Performance study of kernel TLS handshakes

1st Alexander Krizhanovsky, 2nd Ivan Koveshnikov
Tempesta Technologies, Inc.
Seattle, USA
ak@tempesta-tech.com, ik@tempesta-tech.com

Abstract

Tempesta TLS, a part of Tempesta FW, implements TLS handshakes in the Linux kernel and focuses on performance to filter out application layer DDoS attacks. We started development from the fork of mbed TLS library, but it took significant effort to make it fast, so we ended up with full reworking of the library. The design of Tempesta TLS was discussed in our previous work [17].

The main focus of this paper is to explore how much performance we can get for TLS handshake itself and the whole application, on example of HTTPS server, by moving TLS handshake into the kernel space. While performance optimization of the TLS handshakes mathematics is still in progress, we observed that Tempesta TLS can establish 40-80% more TLS connections per second than OpenSSL/Nginx and provide up to x4 lower latency in some tests.

In this paper we study software optimizations of server-side full TLS 1.2 handshakes using ECDHE and ECDSA on the NIST P-256 curve as well as abbreviated TLS 1.2 handshakes. We conclude the paper with the architecture and the user space API proposals for the Linux kernel upstream.

Keywords

TLS, Linux kernel, fast computations on elliptic curves

User space TLS handshakes

TLS handshake is one of the slowest part of an HTTPS transaction on a web accelerator or a load balancer. TLS 1.2 session resumption and 0-rtt mode of TLS 1.3 solve the problem for returning users, but new users still cause significant load onto a server. Moreover, some modes, e.g. RSA handshakes or TLS 1.2 session resumption, require move intensive computations on the server side than on the client side. These two factors make TLS handshakes an attractive target for asymmetric application level DDoS attacks [17].

Tempesta FW [18] is an open source hybrid of a web accelerator and an application level firewall embedded into the Linux TCP/IP stack to reach maximum performance and flexibility. The main target for Tempesta FW is to provide an open source alternative to proprietary solutions like F5 BIG-IP and cloud based software stacks like CloudFlare. Such solutions must efficiently process massive legitimate traffic as well as to filter out various DDoS and web attacks.

Tempesta TLS [17] is a high performance TLS implementation and is the part of Tempesta FW project. Tempesta TLS is a Linux kernel module and at the moment is tightly coupled with the main Tempesta FW module. However, in this paper we discuss required changes of the module to include it into the main Linux kernel tree.

In this work we evaluate how much performance we can get by moving TLS handshake into the kernel space. While our development work is still in progress we see that Tempesta TLS establish 40-80% more TLS connections per second in comparison with OpenSSL/Nginx. We also cover the modern research and CPU technologies, which are still not employed in TLS libraries such as OpenSSL or WolfSSL, but can significantly improve TLS handshakes performance.

Besides higher performance, the kernel TLS handshake implementation allows to separate the management of private keys and certificates from a web accelerator working process. In this scheme the worker process doesn't have access to the security sensitive data stored in the kernel space. A separate, privileged, process can be introduced to load and manage TLS keys and certificates in the Linux kernel. This is a best practice for large secure web clusters [8], but it is still unavailable for small installations without loss of performance.

KPTI impact on performance

KPTI (Kernel Page Table Isolation) is a mitigation technology against recent Meltdown [27] vulnerability. Two sets of address space page tables are maintained in the kernel. One is "full" and contains both the kernel and the user space addresses. The other contains user space addresses and only the minimal set of kernel-space mappings. Every syscall or interrupt switches the minimal user space page table to "full" kernel copy and switches them back on returning to the user space [16]. This causes a notable overhead for syscall-heavy and interrupt-heavy workloads [22, 30].

An in-kernel TLS handshake implementation does not need to switch between the kernel and user spaces, so KPTI is not involved. Our goal was to see how performance of a user-space HTTPS server suffers from KPTI.

We expected to see high KPTI impact on the performance due to many network I/O system calls and a lot of memory allocations. But our measurements showed less than 4% performance drop on system with security mitigations enabled.

TLS	w/o KPTI	w/ KPTI	Delta
1.2	8'735	8'401	-3.82%
1.3	7'734	7'473	-3.37%

TLS handshake involves not so many network I/O system calls, so the performance impact was not very high. All the hottest functions are related to cryptographic operations and memory allocations. Network I/O is not very high. Glibc caches memory allocations to reduce the number of system calls. Thus, the overall kernel overhead was not high enough to cause a severe performance gap. Web server functions are also not noticeable in the perf report since no HTTP message exchange happens in this scenario. At the below is CPU utilisation by the hottest functions.

Functions	Overhead
libcrypto.so: __ecp_nistz256_mul_montx, __ecp_nistz256_sqr_montx, sha256_block_data_order_avx2, ecp_nistz256_avx2_gather_w7, OPENSSL_cleanse, ecp_nistz256_ord_sqr_montx, ecp_nistz256_point_doublex, __ecp_nistz256_sub_fromx, __ecp_nistz256_mul_by_2x, ecp_nistz256_point_addx, ecp_nistz256_point_add_affinex, aesni_ecb_encrypt, BN_num_bits_word, EVP_MD_CTX_reset	30.7%
libc.so: _int_malloc, _int_free, malloc, malloc_consolidate, cfree, __memmove_avx_unaligned_erms, __memset_avx2_unaligned_erms	13.2%
kernel: do_syscall_64, entry_SYSCALL_64, prepare_exit_to_usermode, syscall_return_via_sysret, common_interrupt	4.5%
nginx: -	0%

Since performance drop for the user-space HTTPS server is not high in this scenario, user-space HTTPS server can not benefit a lot by disabling KPTI.

