



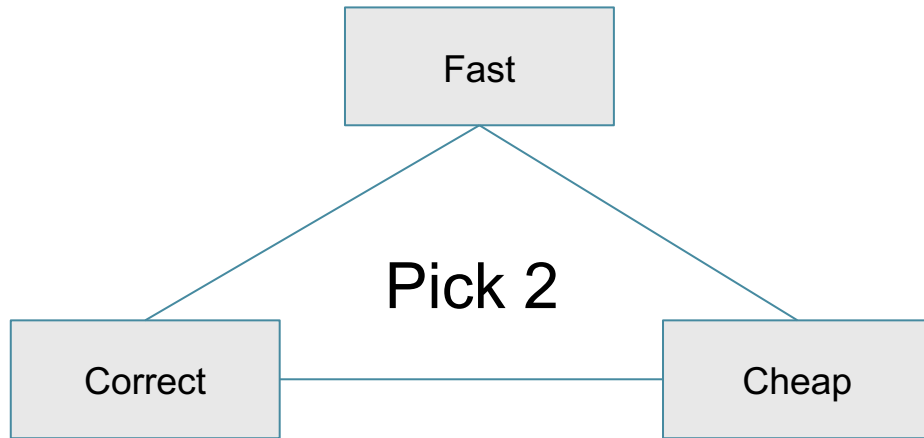
OpenSSL Performance Improvements



OpenSSL development goals

OpenSSL Development

OpenSSL software development suffers from the same constraints that all software does



OpenSSL Development

Where does our current
Development Practice sit in these
constraints?

OpenSSL goal review

- Correct
 - Generally speaking, we do pretty well here. We get our fair share of corner cases that fail and get addressed reasonably quickly, but generally, for most use cases we operate correctly
- Cheap
 - Its open source, and users can choose from a wide array of free and paid offerings, so we meet this goal pretty well.
- Fast
 - Ahem...Moving on.....

OpenSSL: Challenges to fast

Fast would be nice too

Invariably, users would like all 3

- Rapid, ongoing feature development
 - We have lots of community engagement for new features - this makes current performance a moving target
- Bug fixes impact performance - i.e. “we have a race condition, so let's add some locking” - impacts performance in an ongoing fashion.
- Mature, stable codebase - some performance problems are an artifact of large old codebases which require significant refactoring/rewriting - this takes time - Don't expect faster to get here faster.

OpenSSL: Getting faster in parts

So what can we do

- We can find smaller latencies that are hidden inside implementations
- Don't require application behavioral change
- In aggregate on high performance systems, these changes add up
- Some examples of what we will cover today
 - X509 Certificate internal layout and reuse
 - Replacement of locking with atomics
 - Object lifetime guarantees to eliminate refcounting
 - X509 Caching to amortize decoding work

