Mathematics and development of fast TLS handshakes

Alexander Krizhanovsky
Tempesta Technologies, Inc.

ak@tempesta-tech.com
Web content delivery & protection

- 2013: **WAF** development by request of Positive Technologies “Visionar” from Gartner magic quadrant’15
  - Web attacks
  - L7 HTTP/HTTPS DDoS attacks
- Nginx, HAProxy, etc. - perfect HTTP proxies, not HTTP filters
- Netfilter works in TCP/IP stack (softirq) => **HTTP(S)/TCP/IP stack**
- **Tempesta FW:**
  - hybrid of HTTP accelerator & firewall
  - embedded into the Linux TCP/IP stack
Tempesta TLS


- Part of Tempesta FW, an open source Application Delivery Controller
- Open source alternative to F5 BIG-IP or Fortinet ADC
- “TLS CPS/TPS” is a common specification for network security appliances & ADCs
Linux kernel TLS handshaks

- Very fast light-weight Linux kernel implementation
  - ...even for session resumption
  - there is modern research in the field
- Resistant against DDoS on TLS handshakes (asymmetric DDoS)
- Privileged address space for sensitive security data
  - Varnish: TLS is processed in separate process Hitch
    http://varnish-cache.org/docs/trunk/phk/ssl.html
  - Resistance against attacks like CloudBleed
Why NIST p256?

- **ECDSA**
  - RSA and NIST curves p256, p384, and p521 are the only allowed for CA certificates
    
    https://cabforum.org/baseline-requirements-documents/
  - P256 is the fastest NIST curve
  - P521 isn’t recommended by IANA
    
    https://www.iana.org/assignments/tls-parameters/tls-parameters.xml#tls-parameters-8
  - RSA is slow and vulnerable to asymmetric DDoS
    

- **Curve25519**
  - Much faster than NIST p256
  - In practice for ECDHE only
TLS libraries performance issues

- Copies, memory initializations/erasing, memory comparisons
- `memcpy()`, `memset()`, `memcmp()` and their constant-time analogs
- Many dynamic allocations
- Large data structures
- Some math is outdated

```
12.79% libc-2.24.so _int_malloc
9.34% nginx(openssl) __ecp_nistz256_mul_montx
7.40% nginx(openssl) __ecp_nistz256_sqr_montx
3.54% nginx(openssl) sha256_block_data_order_avx2
2.87% nginx(openssl) ecp_nistz256_avx2_gather_w7
2.79% libc-2.24.so _int_free
2.49% libc-2.24.so malloc_consolidate
2.30% nginx(openssl) OPENSSL_cleanse
1.82% libc-2.24.so malloc
1.57% [kernel.kallsyms] do_syscall_64
1.45% libc-2.24.so free
1.32% nginx(openssl) ecp_nistz256_ord_sqr_montx
1.18% nginx(openssl) ecp_nistz256_point_doublex
1.12% nginx(openssl) __ecp_nistz256_sub_fromx
0.93% libc-2.24.so __memmove_avx_unaligned_erms
0.81% nginx(openssl) __ecp_nistz256_mul_by_2x
0.75% libc-2.24.so __memset_avx2_unaligned_erms
0.57% nginx(openssl) aesni_ecb_encrypt
0.54% nginx(openssl) ecp_nistz256_point_addx
0.54% nginx(openssl) EVP_MD_CTX_reset
0.50% [kernel.kallsyms] entry_SYSCALL_64
```
The source code
https://github.com/tempesta-tech/tempesta/tree/master/tls

- **Still in-progress**: we implement some of the algorithms on our own
- Initially the fork of mbed TLS 2.8.0 (https://tls.mbed.org/) - x40 faster!
  - very portable and easy to move into the kernel
  - cutting edge security
  - too many memory allocations (https://github.com/tempesta-tech/tempesta/issues/614)
  - big integer abstractions (https://github.com/tempesta-tech/tempesta/issues/1064)
  - inefficient algorithms, no architecture-specific implementations, ...
- We also take parts from WolfSSL (https://github.com/wolfSSL/wolfssl/)
  - very fast, but not portable
  - security https://github.com/wolfSSL/wolfssl/issues/3184
ECDSA & ECDHE mathematics: Tempesta TLS, OpenSSL, WolfSSL