SRBDS impact on performance

Mitigation against the special register buffer data sampling (SRBDS) attack [12] can be applied only as a microcode update. The mitigation affects RDRAND processor operation performance which is used in the handshake processing. Performance impact is higher for TLS 1.3 protocol, telling that RDRAND operation is used in TLS 1.3 more frequently than in TLS 1.2.

TLS	w/o SRBDS	w/ SRBDS	Delta
1.2	8'735	8'281	-5.20%
1.3	7'734	6'605	-14.60%

Both the user-space and kernel-space TLS handshakes suffer from SRBDS mitigation, unlike KPTI there is no possibility to avoid extra overhead by moving code to the kernel.

The kernel and TLS versions impact on performance

As Tempesta TLS was initially based on Linux kernel 4.14 and will be ported on the next LTS release after Linux 5.7, we also compared TLS handshake performance on that kernel versions. All mitigations available in both kernels was switched on.

Kernel	Handshakes/s	95P Latency, ms
TLS 1.2 New sessions		
4.14	7'624	498
5.7	7'284	466
	-4.5%	-6.4%
TLS 1.2 Session Resumption		
4.14	19'203	246
5.7	17'452	112
	-9%	-54%
TLS 1.3 New sessions		
4.14	7'147	315
5.7	6'811	466
	-4.7%	+47%
TLS 1.3 Session Resumption		
4.14	6'472	287
5.7	6'183	342
	-4.4%	+19%

The 5.7 kernel in all tests makes 4.5-9% less handshakes in second than the 4.14 kernel. It's also interesting that the latency on TLS 1.2 is significantly lower on 5.7 kernel while TLS 1.3 shows the opposite picture.

As only handshake performance is evaluated, no application data is sent and RTT between our servers is too small for real life scenario, TLS 1.3 can't benefit from 0-RTT capability in this test. Moreover, TLS 1.3 handshake performance is always lower than TLS 1.2 for 7% in our tests. This happens because TLS 1.3 performs more encryption/decryption and hashing operations. Also TLS 1.3 doesn't show significant performance boost on session resumption, instead it even loses 9%. Unlike in TLS 1.2, the master key is not directly reused in TLS 1.3, instead a unique shared secret is generated and combined with the master key [35]. This performance trade-off allows forward secrecy and saves TLS 1.3 connection from the master key compromission.

FPU state manipulations

The Linux kernel provides kTLS [34] acceleration of TLS AES-GCM encryption and decryption: now sendfile() can transfer encrypted TLS payload without going to the user space for encryption. The implementation uses x86-64 CPU extensions involving the FPU, which is responsible for SIMD CPU extensions and which is usually unavailable for the Linux kernel code. To use the extensions the FPU state must be saved with kernel_fpu_begin() function and then restored with kernel_fpu_end().

The FPU state manipulations aren't cheap. In our previous work [17] we evaluated the cost of the FPU manipulations using a micro-benchmark [19]: copying of 1500 bytes with the proper FPU state saving and restoring takes x4.78 more time than the raw copying.

TLS handshake in a full HTTPS transaction

Another subject for the research is how big is the TLS handshake overhead in a short HTTPS transaction. Long living connections can not benefit from handshake optimisations since the handshake takes a relatively small part of a communication process. Short living connections shows the opposite picture: the handshake overhead is huge enough to dominate in the communication process.

To prove that an HTTPS transaction performance depends on handshake processing we compared a server performance in the scenarios when a client opens and immediately drops a TLS connection right after a TLS connection was established (no data transfer), right after a 1KB server response transmission, or right after 10KB server response transmission. In all the scenarios the client sends HTTP/1.1 request with the usual headers set for popular browsers. Firstly, we used unencrypted HTTP connections, then HTTPS. In order to reduce HTTP-related processing and evaluate only TLS handshake overhead, the server was configured to serve from the web cache.

Handshakes/s	Resp 1Kb/s	Resp 10Kb/s
Plain HTTP		
N/A	77'109	55'060
HTTPS with TLS 1.2 (new sessions)		
7'225	6'402	6'238
HTTPS with TLS 1.2 (session resumption)		
17'274	13'472	12'630

The TLS handshake overhead is high enough to reduce performance on encrypted short living connections by x12 for 1KB responses, and by x8.5 for 10KB responses. TLS session resumption reduces the difference by x2, but the handshake still costs a lot. While in unencrypted HTTP connections increasing of a response size lowers RPS by 28%, the effect is much less on TLS connections: 2.5% for the full handshake and 6,25% for the abbreviated one.

Compare the kernel- and user-space TLS handshakes

To compare in-kernel and user-space TLS handshakes performance it is required to benchmark basic cryptographic operations first. For elliptic cryptography they are ECDSA, Elliptic Curve Digital Signature Algorithm, and ECDHE, Elliptic-curve DiffieHellman key agreement protocol.

	ECDSA, op/s	ECDHE, op/s
OpenSSL 1.1.1d	36'472	16'619
WolfSSL*	41'527	55'548
Tempesta TLS**	27'261	6'690

* - before the non constant-time fixed point multiplication was fixed [37].
** - including ephemeral keys generation.

In the benchmarks OpenSSL and WolfSSL measure ECDSA signing and ECDHE shared secret generation only. Meantime, for Tempesta TLS we measure the full operations, including the ephemeral keys generation, which for ECDHE involves an additional point multiplication, the slowest operation. However, despite multiple optimisations in crypto layer (described in sections below) and heavier computations in the benchmark, we still get lower performance than OpenSSL and WolfSSL. We reference WolfSSL here since we borrow some code from this library as well.

