RustCrypto AES/GCM and ChaCha20+Poly1305 Implementation Review

MobileCoin

February 13, 2020 – Version 1.0

Prepared for
MobileCoin

Prepared by
Gérald Doussot
Thomas Pornin

Executive Summary

Synopsis

In December 2019, MobileCoin engaged NCC Group to conduct a review of the AES/GCM and ChaCha20+Poly1305 implementations provided by the RustCrypto/AEADs crates. The intended usage context of these crates includes SGX enclaves, making timing-related side channel attacks relevant to this assessment. Two consultants provided five person-days of efforts.

Scope

The main target of evaluation consisted of the `aes-gcm` and `chacha20poly1305` crates that are part of the RustCrypto/AEADs repository on GitHub. These crates use other crates from other RustCrypto repositories. In all cases, the source code reviewed was the latest version available at the time of the engagement:

- `aes-gcm` and `chacha20poly1305` from RustCrypto/AEADs:
  https://github.com/RustCrypto/AEADs/tree/a15698fdba23ffb17b84d9ecaa2c9c80706ecf03
- `aes` from RustCrypto/block-ciphers:
  https://github.com/RustCrypto/block-ciphers/tree/e385f1ebb2ec48547194e51c5193309ee328d93b
- `chacha20` and `salsa20-core` from RustCrypto/stream-ciphers:
  https://github.com/RustCrypto/stream-ciphers/tree/1235638004c21dee4e76af4cc932cf1cd815e8f9
- `aead`, `stream-cipher` and `universal-hash` from RustCrypto/traits:
  https://github.com/RustCrypto/traits/tree/4569d256f02ac0ecefa393baf225fb4a6df35875
- `ghash`, `poly1305` and `polyval` from RustCrypto/universal-hashes:
  https://github.com/RustCrypto/universal-hashes/tree/1ab06bd79542e75490468b227dd3c2cbe42d3d92

Key Findings

NCC Group did not find any vulnerability in the audited crates. The RustCrypto implementations use all the recommended techniques to achieve constant-time implementations; in particular, the fallback AES implementation (to be used when there are no usable hardware AES opcodes) uses bitslicing to avoid any table lookups at secret-dependent addresses. Similarly, received authentication tags are compared with constant-time comparison functions.

A few cosmetic remarks, mostly related to potential performance improvements, have been assembled into RustCrypto Audit Notes on the following page.

In this section, NCC Group provides a description of the audit process on RustCrypto, and a few remarks. None of these observations constitute a vulnerability, even in theoretical contexts.

Crate Structure

The scope of the audit was the AES/GCM and ChaCha20+Poly1305 implementations from the RustCrypto AEADs repository:

- AES/GCM: https://github.com/RustCrypto/AEADs/tree/master/aes-gcm
- ChaCha20+Poly1305: https://github.com/RustCrypto/AEADs/tree/master/chacha20poly1305

These two crates themselves rely on features provided by other RustCrypto crates:

- `aes`, `aes-soft` and `aesni` from `block-ciphers`: https://github.com/RustCrypto/block-ciphers/tree/master/aes
- `chacha20` and `salsa20-core` from `stream-ciphers`: https://github.com/RustCrypto/stream-ciphers
- `ghash`, `poly1305` and `polyval` from `universal-hashes`: https://github.com/RustCrypto/universal-hashes
- `aead`, `block-cipher-trait`, `crypto-mac`, `stream-cipher` and `universal-hash` from `traits`: https://github.com/RustCrypto/traits

In general, implementations are structured in layers that allow sharing of common parts (e.g. `salsa20-core` provides the support for data buffering that is used in all ChaCha20 and Salsa20 variants).

AES Implementations

The `aes` crate is a wrapper for an underlying implementation, which will be `aesni` (if the target platform is 32-bit or 64-bit x86 with the AES-NI opcodes), or `aes-soft` (for all other platforms).

aesni

The `aesni` implementation uses the dedicated hardware opcodes (through the special functions from `arch`, such as `_mm_aesenc_si128()`). It is rather straightforward, but evaluates up to eight instances in parallel (with the same key) by interleaving the opcodes: this is meant to provide better performance, since the CPU opcodes typically have high throughput but nonnegligible latency (1 `aesenc` per cycle on a Skylake core, but with a latency of 4 cycles).