- **OpenSSL 1.1.1h**
  - 256 bits ecdsa (nistp256) 36473 sign/s
  - 256 bits ecdh (nistp256) 16620 op/s

- **WolfSSL (current master)**
  - ECDSA 256 sign 43260 ops/sec (+19%)
  - ECDHE 256 agree 40878 ops/sec (+146%)

- **Tempesta TLS (full TLS handshake operation)**
  - ECDSA sign (nistp256): ops/s=38393
  - ECDHE srv (nistp256): ops/s=13418

- **OpenSSL & WolfSSL don’t include ephemeral keys generation**
  - (one more $m \times G$ operation)
Demo!

- Tempesta TLS, Nginx-1.14.2/OpenSSL-1.1.1d, Nginx-1.17.8/WolfSSL
- TLS 1.2
  - full handshakes
  - abbreviated handshakes
- `tls-perf`
  https://github.com/tempesta-tech/tls-perf
  - establish & drop many TLS connections in parallel
  - like TLS-THC-DOS, but faster, more flexible, more options
Data for proprietary vendors

- BIG-IP is only **30-50%** faster than Nginx/OpenSSL/DPDK
  
  https://www.youtube.com/watch?v=Plv87h8GtLc

- Avi Vantage (VMware) makes ~**2000** handshakes/second per 1CPU
  
  https://avinetworks.com/docs/latest/ssl-performance/
Why faster?

- No memory allocations in run time
- No context switches
- No copies on socket I/O
- Less message queues
- Zero-copy handshakes state machine
- State of the art cryptography mathematics
Elliptic curve cryptography

- **Secp256r1**: \( y^2 = x^3 - 3x + b \) defined over the field \( \text{GF}(p) \)
  
  \[ p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \]

- The group law
  
  - negatives: if \( P = (x, y) \), then \(-P = (x, -y)\)
  
  - addition: \( R = P + Q \)
  
  - doubling: \( R = P + P = 2*P \)

- **ECDSA**: \( k \) – secure random, \( G \) – known point
  
  \( k \times G \) is used for the signature

- **ECDH**: \( d \) – private key, \( Q \) – public key
  
  shared secret: \( d \times Q \)
Point multiplication

OpenSSL: “Fast prime field elliptic-curve cryptography with 256-bit primes” by Gueron and Krasnov

- $Q = m \cdot P$ - the most expensive elliptic curve operation

  ```python
  for i in bits(m):
    Q ← point_double(Q)
    if $m_i == 1$:
      Q ← point_add(Q, P)
  ```

- Point multiplications in TLS handshake:
  - known point multiplication: precompute the table for doubled $G$
  - perfect forward secrecy ECDHE: generate keys $G \cdot d$ ($d$ – random)
  - handshake: 2 known & 1 unknown point multiplications
Point representation and coordinate systems

http://www.hyperelliptic.org/EFD/g1p/auto-shortw-jacobian.html


- **Jacobian coordinates** (rough estimations)
  - conversion overhead: \(39 \times M + 4 \times S + 3 \times I\) (for \(w(-\text{indow}) = 4\))
  - point addition (**mixed**) - \(8 \times M + 3 \times S\), doubling - \(2 \times M + 4 \times S\)

- **Affine coordinates** (rough estimations)
  - point addition - \(13 \times M + 4 \times S\), doubling - \(4 \times M + 5 \times S\)

- **NIST 256 bits**, \(D = 256 / w = 64\)
  Comb rounds (addition & doubling):
  \[64 \times (10 \times M + 7 \times S) \ll 64 \times (17 \times M + 9 \times S)\]
Point addition

http://www.hyperelliptic.org/EFD/g1p/auto-shortw-jacobian.html#addition-add-2007-bl