	RPS	Latency, ms		
		Avg	95P	Max
1 CPU VM: TLS 1.2 New sessions				
Nginx/OpenSSL	1'545	1340	1643	**2154**
Tempesta FW	**2'180**	**307**	**757**	2941
	+41%	**-77%**	**-53%**	+36%
2 CPU bare-metal: TLS 1.2 New sessions				
Nginx/OpenSSL	7'426	**418**	**467**	12681
Tempesta FW	**7'921**	622	1280	**3460**
	+6%	+48%	+170%	**-72%**
2 CPU bare-metal: TLS 1.2 Session resumption				
Nginx/OpenSSL	19'203	**74**	**246**	**925**
Tempesta FW	**35'263**	77	268	28262
	+83%	+4%	+9%	+3000%

In-kernel handshakes allow to serve more requests per second in all test cases. While we saw 40-80% performance improvement for the benchmarks in a virtual environment and session resumptions on bare-metal, we observed only 6% performance improvement for new TLS sessions in a bare-metal environment. We believe that Tempesta TLS shows 6% only performance improvement for the bare-metal case due to slower cryptography routines. During the tests we had no hardware with virtual APIC support and the TLS benchmarks produce many small network packets, which are the known problem [14] for the current virtualization solutions.

In the virtual environment the average and 95 percentile latencies are 50-70% lower for in-kernel handshakes. Situation is opposite in bare-metal environment: Nginx/OpenSSL always delivers lower average and 95 percentile latency, 50-170% lower for new sessions and only 4-9% for session resumption.

In some test we observer huge values of tail latency for the in-kernel TLS handshakes, up to 3000% larger values in the worst case. These tail latency spikes are the other side of the coin with Tempesta TLS processing in SoftIRQ: in the virtual environment there are less number of VM-exit events for Tempesta TLS, so we see the better performance, but we may drop more packets under heavy load on bare-metal setups. We investigate the possible ways to fix the problem [24].

Performance tools and environments

In our performance evaluation of TLS handshakes we used 2 benchmark tools and 2 different environments.

The baseline

Nginx 1.19.1 and OpenSSL 1.1.1d were used as the reference for our evaluations. Tempesta FW is a reverse proxy, so we focused on server side performance. In TLS handshakes elliptic curves significantly outperform RSA on the server side, so we chose to concentrate on elliptic curves only. RSA and NIST curves p256, p384, and p521 are the only allowed for CA certificates [4]. NIST curves p384 and p521 seem not used widely: p384 is not optimized in OpenSSL at all and p521 is even not recommended by IANA [11]. Thus, NIST secp256r1 used in the study.

The most recent stable kernel version was used - 5.7. In the most tests all security mitigations were applied (including microcode updates). The only exception is the tests for mitigations performance evaluation, where mitigations were disabled by mitigations=off kernel parameter and microcode updates were disabled as well.

Benchmarking tools

To evaluate the raw speed on TLS handshakes without extra logic, we used tls-perf [33] tool, which just establishes and immediately drops many TLS connections in parallel. The tool can establish full or abbreviated TLS 1.2 or 1.3 handshakes. Different ciphersuites can be specified for the tests. This is a regular multi-threading program using epoll for network I/O and OpenSSL for all the cryptographic logic.

We used wrk [38] to evaluate performance of full HTTPS transactions, involving full or abbreviated TLS handshake, one HTTP request, and one response.

SUT configurations

We used 2 environments for the system under test. The first one is a 1 vCPU KVM virtual machine running on a host system with Intel Core i7-6500U CPU (no vAPIC). virtio_net network driver was used. The virtual machine ran Tempesta FW or Nginx/OpenSSL, while the benchmarking tools were ran from the host system.

The second SUT environment is 2 bare-metal servers with Intel Xeon CPU E3-1240v5 CPUs and Mellanox ConnectX-2 Ethernet 10Gbps network adapters. The average round trip time between the servers is 0.06ms.

We used following *sysctl* settings in all the performance tests to avoid TCP connections hash table pollution with *TIME-WAIT* connections:

```
net.ipv4.tcp_max_tw_buckets=32
net.ipv4.tcp_max_orphans=32
net.ipv4.tcp_tw_reuse=1
net.ipv4.tcp_fin_timeout=1
```

Plus to these settings, Tempesta FW also sets

```
net.core.netdev_max_backlog=10000
net.core.somaxconn=131072
net.ipv4.tcp_max_syn_backlog=131072
```

Since load distribution in TLS handshake is not equal and the client needs more computations than the server, we had to disable some server CPU cores to stress the server. We limited CPUs only to first physical core (2 hyperthreads). We switched off the rest of the CPUs to avoid side effects onto the performance results.

```
cd /sys/devices/system/cpu/
for cpu in cpu[2-7]*/online; do
  echo 0 >"$cpu"
done
```

The fork of mbed TLS

Originally, Tempesta TLS was forked from GPLv2 version of mbed TLS 2.8.0 [18, 17]. However, it was significantly reworked to make the code fast and at the moment only PKI code is left from the original mbed TLS code. The main changes are:

- Zero-copy I/O [17];
- Awareness of the current TCP congestion and send windows [17];
- Using Linux native crypto API for symmetric ciphers and hashes [17];
- Memory allocations and MPI overheads were removed from the hot paths.

The first three points were addressed in our previous work [17].

While not all possible performance optimizations are finished yet, we have improved performance of the original mbed TLS code for 40 times. The table at the below shows tls-perf [33] average results for the original port of mbed TLS [18] and the current Tempesta TLS code. The both implementations were run in a single CPU VM.

