ChaCha20-Poly1305 Authenticated Encryption for High-Speed Embedded IoT Applications

Fabrizio De Santis*, Andreas Schauer*, and Georg Sigl*†
*Lehrstuhl für Sicherheit in der Informationstechnik
Technische Universität München (TUM), Munich, Germany
{desantis,a.schauer,sigl}@tum.de
†Fraunhofer Institute for Applied and Integrated Security (AISEC), Munich, Germany
georg.sigl@aisec.fraunhofer.de

***Abstract*—The ChaCha20 stream cipher and the Poly1305 authenticator are cryptographic algorithms designed by Daniel J. Bernstein with the aim of ensuring high-security margins, while achieving high performance on a broad range of software platforms. In response to the concerns raised about the reliability of the existing IETF/TLS cipher suite, its performance on software platforms, and the ease to realize secure implementations thereof, the IETF has recently published the RFC7905 and RFC7539 to promote the use and standardization of the ChaCha20 stream cipher and Poly1305 authenticator in the TLS protocol. Most interestingly, the RFC7539 specifies how to combine together the ChaCha20 stream cipher and Poly1305 authenticator to construct an Authenticated Encryption with Associated Data (AEAD) scheme to provide confidentiality, integrity, and authenticity of data. In this work, we present compact, constant-time, and fast implementations of the ChaCha20 stream cipher, Poly1305-ChaCha20 authenticator, and ChaCha20-Poly1305 AEAD scheme for ARM Cortex-M4 processors, aimed at evaluating the suitability of such algorithms for high-speed and lightweight IoT applications, e.g. to deploy fast and secure TLS connections between IoT nodes and remote cloud servers, when AES hardware acceleration capabilities are not available.**

I. Introduction

Nowadays, the number of interconnected smart devices, typically referred to as “Internet of Things” (IoT) devices, is massively growing due to almost endless applications, which are quickly getting part of the everyday life, such as wearable technologies, smart buildings, or health monitoring devices. IoT devices are typically based on resource constrained embedded processors, such as the family of ARM Cortex-M processors, and are equipped with sensing and short-range communication capabilities, e.g. Bluetooth Low Energy (BLE) modules. These devices typically collect social and environmental (potentially sensitive) data from their sensors and send them over to a remote server, located in the Internet cloud, through an IoT-Internet gateway. In order to protect the communication between IoT nodes and Internet servers, and ensure the customer’s trust in new emerging IoT technologies, the deployment of strong and efficient cryptographic mechanisms is necessary. Authenticated Encryption (AE) provides confidentiality as well as integrity and authenticity for encrypted data. Authenticated Encryption with Associated Data (AEAD) additionally allows to check the integrity and authenticity of so called Associated Data (AD), which is public data that must be authenticated from the receiver, but does not contain confidential information. In the past few years, the concerns raised about the performance and ease to realize side-channel secure implementations of AES in Galois Counter Mode (AES-GCM) resulted in the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) [1]. Today, the CAESAR competition is still running with the goal of determining a portfolio of authenticated ciphers that offers advantages over AES-GCM in terms of performance, security, and ease of correct and secure implementations [2]. About at the same time, the CFRG started the process of evaluating the suitability of the ChaCha20 stream cipher and Poly1305 authenticator, as alternative ciphers in the Transport Layer Security (TLS) protocol, due to major security issues in the RC4 cipher and major implementation pitfalls in the CBC block mode. The ChaCha20 stream cipher is a high-throughput stream cipher, designed by D. J. Bernstein in 2008, as a refinement of the Salsa20 stream cipher, aimed at improving its security bounds without losing performance on software platforms [3]. The Poly1305 authenticator is a one-time polynomial-evaluation message authentication code (MAC), designed by D. J. Bernstein in 2005, aimed at providing fast authentication mechanisms on software platforms [4]. When ChaCha20 is used to generate the one-time secret keys of Poly1305 from public nonce values, then the resulting authenticator is called Poly1305-ChaCha20. In contrast, when ChaCha20 and Poly1305 are combined together to realize an AEAD scheme, then the construction is named ChaCha20-Poly1305. The evaluation of the CFRG recently resulted in the publication of the RFC7539 [5] and RFC7905 [6]: while the former provides a specification guide for the ChaCha20 stream cipher, Poly1305 authenticator, and ChaCha20-Poly1305 AEAD construction, the latter specifies the use of the ChaCha stream cipher and Poly1305 authenticator in the TLS protocol. Poly1305 and ChaCha20 are currently officially supported ciphers by world leading companies like Google Inc. and global projects like OpenSSH.