- ex. addition in Jacobian coordinates (cost: \(11M + 5S\))
  \(A = (x_1, y_1, z_1), B = (x_2, y_2, z_2),\) then \(C = A + B = (x_3, y_3, z_3)\) is

  \[
  U_1 = X_1Z_2^2 \\
  U_2 = X_2Z_1^2 \\
  S_1 = Y_1Z_2^3 \\
  S_2 = Y_2Z_1^3 \\
  H = U_2 - U_1 \\
  R = S_2 - S_1 \\
  z_3 = HZ_1Z_2 \\
  x_3 = R^2 - H^3 - 2U_1H^2 \\
  y_3 = (U_1H^2 - x_3)R - S_1H^3
  \]
The cost

- Modular multiplication (\(M\)) is the most expensive basic scalar operation.
- Modular squaring (\(S\)) is faster than \(M\), usually 0.8\(M\) (Montgomery) (0.9 for optimized FIPS due to more expensive modular reduction).
- Modular inversion (\(I\)) is very expensive, about 100\(M\).
Modular arithmetics

- **ex. prime field** $F(29)$
  - **addition:** $17 + 20 = 8$ since $37 \mod 29 = 8$
  - **subtraction:** $17 - 20 = 26$ since $-3 \mod 29 = 26$
  - **multiplication:** $17 \times 20 = 21$ since $340 \mod 29 = 21$
  - **inversion:** $17^{-1} = 12$ since $17 \cdot 12 \mod 29 = 1$

- **Montgomery reduction** (the most used)
  - there is some overhead, but each modular operation is cheaper

- **FIPS reduction**
  - Can be faster if small number of modular operations is used
  - There are optimization techniques, e.g. “Low-Latency Elliptic Curve Scalar Multiplication” Bos, 2012
  - But still about 65% slower than Montgomery reduction
Montgomery multiplication in P256


- Fast 256-bit integer multiplication with modular reduction on P256
- \( a, b < m \) \((m - \text{modulus P256})\)
- Set \( n = 2^{256} \)
- Transform multipliers to Montgomery domain (overhead):
  \( a' = a \mod m \quad b' = b \mod m \)
- Fast multiplication with reduction: \( u = a'b'/n \mod m \)
  - compute only 256 bits of \((a'b' + (-m^{-1}a'b' \mod n)m)/n\)
  - if \( u > m \), then \( u \leftarrow u - m \) \(\) (unconditionally, carry as a mask)
- Convert to ordinary number: \( v = un^{-1} \mod m \)
The math layers

- Different multiplication algorithms for fixed and unknown point
  - "Efficient Fixed Base Exponentiation and Scalar Multiplication based on a Multiplicative Splitting Exponent Recoding", Robert et al 2019

- Point doubling and addition - everyone seems use the same algorithms

- Jacobian coordinates: different modular inversion algorithms
  - “Fast constant-time gcd computation and modular inversion”, Bernstein et al, 2019

- Modular reduction for scalar multiplications:
  - Montgomery has overhead vs FIPS speed: if we use less multiplications it could make sense to use different reduction method FIPS (seems deadend)
  - “Low-Latency Elliptic Curve Scalar Multiplication” Bos, 2012

=> Balance between all the layers
Example of math layers balancing

- For $w=5$ we need 52 point additions for an unknown point multiplication
- Jacobian coordinates addition takes $11M + 5S$
- Affine-Jacobian coordinates addition takes $8M + 3S$
  - about 4.4M cheaper if $S = 0.8M$
  - requires 2 coordinates normalizations for Comba precomputation
  - coordinates normalization: $2^{w-1} * (6M + 1S) + 1I$
- Almost the same for $S = 0.9M$ and fast inversion $I < 100M$
- Montgomery arithmetics ($S = 0.8M$):
  - ECDHE +28% and ECDSA +6% performance
Side channel attacks (SCA) resistance

- Timing attacks, simple power analysis, differential power analysis etc.