Implementation	handshakes/s	latency (ms)
mbed TLS	51	2'566
Tempesta TLS	2'180	307

Cryptographic routines

The previous sections showed that while the raw performance of the Tempesta TLS cryptographic routines is still lower than for OpneSSL, Tempesta FW can establish more TLS connections per second than Nginx with OpenSSL. The reasons for the better performance of the whole TLS handshake are:

- no memory allocations in run time;
- no user/kernel spaces context switches;
- more efficient network I/O without copies and extra queueing [17];
- zero-copy TLS handshake state machine [17].

This section describes the performance improvements in the Tempesta TLS cryptography and discusses the further work directions.

The big integers overhead

After the mbed TLS code integration with the Linux TCP/IP stack and the main Tempesta FW module [17], we came to the following profile for TLS handshakes:

```
12.12%  memset_erms
10.69%  ecp_mod_p256
 7.10%  __kmalloc
 5.58%  kfree
 5.08%  memcpy_erms
 4.99%  mpi_mul_hlp
 4.48%  ttls_mpi_copy
 4.44%  ttls_mpi_cmp_abs
 3.91%  ttls_mpi_sub_abs
 3.68%  ttls_mpi_cmp_mpi
 3.25%  mpi_sub_hlp
 2.71%  ttls_mpi_free
 2.62%  ttls_mpi_shift_r
 2.55%  ttls_mpi_mul_mpi
```

```
2.52%  ___cache_free
1.81%  ttls_mpi_bitlen
1.69%  ttls_mpi_grow.part.0
1.21%  ttls_mpi_shift_l
0.88%  ttls_mpi_add_abs
0.88%  ecp_modp
0.83%  ttls_mpi_lset
```

The memset_erms() function zeroes the security sensitive data. This call was easily optimized by using AVX2 version of memset() function [17] and calling it only once at the end of a full cryptographic operation.

The functions with prefix ttls_mpi_ are big integer wrappers, inherited from mbed TLS. For the NIST P-256 elliptic curve and a 64-bit machine a big integer is an array of 4 long integers (in other words, 4 *limbs*). In a simple case, a big integer wrapper can be defined as

```
typedef struct {
    short           sign;
    unsigned short  limbs;
    unsigned short  used;
    unsigned long   *data;
} BigInteger;
```

The wrapper handles the sign of the big integer, the total number of allocated limbs, number of currently used limbs, and a pointer to the actual data storing the big integer.

However, since mbed TLS uses the same big integer abstraction layer for other algorithms using much bigger integers, the layer employs a lot of conditional statements and, more importantly, implicit memory allocations. For example, if two 4-limb big integers X and Y are multiplied and the result is stored in X, then X can be reallocated by the multiplication function to store the 8-limb result. Also in this example the multiplication is performed in two nested loops, while exact number of iterations for each loop is known on compile time and could be efficiently unrolled.

OpenSSL [28] and WolfSSL [36] libraries use specialized implementations of all elliptic curves routines. Just moving from the generic big integers implementation to the specialized routines improved ECDSA signing time for 12% and ECDHE key pair and the shared secret generation times for 19% in total.

The calls __kmalloc(), kfree(), and __cache_free() are responsible for implicit dynamic memory allocations on the big integers layer. Mbed TLS allocates temporary big integers also from the heap, which requires memory zeroing on each freeing. This prevents exposing of a security sensitive data if the system allocator returns the freed memory chunk in a subsequent allocation as is.

Tempesta TLS allocates all temporary big integers on stack as plain arrays of 64-bit integers, without big integer wrappers. All long-living big integers are allocated at once at begin of the TLS handshake process. There are no memory allocations among the top 30 hottest functions of the current Tempesta TLS profile.

Side channel attacks restrictions

As [29] we also focus on server side software which is physically not accessible to an attacker, so resistance of the cryptographic computations against power analysis attacks is out of the scope of our work.

Mbed TLS [26] randomizes projective coordinates [5], employs odd-only comb method [7] and constant time computations, not everywhere though. Non-constant time algorithms, e.g. modular inversion, are coupled with randomization. Neither OpenSSL [28] nor WolfSSL [36] use randomization, instead the libraries use constant time computations in all security crucial places [10]. (WolfSSL team has fixed non-constant time elliptic curve fixed point multiplication after our report [37].)

At the moment Tempesta TLS uses the same mix of constant time and non-constant time algorithms with randomization as mbed TLS. However, we consider to replace the constant time comb method for point multiplications with the recent non-constant time algorithms (e.g. [6] or [29]) coupled with randomization to achieve better performance.

Randomization

Constant time algorithms are expensive. For example, the elliptic curve fixed point multiplication in OpenSSL and WolfSSL must scan the 150KB precomputed table for 36 times [10].

RDRAND CPU instruction provides very fast random number generator, so in our case we can replace the full table scans with a point randomization, which implies much lower overhead: a random number generation, 1 squaring, and 1 multiplication. Moreover, with the approach we can use a much larger precomputed table to get even better performance.

Unfortunately, the mitigation of the recent SRBDS vulnerability [12] leaves only 3% of the original performance of the RDRAND instruction [23]. Hopefully, the newest Ice Lake Intel CPUs family fixes the SRBDS vulnerability [13].

Since Tempesta TLS extensively uses randomization, the Tempesta FW system prerequisites require the newest CPUs not affected by SRBDS. These requirements significantly limits the system applicability, so we plan introduce to two versions of the code: one extensively using randomization for newer CPUs and the second mbed TLS-like approach mixing randomization with constant time algorithms for legacy hardware. A new Linux kernel configuration variable should control which version of the code is compiled.

Point multiplication

The original mbed TLS code uses the Lim and Lee [25] method to multiply a scalar by a fixed and unknown elliptic curve points. The only difference between multiplication for fixed and unknown points is in the size of the precomputed tables.