Noteworthy is the presence of a CTR implementation (in `aesni/src/ctr.rs`). This implementation appears to follow the "64-bit" variant; indeed, CTR can be defined in several ways, depending on how many bits in the AES block are used for the block counter. AES/GCM uses 32-bit counters; other standard AEAD modes (e.g. CCM and EAX) use full-width 128-bit counters. What this CTR implementation with 64-bit counters is meant to be used for is unclear. The GCM implementation has its own CTR code (with 32-bit counters) and does not use the code defined in `aesni/src/ctr.rs`. Also, the non-AES-NI implementation does not feature any counterpart to this code.

aes-soft

The `aes-soft` implementation provides a constant-time software AES implementation. The implementation technique is bitslicing: individual state bits are spread onto many registers, all operations being performed with bitwise operators. Full bitslicing would involve using 128 registers for the 128 bits of an AES block; this would naturally lead to n parallel evaluations of AES, if registers have size n bits. Since most CPUs do not have 128 distinct registers, this would result in high pressure on the register allocator and waste of CPU resources for exchanges of data between CPU registers and stack slots. Instead, the usual implementation structure for bitsliced AES is to keep 16 bits of state in a single register: the 16 S-Boxes of each AES round are still evaluated in parallel with the same sequence of code, but the other AES operations (MixColumns, ShiftRows...) require slightly more data movement operations. The famous Käsper-Schwabe AES implementation[1] uses this structure. Since each register can store 16 bits from a given AES instance, only eight registers are needed, and $n/16$ AES instances are evaluated in parallel.

Interestingly, the `aes-soft` implementation works over 32-bit values (`u32`) but defines a custom 128-bit type:

[1] https://eprint.iacr.org/2009/129

```
pub struct u32x4(pub u32, pub u32, pub u32, pub u32);
```

It is unclear why it is done that way. If the actual registers are 32-bit values, there should be no additional benefit to increasing parallel evaluation beyond what can fit in a register: an XOR of two 32-bit registers is one operation, but the XOR of two 128-bit custom types will involve four individual 32-bit XOR operations. The use of `u32x4` is then equivalent to using a 32-bit bitsliced AES implementation (with two instances in parallel), but running four such implementations with their code interleaved. The number of required registers then rises to 32, which will increase pressure on the register allocator.

Therefore, while the current `aes-soft` implementation is correct and secure, a similar implementation with only 32-bit values and limited parallelism (two AES instances in parallel) would yield simpler source code, smaller binaries, and probably better performance.

GHASH

GHASH is the universal hash function used for message authentication in GCM. The implementation uses POLYVAL (from RFC 8452, section 3[2]), which is identical to GHASH except for encoding endianness. The core operation is a multiplication in the binary field $GF(2^{128})$. When the AES-NI opcodes are available, the `pclmulqdq` opcode is used, leading to reasonably good performance, though not completely on par with AES-CTR. GHASH (POLYVAL) could be made faster by processing blocks in parallel; indeed, for an input consisting of n blocks m_1 to m_n, and secret key h, the computation is:

$$\text{POLYVAL}(m) = m_1 h^n + m_2 h^{n-1} + m_3 h^{n-2} + \cdots + m_{n-1} h^2 + m_n h$$

Thus, if h^2, h^3 and h^4 are pre-computed, then it becomes possible to process four blocks in parallel, i.e. such that the output of one multiplication is not used as input for the next three blocks. This promotes performance in two ways:

- Like AES-NI opcodes, `pclmulqdq` has high throughput (1 instruction per cycle on Skylake) but high latency (7 cycles on Skylake). Parallelism reduces the effect of that latency.
- `pclmulqdq` computes the multiplication on binary polynomials (in $GF(2)[X]$) but an extra reduction modulo the irreducible polynomial that defines the finite field $GF(2^{128})$ is also needed. That reduction consists of some shifts and XORs, and has nonnegligible cost. If four blocks are processed in parallel, then the reduction can be mutualized: only one reduction every four blocks will be needed.