- Protections against SCAs:
  - Constant time algorithms
  - Dummy operations
  - Point randomization
  - e.g. modular inversion 741 vs 266 iterations

- **RDRAND** allows to write faster non-constant time algorithms
  - **SRBDS** mitigation costs about 97% performance

https://software.intel.com/security-software-guidance/insights/processors-affected-special-register-buffer-data-sampling?fbclid=IwAR1ifj3ZuAtNOabKkj3vFIltBLSvOnMqlxH2l-QeN5KB-aji54J1BCJa9lLk
https://www.phoronix.com/scan.php?page=news_item&px=RdRand-3-Percent&fbclid=IwAR2vmmR_Lir oekUuw7KMRaHB7KThpqz0tI1fX2GCW3HAVwt5Kb1p9xpLKo
Memory usage & SCA

- ex. ECDSA precomputed table for fixed point multiplication
  - mbed TLS: ~8KB dynamically precomputed table, point randomization, constant-time algorithm, full table scan
  - OpenSSL: ~150KB static table, full scan
  - WolfSSL: ~150KB, direct index access (fixed in the new version) https://github.com/wolfSSL/wolfssl/issues/3184

=> 150KB is far larger than L1d cache size, so many cache misses:
Big Integers (aka MPIS)

“BigNum Math: Implementing Cryptographic Multiple Precision Arithmetic”, by Tom St Denis

- All the libraries use them (not in hot paths), mbed TLS overuses them
- linux/lib/mpi/, linux/include/linux/mpi.h

```c
typedef unsigned long int mpi_limb_t;
struct gcry_mpi {
    int allocated;       /* array size (# of allocated limbs) */
    int nlimbs;          /* number of valid limbs */
    int nbits;           /* the real number of valid bits (info only) */
    int sign;            /* indicates a negative number */
    unsigned flags;
    mpi_limb_t *d;       /* array with the limbs */
};
```

- Need to manage variable-size integers
  => size-specific assembly implemetations
Easy assembly

// a := a + b
// x[0] is the less significant limb,
// x[1] is the most significant limb.
void s_mp_add(unsigned long *a, unsigned long *b) {
    unsigned long carry;
    a[0] += b[0];
    carry = (a[0] < b[0]);
}

// Pointer to a is in %RDI, pointer to b is in %RSI
movq (%rdi), %r8
movq 8(%rdi), %r9
addq (%rsi), %r8  // add with carry
addc 8(%rsi), %r9  // use the carry in the next addition
movq (%r8), (%rdi)
movq (%r9), 8(%rdi)
Open questions and further research

- **Ice Lake** CPUs have negligible downclocking on **AVX-512**
  https://travisdowns.github.io/blog/2020/08/19/icl-avx512-freq.html

- Parallel Montgomery computations
  J.W.Bos, "Montgomery Arithmetic from a Software Perspective", 2017
  - SIMD multiplications & squarings of two and more products
  - Interleaved Montgomery multiplications

- Better methods for point multiplications
Going to the Linux kernel upstream

- Details and the discussion
  - https://github.com/tempesta-tech/tempesta/issues/1433

- Server-side only

- The full TLS handshake is in `softirq` (just like TCP)
- Fallback to a user-space TLS library on ClientHello
- Batches of handshakes in **1 FPU context**
TODO

- More cryptography mathematics performance optimizations
  https://github.com/tempesta-tech/tempesta/issues/1064
  https://github.com/tempesta-tech/tempesta/issues/1335

- **TLS 1.3**
  https://github.com/tempesta-tech/tempesta/issues/1031

- Moving to the kernel asymmetric keys API
  https://github.com/tempesta-tech/tempesta/issues/1332

- The Linux kernel `/crypto` API performance issues
  - SHA-256 (crucial for TLS handshake) **30-100%** slower than OpenSSL
    https://github.com/tempesta-tech/tempesta/issues/1483
  - Extra copying and memory allocations in kTLS
    https://github.com/tempesta-tech/tempesta/issues/1064
Netdev papers about Tempesta TLS

Thanks!

Contact us if you’re interested in fast Linux kernel TLS!

https://github.com/tempesta-tech/tempesta

ak@tempesta-tech.com