We wrote a designated function for the fixed point multiplication. The precomputed table is built on compile time. We use the comb window size 7, which for 256-bits NIST P-256 curve leads to $\lceil 256/7 \rceil = 37$ iteration in the main loop. To avoid point doublings in the main loop completely we precompute 37 tables. This leads us to the very similar algorithm and the precomputed table as used by OpenSSL [10] and WolfSSL [36].

The whole size of the all precomputed tables is about 152KB, which significantly larger than L1d cache size on the modern CPUs. Since a secret value is used to access the tables in all the ECDSA and ECDHE computations, cache misses caused by direct accessing the tables may lead to revealing bits of the secret value. To prevent the side channel attack, OpenSSL [10] scans the table on each iteration of the point multiplication loop, 36 times in total. WolfSSL also recently replaced direct access of the table with full scan [37]. The WolfSSL developers reported 5-17% performance degradation introduced by the change.

Instead of scanning the whole precomputed tables, we multiply the secret value by a random number before the point multiplication loop and use direct access to the tables. Since the scalar is randomized, the different access time doesn't expose bits of the secret value. We use RDRAND (with prerequisites of modern CPUs not affected by SRBDS) to quickly generate random numbers for the multiplication.

It seems the Lim and Lee [25] algorithm with multiple precomputed tables is still one of the most efficient algorithms for fixed point multiplication. The algorithm was used with maximum number of precomputed tables, such that only point additions are required and there are no point doubling in the main loop. We use mixed point addition formula with $8M + 3S = 10.4M$ complexity (expressed in required number of scalar multiplications M and squarings S having that squaring takes 0.8 of time of multiplication as in [29]), i.e. the whole point multiplication costs $10.4M * 36 = 374.4M$ using 152KB of storage.

We compared the Lim and Lee multi-table method with the recent research by Robert et al [29], which proposes the novel R-prime algorithm. It's not stated explicitly in their paper, but we believe the single table variation of the Lim and Lee algorithm was used in all their comparisons. We evaluated the required memory size and computation cost (in number of required multiplications) for several Lim and Lee window values and maximum number of tables:

w size	multiplications	storage (KB)
6	452	87
8	342	263
10	275	848
11	251	1548
13	214	5278
15	187	18432

We compared the numbers with the results for R-prime (table XI for NIST curve P-256 [29]) using the Figure 1.

The figure shows that the smallest table sizes, including our 152KB table, and the largest table sizes are more efficient for R-prime, but 1548KB table size is bit more efficient for the multi-table comb. However, it's more efficient to use Lim and Lee method with bigger window and smaller number of tables in case of memory constrains. In the Robert et al [29] comparisons of R-prime with single-table Lim and Lee, Lim and Lee is more efficient on small storage sizes. Thus, R-prime should be used only with the table sizes larger than 1548KB, all lower table sizes are better handled by multi-table Lim and Lee. Both the algorithms work in constant time, but Robert et al [29] also introduce the non-constant time m0m1 algorithm, which is faster than R-prime.

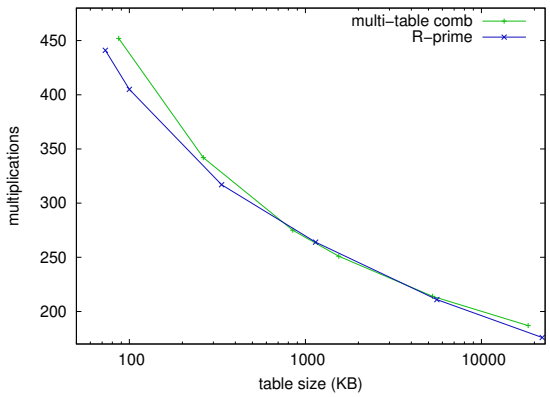

Figure 1: Comparison of R-prime and multi-table comb methods

Unknown point multiplication must pre-compute all values in the run time, so the tables must be small. Thus, Lim and Lee remain the best choice for unknown point multiplication.

From the other side, the fixed point multiplication, having fast random generator, can benefit from large precomputed tables. Using constant-time algorithms with full table scan makes sense only for architectures having no fast or secure random number generator. We're going to evaluate m0m1 and one more non-constant time point multiplication algorithm proposed by Mohamed et al [6], with different sizes of precomputed tables.

Modular inversion

Tempesta TLS, mbed TLS, OpenSSL, and WolfSSL use Jacobian coordinates, so a modular inversion is used to convert coordinates from Affine to Jacobian and back. The modular inversion is the second most hot spot after the point multiplication [10]. OpenSSL and WolfSSL use SIMD implementations of the algorithm based on the Fermat's little theorem, i.e. using 255 modular squarings and 13 modular multiplications [10]. While the Fermat's little theorem method is computationally heavy, it works in constant time.

The original mbed TLS code for the modular inversion uses non-constant time algorithm, so the implementation must use multiplication by a random number to blind a secret value.

Bernstein and Yang recently proposed a constant time modular inversion algorithm [2], significantly faster than the Fermat's method. As Tempesta TLS inherits the randomization for the secret value, we can use non-constant time algorithm for modular inversion. The original Bernstein and Yang algorithm always executes 741 division steps to inverse a 256-bit integer. In our version the algorithm executes 560 iterations in average and also saves several big integer operations at the end of the algorithm. With moving to the new modular inversion algorithm we improved performance of ECDSA for 56% and ECDHE for 18%. The modular inversion algorithm currently still uses big integers and can be optimized further by moving to specialized assembly routines.