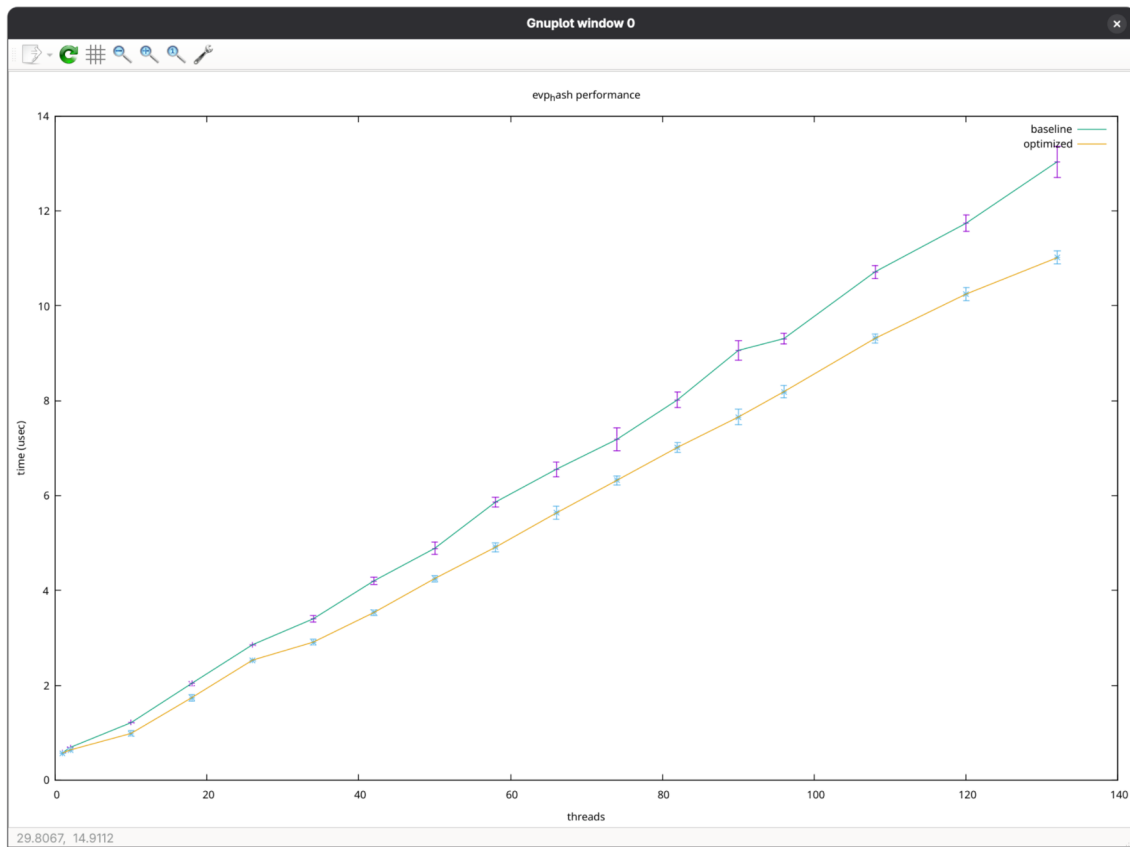
OpenSSL: Locking->atomics

So what can we do

- Most prevalent in our method store
 - The method store caches all our provider methods
 - Each lookup required a read lock
 - Each addition required a write lock
 - We hit this path (particularly the read lock) frequently
 - Even a read lock requires a write to update our readers
 - This is typically a partial or full memory barrier
- Replace all those locks with atomic reads and stores/exchanges
 - Atomic exchanges limit memory writes to be done only on updates
 - Lookups are now pure reads (no synchronization required)
 - Refcounting (already there) ensures memory doesn't get freed under us

