use, and either uses specific, mutually-acceptable key share.

A `HelloRetryRequest` can be expensive, because it introduces an additional rou

It's therefore advantageous for a client to avoid `HelloRetryRequest`s, by:

- **Having prior knowledge of the server's preferences.** draft-ietf-tls-k
- **Remembering a server's preferences from a previous connection.** server may avoid repeated `HelloRetryRequest`s.
- **Sending many presumptive key shares.** Though there's an obvious t
- **Following ecosystem preferences.** X25519 key exchange is overwhelr

The key shares in a `ClientHello` would look like:

https://rustls.dev/perf/2024-12-17-pq-kx

Improving performance of ServerSessionMemoryCache #1503

Open

jsha opened on Sep 29, 2023 — Contributor

Following up on #1200, I wanted to open an issue to talk about possible performance improvements to ServerSessionMemoryCache, particularly ones that improve worst-case performance.

@dhobsd mentioned using the idea of having high and low water marks so we can be more efficient about cleaning up old sessions. Here's a sketch of that:

The current ServerSessionMemoryCache has a map and a queue, maintained at the same size. When the capacity is hit and the oldest item needs to be expired, that's cheap: pop the oldest item off the queue and delete the corresponding entry from the map. Removing a session from the middle of the queue is expensive, though. We look it up in the hashmap (cheap), then have to find it in the middle of the queue (O(n)) and delete it from the queue (O(n)).

One tweak would be to let the queue grow bigger than the map, up to a point. When we remove a session, we remove it from just the hashmap but allow it to linger in the queue. When capacity of the hashmap is hit, we pop items off the queue until we find one that's still in the hashmap, then delete that from the hashmap. That's still O(n) worst case but it's probably pretty cheap most of the time.

Another approach would be to have a queue of hashmaps (`VecDeque<HashMap<K, V>>`), with each hashmap representing an "epoch". When the last hashmap is full, we pop one off the front (dropping a bunch of old sessions at once), and push a new one onto the back. To look up a session, we look it up in each hashmap in the queue, so we would want to keep the queue short (e.g., around 10). Hashmaps being popped off the front would be partially empty, because many of their sessions would have been redeeemed by then.

https://github.com/rustls/rustls/issues/1503

Takeaways

OpenSSL CONFERENCE

- Client-Server keyshare preference misalignment may trigger unnecessary round-trips.

- Keyshare preference caching during the first connection can be a simple solution.
- Caching is already used in some cases (session resumption, rustls).

- For multiple destination use cases, caching can be optimized based on
  - cache size
  - cache eviction mechanisms and
  - the order of visiting servers

OpenSSL
CONFERENCE

Thank you