Modular reduction

At the moment modular multiplication and squaring dominate in all the computations (18% and 11% of the whole TLS handshake computations correspondingly). Both the operations are merged with modular reduction using FIPS 186-4 D.2 algorithm. We inherited the FIPS modular reduction method from the original mbed TLS implementation. Several optimization strategies were applied to the algorithm, including the final table-based subtractions and additions proposed by Bos [3]. After all the optimization the most performance profiling samples hit the modular reduction part of the multiplication and squaring functions. Moreover, the ratio between the execution time of both the functions is 0.9, which is significantly more than usual $[0.8, 0.86]$ ratio for the implementations using Montgomery reduction. OpenSSL and WolfSSL use Montgomery reduction and we believe that moving to Montgomery reduction will significantly improve performance of modular arithmetics.

TLS handshakes for the Linux kernel

We propose the kernel TLS handshakes for the inclusion into the main line of the Linux kernel. Current Tempesta TLS implementation takes into account only requirements for application delivery controllers, but the generic implementation must take care about wider range of use cases. The state of the proposal and the implementation discussions can be found in the corresponding task on GitHub [21].

The Linux kTLS [20] implements only TLS data transfer part using symmetric cryptography. TLS handshakes were not considered to be a part of kernel and are processed in the user space. A user-space application must open a TCP connection, complete a TLS handshake over it, and at the end the socket can be configured for kTLS by addition of a session key via setsockopt() system call. When these prerequisites are done, the application can use send()/receive()/sendfile() to send and receive data buffers or use sendmsg()/recvmsg() to send TLS control messages [20].

With this work flow a server application runs the following set of operations:

1. A TCP handshake is processed in SoftIrq context for a new incoming connection;
2. A new socket descriptor is passed into user-space application and a TLS handshake is processed by the application;
3. A setsockopt() system call is used to configure a session key for the socket.
4. send()/receive() operations over the TLS socket are used as for usual TCP socket, but the kernel transparently encrypts and decrypts all passed data.

Limitations

There is no intention to move the whole TLS handshake implementation into the kernel. For example, the whole OpenSSL's code base is big enough and many outdated features are supported there. Instead we propose to move to the kernel only the most beneficial features for now: TLS 1.2 and 1.3 server handshake state machine, including for the abbreviated versions, ECDHE and ECDSA implementations over the NIST P-256 and x2559 elliptic curves. This minimal features set limits the size of TLS handshakes code in kernel, but it also limits acceptable TLS options and supports only modern clients.

To avoid the restrictions and to support all flavours of TLS clients we should keep the user-space handshake processing as a fallback mechanism.

Today Tempesta TLS supports only the modern CPU families of x86-64 architecture to employ AVX, BMI, and ADX extensions for the maximum performance. There is no generic C implementation of some cryptographic routines. However, there are other CPU architectures becoming more and more wide spread in the cloud and enterprise environments, e.g. ARM. There are also many Linux installations with old hardware. The much wider set of CPUs must be supported in the upstream version of the TLS handshakes.

Architecture overview

Unlike current kTLS implementation, working in a process context, TLS handshakes should be done in SoftIRQ context, just like TCP handshakes. A separate context performs TLS handshakes on non-blocking sockets without extra context switches [1] and a user-space application is notified only once, when a handshake completed. To speedup cryptographic computation, SoftIRQ must acquire FPU context on getting CPU time and release it on exit. This way, TLS handshakes can be processed in batches, eliminating extra FPU context switches. The FPU context save/restore operation was not used in SoftIRQ before and may affect performance of applications aggressively using FPU registers, which is the majority of modern applications due to auto-vectorization. Thus the operations must be enabled only when the kernel is configured with TLS handshakes.

Heavy cryptographic computations in SoftIRQ may introduce large latencies [24], so designated per-cpu kernel threads can be a good alternative. Meantime, the work queue mechanism has too high overheads to be used for TLS handshakes.

The workflow for an application with kernel TLS handshake processing will be changed to:

1. TCP handshake is processed in SoftIRQ context for a new incoming connection;
2. TLS handshake is started in SoftIRQ for the same connection
   (a) if it succeeds, a new socket descriptor with an established TLS session is returned to the user-space application as ready to read or write.
   (b) otherwise (fallback scenario), if the kernel can not complete the handshake, a new socket descriptor with TCP connection only established is returned to the user space and a TLS handshake is fully processed by the application.
3. The setsockopt() system call configures a session key for the socket to enable kTLS.
4. send()/receive() operations over the TLS socket are used as for current TCP sockets with enabled kTLS.

The usual requirements for TLS servers also include secret keys and certificates management and possibility to run different virtual servers over the same listen address. We propose to use Kernel Key Retention Service [15] to store and manage keys for every virtual server. The service already has access management and SELinux support, so it's possible to use a privileged process for sensitive data management while worker process won't be able to access neither the private key nor the session key during servicing remote user connection. But such role management between processes is not enforced and developers are free to use single process for the both roles.

Virtual servers on the same listening address are differentiated via Server Name Identifier (SNI) extension, while the same listening address might be requested by multiple processes. SNI of a virtual server in our proposal is a root node in a keyring [15]. Since it allows access rights management, different processes and containers can be able to update/revoke their certificates and keys, but can't have access to certificates and keys of other processes and containers working on the same host.

Socket API

Before a TLS socket is created, it's required to load a certificate and a private key. Kernel Key Retention Service provides add_key(2) system call with asymmetric-key(7) API which allows to load X509 certificates, including ECDSA keys. Since the functions request_key() and request_key_rcu(), used for the key search by asymmetric keys system, can be called in a process context only, the interface can't be used to find a key during the handshake process. Thus, we should use the setsockopt(2) system call to extract the keys in the process context and place them into an alternate storage associated with the listening socket. The storage must provide non-sleepable and fast API to be used in SoftIRQ context. The storage must be keyed by SNI string.