Such performance improvements are not required for security, but can help in high bandwidth contexts (e.g. being able to process 5 GB/s instead of 1 GB/s).

Still on performance, it appears that the `universal-hashes/polyval/src/field/pclmulqdq.rs` file defines a function called `shufpd1()`:

```
unsafe fn shufpd1(a: __m128i) -> __m128i {
    let a = _mm_castsi128_pd(a);
    _mm_castpd_si128(_mm_shuffle_pd(a, a, 1))
}
```

This function swaps the lower and upper 64-bit halves of its 128-bit input. It does so by way of the `shufpd` opcode. That opcode was designed for operating with floating-point values. The SSE2 instructions are defined to work with a specific number of bits, but some instructions interpret these bits as integer values, and others use the bits as IEEE-754 floating-point values. Within the CPU, separate units are used for the two interpretations; thus, use of a floating-point opcode like `shufpd` on registers that have just been filled by integer opcodes like `pclmulqdq` will induce, in the hardware, extra latency for the transfer of data from one unit to the other (exact slowdown depends on the CPU

[2] https://tools.ietf.org/html/rfc8452

core and version, but 3 cycles would be typical). In that sense, the operation implemented by `shufpd1()` would be better provided by `_mm_shuffle_epi32(a, 0x4E)`, i.e. the `pshufd` opcode, which is of the "integer" kind.

Amazingly, this would make no difference here for performance, because the replacement of `shufpd` by `pshufd` is already done automatically by the compiler. This is an LLVM feature: the code generator notices that the input to `shufpd` comes from the integer side, and the output goes back to integers too, and thus replacing `shufpd` with an equivalent `pshufd` can only improve things. The same optimization can be observed with the equivalent C code, when compiled by Clang (which is based on LLVM); however, GCC does not perform that optimization.

When `pclmulqdq` is not available, the implementation of POLYVAL uses plain integers over 32 or 64 bits. The multiplication of binary polynomials is performed with plain integer multiplications, with most bits masked out so that carries can be removed (multiplication of binary polynomials can be described as "carry-less multiplication"). This trick, and indeed the code itself, were imported from BearSSL.[3] This computes the correct values in all cases, and is constant-time as long as the underlying multiplication opcodes are constant-time, which is true of all modern "large" CPUs, although not necessarily of older CPUs, or of small microcontrollers.[4]

GCM

The `aes-gcm` crate aggregates the AES and GHASH implementations into a full AES/GCM implementation. The main potential security issue with GCM implementations would be the use of a non-constant-time comparison for verifying the authentication tag when decrypting; however, this operation is indeed implemented by `aes-gcm` with the `ct_eq()` function from the third-party `subtle` crate,[5] which performs the comparison in a fully constant-time way.

A few extra remarks on `aes-gcm`:

- The implementation assumes that the full message and additional data are available, each as a single slice. Streamed processing is not supported. This is not normally a strong limitation, since typical GCM use is on short chunks (e.g. TLS records, or Noise message payloads); and, indeed, when decrypting, the received data should not be used until it has been fully verified, making some sort of buffering unavoidable.
- Only 12-byte nonces are supported. The AES/GCM specification[6] nominally supports all nonce sizes from 1 to $2^{64}-1$ bits. Nonces of exactly 12 bytes (96 bits) are the recommended size, especially when nonce values are generated with a mechanism that guarantees against reuse, as is the case in Noise (a message counter is used). In other contexts, where nonces are generated randomly and non-reuse is only probabilistically ensured, 16-byte nonces are slightly better (for a nonce of size other than 12 bytes, GHASH is used to derive both a 12-byte IV for CTR mode, *and* the starting value of the 32-bit block counter; the latter further decreases the risk of block collisions).
- The implementation enforces maximum sizes for plaintexts, ciphertexts, and additional data (`AEADs/aes-gcm/src/lib.rs`, line 109): plaintexts and additional data must not exceed 2^{36} bytes, and ciphertext must not go beyond $2^{36}+16$ bytes. These sizes are slightly off:
  - The maximum plaintext size, per the GCM specification, is $2^{39}-256$ bits, i.e. $2^{36}-32$ bytes. The implementation thus allows plaintexts to slightly exceed the maximum allowed size.
  - The maximum ciphertext size should indeed be 16 bytes more than the maximum plaintext size, to account for the extra 16-byte authentication tag. However, the code applies the check against its `C_MAX` constant on the ciphertext size without the tag (see the `decrypt_in_place_detached()` function in `AEADs/aes-gcm/src/lib.rs`).
  - As per the GCM specification, additional data length is allowed to be up to $2^{64}-1$ bits, i.e. $2^{61}-1$ bytes; but the implementation prohibits using more than 2^{36} bytes of additional data.