OpenSSL: Locking->atomics

So what can we do



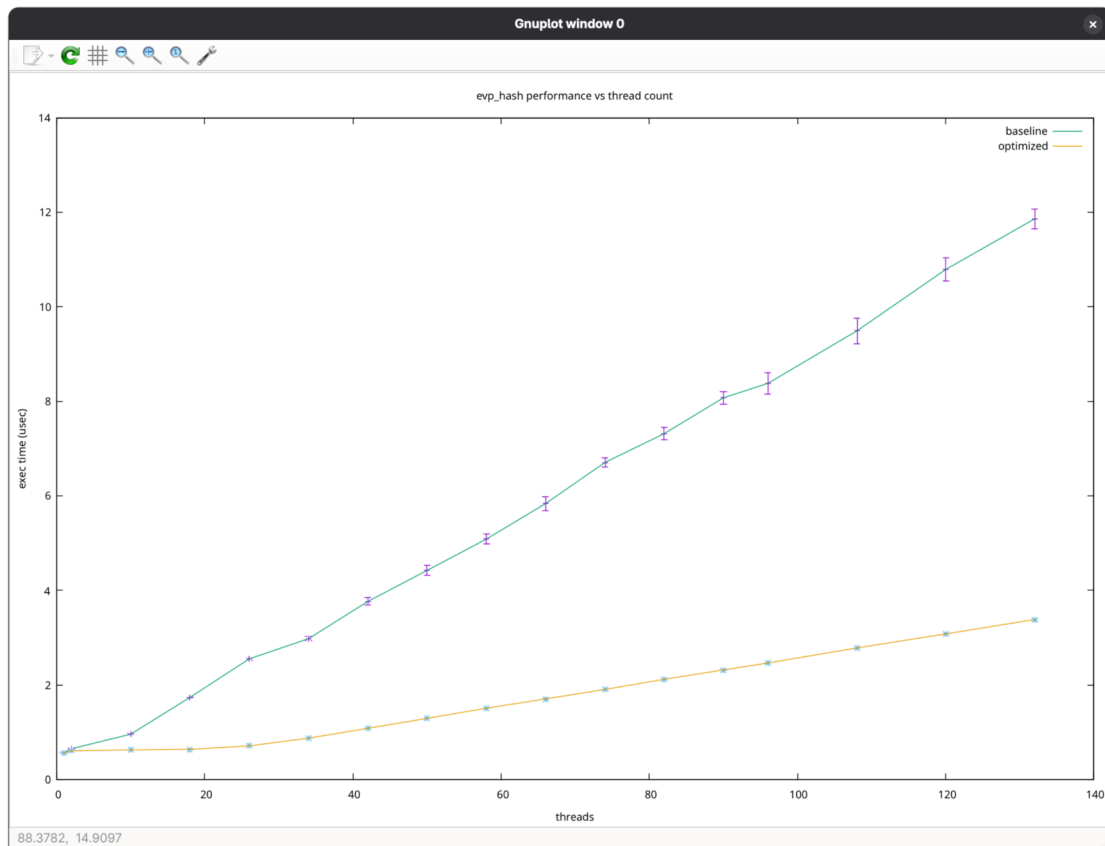
OpenSSL: Refcount elimination

So what can we do

- Again with the method store
 - When allocating an algorithm, we generally need to track that is in use
 - REFCOUNTING!
 - That's expensive. Everytime someone takes an alg from the same libctx, we have to up the refcount, and drop it on free
 - That's a lot of synchronized integer updating
- We tried avoiding this about 4 different ways
- We thought, what if we could guarantee that an algorithm stayed available until such time as the libctx it was fetched from was itself removed?
 - So we did
 - Algorithm removal (on provider unload), archived an alg, making it unfindable, but still memory resident until the libctx was freed
 - This let us eliminate almost all refcounting for algs.