The TLS implementation must register 2 new key_types: tls_cert for TLS certificates and tls_priv for private keys. A keyring defines the pair of a certificate and corresponding private key for a particular SNI.

Following code can be used to create a new keyring for SNI example.com:

```
key_serial_t my_sni_kr =
  add_key("keyring", "example.com",
          NULL, 0, 0);
```

Loading the certificate and the private key for the SNI can be done with:

```
key_serial_t cert =
  add_key("tls_cert", "tlscrt:",
          cert_data, cert_len,
          my_sni_kr);
key_serial_t priv_key =
  add_key("tls_priv", "tlspk:",
          priv_key_data, pk_len,
          my_sni_kr);
```

After the keyring is populated a normal TCP socket is created and bound to a listening port:

```
int sd = socket(AF_INET6, SOCK_STREAM,
                0);
bind(sd, ...);
```

Next, a TLS context is created for the keyring my_sni_kr and setsockopt(2) is called to setup the TLS context for the listening socket:

```
struct tls12_crypto_info_aes_gcm_128 ci
= {
  .versions =
      TLS_1_2_VERSION | TLS_1_3_VERSION,
  .tls_12_cipher_suites =
      ECDHE_ECDSA_AES128_GCM_SHA256
      | ECDHE_ECDSA_AES256_GCM_SHA384,
  .tls_13_ciphersuites =
      TLS_AES_128_GCM_SHA256
      | TLS_AES_256_GCM_SHA384,
  .ecurves = secp256r1 | x25519,
  .tls_keyring = my_sni_kr,
};
setsockopt(sd, SOL_TCP, TCP_ULP, "tls",
           sizeof("tls"));
setsockopt(sd, SOL_TLS, TLS_HS, &ci,
           sizeof(ci));
```

The TLS handshake happens on a listening socket. Polling system calls, e.g. epoll(2), poll(2), and select(2), must return a TLS socket as ready when TLS handshake is completed. If the kernel implementation lacks some necessary logic to finish the handshake, but there is no error on the TLS layer, then accept(2) returns a descriptor for a new TCP connection socket. The user space application can pass the socket to a TLS library, which handles the TLS handshake. We introduce the new protocol family PF_TLS to check the status of the socket from accept(2) system call without extra getsockopt(2) calls.

```
listen(sd, ...);
struct sockaddr_in sa;
int new_sd = accept(listen_fd,
                    (struct sockaddr *)&sa,
                    sizeof(sa));
if (sa.sin_family == PF_TLS) {
  /* Ready to read()/write() */
} else {
  /*
   * Fallback to user-space handshake.
   */
}
```

If the kernel fails to establish the TLS handshake, then the user space application must be able to read ClientHello message from the socket descriptor, so that a regular TLS library, without any changes, can perform a TLS handshake on its own. It's guaranteed that the first read operation won't block and will contain the full ClientHello message. This also implies that the kernel TLS handshake mechanism can not fallback to the user space on later handshake phases.

Now, the socket is ready to read and send data with decryption and encryption on the kernel layer.

The server can close the socket using normal close(2) or shutdown(2) system calls. With SHUT_RD shutdown(2)

mode, any read operation from the user space ends with an error, but the socket is still in reading mode to read TLS alerts from the peer and properly close the TLS session.

Normally, multiple virtual servers may listen on the same address, and target virtual host is chosen by SNI value. A simple function can be used to match SNIs listed in the keyring one-by-one until a match is found.

Linux kTLS performance issues

The recent work [31] reveals several performance issues with the current kTLS [34] implementation, which are also applicable to Tempesta TLS since both the implementations use the same Linux cryto API. In particular, the work discusses a possibility to use precomputations for the Karatsuba algorithm. Also the authors mentioned the memory issues with AES-GCM.

We studied the problem [32] with extra memory allocations and copyings in the gcmaes_encrypt() Linux crypto API function. Following perf profile was collected for transmission of a 19KB web page using kTLS:

```
11.70%  _encrypt_by_8_new8
 5.21%  scatterwalk_copychunks
 2.38%  skb_release_data
 1.54%  get_page_from_freelist
 1.47%  free_hot_cold_page
 1.46%  __alloc_skb
 1.45%  __kmalloc
 1.31%  tfw_tls_encrypt
 1.26%  aesni_gcm_precomp_avx_gen2
```

The scatterwalk_copychunks() and __kmalloc() calls can be optimized out.

Linux kTLS works in a process context, so a TLS record encryption and transmission happen in different contexts and points in time. The TLS records formed in this way may exceed the size of currently allowed TCP transmission, leading to extra delays on TLS decryption on a peer [9].

Tempesta TLS encrypts [17] data in sk_write_xmit() callback called by the Linux TCP/IP stack right on transmission time, when we know precisely how much data we can send. KTLS may benefit from the same approach, removing unnecessary delays, especially at the beginning of a TLS connection.

Availability

The Tempesta TLS is published under GPLv2 and is available as the part of Tempesta FW source code at `https://github.com/tempesta-tech/tempesta`.

References

[1] Benjamin, D. 2020. TLS 1.3 and TCP interactions. In *IETF TLS mailing list*. `https://www.mail-archive.com/tls@ietf.org/msg12569.html`.

[2] Bernstein, D. J., and Yang, B.-Y. 2019. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 340–398.

[3] Bos, J. W. 2010. High-performance modular multiplication on the cell processor. In *Arithmetic of Finite Fields*, 7–24. Springer Berlin Heidelberg.