The limits on plaintext and ciphertext sizes come from the 32-bit block counter: value `0` is reserved (so as not to collide

[3] https://www.bearssl.org/constanttime.html#ghash-for-gcm
[4] https://www.bearssl.org/ctmul.html
[5] https://crates.io/crates/subtle
[6] https://csrc.nist.gov/publications/detail/sp/800-38d/final

with the input for the definition of the authentication key h), value 1 is used for the extra block that protects the GHASH output, and the counter for payload encryption starts at 2. If the plaintext size exceeds 2^{36} bytes, this may reveal the authentication key to outsiders, provided that they can observe the ciphertext and reliably guess the contents of the last few bytes of the plaintext (if the size is not greater than $2^{36} + 16$ bytes, this attack would furthermore require that the nonce is all-zeros, which, in a Noise context, happens only for the first post-handshake message).

None of these present an actual risk, because the checks are not part of normal input data validation, but only as failsafe detection mechanisms against programming errors. Moreover, 2^{36} bytes is about 69 gigabytes, and such oversized data is unlikely to fit in the available memory; if the implementation tried, RAM exhaustion would itself be a more worrisome issue. Even on computers with enough RAM, the cost of handling that much data would make the problem apparent.

ChaCha20

The `chacha20` crate provides two ChaCha20 implementations, based on either 32-bit integers (`u32`), or on SSE2 types and opcodes, if available. The implementations are straightforward and correct.

Since ChaCha20 produces a pseudorandom key-dependent stream in chunks of 64 bytes, some buffering is necessary for a streamable API (which the ChaCha20+Poly1305 implementation in `AEADs/chacha20poly1305` will not fully use, since, like the GCM implementation, it requires the whole plaintext/ciphertext to be provided as one slice). The buffering is implemented in `stream-ciphers/salsa20-core/src/lib.rs`. This code seems correct, but slightly more complicated than necessary. NCC Group noted the following:

- On line 210, a value is uselessly written into `self.offset`, since it will be immediately overwritten by another value (on line 228).
- On line 214, the value `self.offset / 4` is reduced modulo `STATE_WORDS` (which is 16). This last modular reduction is not needed since `self.offset` is always between `0` and 63.
- Constants `STATE_BYTES` and `STATE_WORDS` are used to abstract away the block size (64 bytes). However, on lines 142 and 143, the block size is used as the literal integer `64`, not the `STATE_BYTES` constant. Similarly, on lines 238, 242 and 243, the fact that the block size is exactly 64 bytes is hardcoded (shift count of 6 bits, use of a mask of value `0x3f`).

None of the above induces a security vulnerability.

ChaCha20+Poly1305

The `chacha20poly1305` crate assembles the ChaCha20 and Poly1305 implementations. The Poly1305 implementation uses the classic representation of integers modulo $p = 2^{130} - 5$ as five limbs of 26 bits each, stored in 32-bit variables; the extra bits allow for delayed carry propagation. The representation is slightly redundant (several valid encodings exist for the same value) and requires a conditional subtraction of the modulus at the end of the computation; this last step is done in the usual, constant-time way: subtraction of p, extension of the resulting sign bit as a mask, addition of masked p (i.e. addition of zero if the result was not negative; addition of p otherwise).

Upon decryption, as in the case of the GCM implementation, the authentication tag as received is compared with the recomputed authentication tag with a constant-time comparison from the `subtle` crate. In that case, the relevant function is part of the generic functions provided in the `traits/universal-hash` crate (the tag is wrapped as an `Output<16>` instance, whose `PartialEq` implementation carefully calls `ct_eq()`).