OpenSSL: Refcount elimination

So what can we do



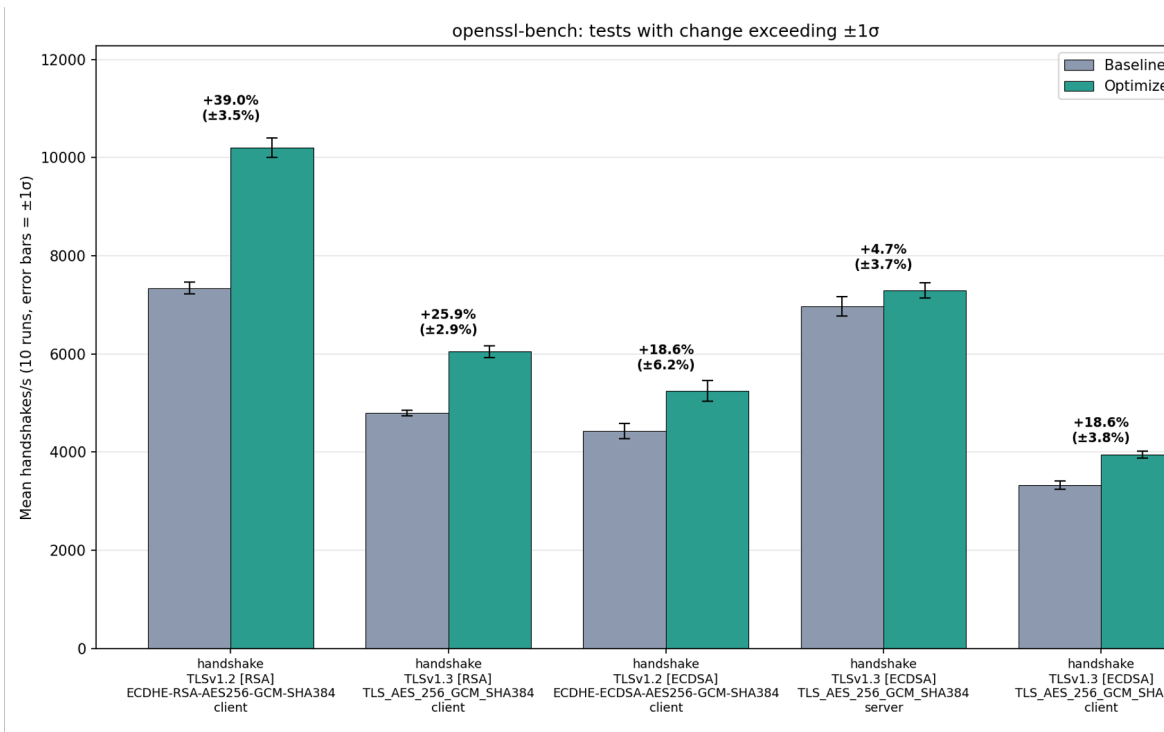
OpenSSL: TLS X509 Caching

So what can we do

- TLS on the receipt of a certificate chain from the server requires that they all be parsed from ASN.1 to their internal representation
 - This is slow. Lots of parsing, lots of small allocations
- We don't have to do that
- We introduced a small cache per SSL_CTX object which stores the last several certificates received, indexed by a hash
- On subsequent connects, instead of reparsing, we just lookup and reuse the previously parsed certificate
- While we're at it, we parse and cache certificate extensions, so the certificate can effectively be immutable higher up the stack.

OpenSSL: TLS X509 Caching

So what can we do



OpenSSL: X509 kind of sucks:

X509 Sucking is bad

- Why do we need to cache X509 at all?
- <https://cryptography.io/en/stable/statements/state-of-openssl/>
 - End to end X.509 path validation 60% faster by not using OpenSSL's X509 parsing

“The fact that we are able to achieve better performance doing our own parsing makes clear that doing better is practical. And indeed, our performance is not a result of clever SIMD micro-optimizations, it's the result of doing simple things that work: we avoid copies, allocations, hash tables, indirect calls, and locks — none of which should be required for parsing basic DER structures.”

OpenSSL: X509 Challenges

Can we do better?

X509 parsing currently happens by building a “cert_info” structure which holds most of the things we are used to looking at. An “extensions cache” is *at some time* populated, once, later. Values in cert_info are allocated copies, sometimes changed from the values in the certificate - but why?

- We think parsing the der is expensive and hard
- We want a convenient representation, a time_t, an integer, a string with a \0 byte on the end, an artisanally rolled generic policy or name constraints structure - and sometimes this is ensconced in public API.
- What if we don't have the DER because we are making a new X509?
- X509 may mutate on demand via a “hidden” api - you have no idea when -> the extensions cache.

Make X509 usable without cert_info

The Road To Doing Better

- Parsing basic ASN1 structures from DER is really not that complicated with reasonable tooling (CBS/CBB PACKET/WPACKET) Just do it.
 - Keep one copy of the original bytes of the certificate
 - When sensible, hand back slices of those bytes, rather than making copies.
 - When sensible, Parse the slices of the bytes as needed, instead of always.
 - If we think we need a further fancy “parsed object” - **measure first**
- Save the DER when parsing X509's - use the der bytes directly for X509 validation / read only access
- Allow for const X509* and const ASN1_STRING*
- Don't parse DER up front if we are not using it.
- If we truly need pre-parsed values up front repeatedly, ok, but measure.
- Get a handle on the mutability contract, document it, and allow cert_info to be built when public accessors are used.

Zero Copy X509 for handshaking

- Ongoing experiment, work not yet landed or ready.
- Goal: Make X509 handshaking possible without using
 - X509->cert_info
 - X509->cache
- Use only a read-only “view” of the DER bytes instead.
 - Bytes from cert are consumed as read-only slices
 - Some values parsed up front
 - Most parsed on demand if complicated
- Preliminary Results for X509 Handshake speedup:

	cert_info always		view only, no cert_info	
RSA-2048				
parse	2 x 15 + cache 8 = 38		2 x 2 + key 5 = 9	
verifier	~21		~15-20	
total	~59		~25-30	(~50..~55%)
P-256 (static groups)				
parse	38 + key 6 = 44		4 + 6 = 10	
verifier	~23		~15-20	
total	~67		~25-30	(~60%)
Ed25519				
parse	38 + 2 = 40		4 + 2 = 6	
verifier	~21		~15-20	
total	~61		~20-25	(~65%)