[4] Baseline Requirements Documents (SSL/TLS Server Certificates), section 6.1.5. `https://cabforum.org/baseline-requirements-documents/`.

[5] Coron, J.-S. 1999. Resistance against differential power analysis for elliptic curve cryptosystems. In *CHES*.

[6] Farah, N.; Hashim, M.; and Hutter, M. 2012. Improved fixed-base comb method for fast scalar multiplication. 342–359.

[7] Feng, M.; B.Zhu, B.; Xu, M.; and Li, S. 2005. Efficient comb elliptic curve multiplication methods resistant to power analysis. In *IACR Eprint archive*.

[8] Graham-Cumming, J. 2017. Incident report on memory leak caused by Cloudflare parser bug. In *The Cloudflare Blog*. `https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/`.

[9] Grigorik, I. 2013. Optimize TLS Record Size. In *High Performance Browser Networking*. `https://hpbn.co/transport-layer-security-tls/#optimize-tls-record-size`.

[10] Gueron, S., and Krasnov, V. 2014. Fast prime field elliptic-curve cryptography with 256-bit primes. volume 5, 141–151.

[11] IANA. 2020. Transport Layer Security (TLS) Parameters, TLS Supported Groups. `https://www.iana.org/assignments/tls-parameters/tls-parameters.xml#tls-parameters-8`.

[12] Intel Corporation Developer Zone. Deep dive: Special register buffer data sampling. `https://software.intel.com/security-software-guidance/insights/deep-dive-special-register-buffer-data-sampling`.

[13] Intel Corporation Developer Zone. Processors affected: Special register buffer data sampling. `https://software.intel.com/security-software-guidance/insights/processors-affected-special-register-buffer-data-sampling`.

[14] Jain, A. S.; Duyck, A. H.; Sarangam, P.; and Jani, N. 2017. Story of Network Virtualization and its future in Software and Hardware. In *Netdev 2.1*. `https://netdevconf.info/2.1/session.html?jain`.

[15] Kernel key retention service. `https://www.kernel.org/doc/html/latest/security/keys/core.html`.

[16] Page Table Isolation (PTI). `https://www.kernel.org/doc/html/latest/x86/pti.html`.

[17] Krizhanovsky, A., and Koveshnikov, I. 2018. Kernel TLS handshakes for HTTPS DDoS mitigation. In *Netdev 0x12*. `https://netdevconf.info/0x12/`

session.html?kernel-tls-handshakes-for-https-ddos-mitigation.

[18] Krizhanovsky, A. 2017. Kernel HTTP/TCP/IP stack for HTTP DDoS mitigation. In *Netdev 2.1*. https://www.netdevconf.org/2.1/session.html?krizhanovsky.

[19] Benchmark for the Linux kernel SIMD memory routines. In *Open source benchmarks and code samples for the Tempesta Technologies blog*. https://github.com/tempesta-tech/blog/tree/master/kstrings.

[20] Kernel TLS. https://www.kernel.org/doc/html/latest/networking/tls.html.

[21] Linux kernel TLS, Tempesta FW project task. https://github.com/tempesta-tech/tempesta/issues/1433.

[22] Larabel, M. 2020a. Looking at the linux performance two years after spectre / meltdown mitigations. https://www.phoronix.com/scan.php?page=article&item=spectre-meltdown-2&num=10.

[23] Larabel, M. 2020b. RdRand Performance As Bad As 3% Original Speed With CrossTalk/SRBDS Mitigation. https://www.phoronix.com/scan.php?page=news_item&px=RdRand-3-Percent.

[24] Latency can increase for huge amount of TLS connections. https://github.com/tempesta-tech/tempesta/issues/1434.

[25] Lim, C. H., and Lee, P. J. 1994. More flexible exponentiation with precomputation. In *Precomputation, Advances in Cryptology - CRYPTO 94*, 95–107.

[26] The mbed TLS project. https://github.com/ARMmbed/mbedtls.

[27] Meltdown and Spectre. Vulnerabilities in modern computers leak passwords and sensitive data. https://meltdownattack.com/.

[28] OpenSSL TLS/SSL and crypto library. https://github.com/openssl/openssl.

[29] Robert, J.-M.; Negre, C.; and Plantard, T. 2019. Efficient fixed-base exponentiation and scalar multiplication based on a multiplicative splitting exponent recoding. In *Journal of Cryptographic Engineering*.

[30] Schwenke, A. 2018. MyISAM and KPTI Performance Implications From The Meltdown Fix. https://mariadb.org/myisam-table-scan-performance-kpti/.

[31] Szymanski, P., and Deval, M. 2020. TLS performance characterization on modern x86 CPUs. In *Netdev 0x14*. https://netdevconf.info/0x14/session.html?talk-TLS-performance-characterization-on-modern-x86-CPUs.

[32] Tls: further performance improvements and cleanups. https://github.com/tempesta-tech/tempesta/issues/1064.

[33] Tls handshakes benchnarking tool. https://github.com/tempesta-tech/tls-perf.

[34] Watson, D. 2016. Kernel TLS (Transport Layer Security) Socket. In *Netdev 1.2*. http://www.netdevconf.org/1.2/session.html?dave-watson.

[35] TLS 1.3 Performance Part 1 Resumption. https://www.wolfssl.com/tls-1-3-performance-resumption/.

[36] wolfSSL (formerly CyaSSL) implementation of TLS/SSL. https://github.com/wolfSSL/wolfssl/.

[37] WolfSSL library. 2020. SP ECC Cache Resitance, pull request. https://github.com/wolfSSL/wolfssl/pull/3195.

[38] Modern HTTP benchmarking tool. https://github.com/wg/wrk.