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

Organization: Section II provides information about ARM Cortex-M4 processors. Section III, IV, and V describe the ChaCha20 stream cipher, the Poly1305 authenticator and the ChaCha20-Poly1305 authenticated encryption scheme, respectively. Section VI provides a detailed description of our implementation on ARM Cortex-M4 processors. Experimental results are presented and discussed in Section VII. A conclusion is provided in Section VIII.

II. ARM Cortex M4 Processors

ARM Cortex-M4 processors are 32-bit RISC processors based on the ARMv7-M architecture, targeting low cost and energy efficient microcontrollers. They are equipped with 13 general-purpose registers, plus the link register (`lr`), the stack pointer (`sp`), and the program counter (`pc`). The `lr` register can also be used as a general-purpose register, after that its content has been stored in memory. Load instructions take $n+1$ clock cycles, where n is the number of words which can be simultaneously loaded from memory. The clock count can be reduced by smartly scheduling independent operations and take advantage of the three-stage pipeline. The ARMv7-M architecture supports the full set of 32-bit Thumb®-2 instructions, which includes *single-cycle* "multiply" and "multiply-and-accumulate" instructions [7]. In particular:

- `UMULL rLO, rHI,` a_i, b_j multiplies two unsigned 32-bit integers a_i and b_j, and stores the 64-bit result into the registers `rLO` and `rHI`.
- `UMLAL rLO, rHI,` a_i b_j multiplies two unsigned 32-bit integers a_i and b_j, and adds the result to the unsigned 64-bit integer contained into the registers `rLO` and `rHI`.
- `UMAAL rLO, rHI,` a_i, b_j multiplies two unsigned 32-bit integers a_i and b_j, adds the two unsigned 32-bit values `rLO` and `rHI` to the result, and finally stores the unsigned 64-bit result back into the registers `rLO` and `rHI`.

Other relevant arithmetic instructions are `ADD`, `SUB`, `ADC`, and `SBC`, which are used to add, subtract, add with carry and subtract with carry, respectively. If the suffix "`S`" is used, e.g. `ADDS`, `SUBS`, `ADCS`, and `SBCS`, then the condition flags in the status register are also updated. A very helpful feature to save clock cycles on ARM Cortex-M4 processors is the so called "flexible second operand". In those instructions supporting the flexible second operand feature, the second operand is a register that can be shifted for free within the same clock cycle using a shift instruction like the arithmetic shift (`ASR`), logical shifts (`LSL` and `LSR`), rotation (`ROR`) and rotation extended by one bit (`RRX`). Since most common instructions like `ADD` and `EOR` support this feature, shifts can be made free in most cases. Finally, note that ARM Cortex-M4 processors support Thumb-2 technology, i.e. 32-bit and 16-bit instructions can be mixed together without switching between the Thumb state for 16-bit instructions and the ARM state for 32-bit instructions, thus allowing to achieve both high code density and high performance at the same time.

III. ChaCha20 Stream Cipher

The ChaCha family of stream ciphers are high-throughput stream ciphers designed for software platforms. The original specification of the ChaCha family allows for different instantiations of the cipher in order to support various "security versus performance" trade-offs [3], e.g. it allows for different key, nonce, and counter lengths as well as for a different number of rounds. In this section, we provide a description of ChaCha20, as specified by the RFC7539, which represents a conservative instantiation with respect to security. Let (p_i, c_i) denote 64-byte blocks of the plaintext $P = p_1||p_2||...$ and ciphertext $C = c_1||c_2||...$, respectively. Then, the ChaCha20 stream cipher $\mathsf{ChaCha20\text{-}SC} : \{0,1\}^{256} \times \{0,1\}^{96} \times \{0,1\}^{*} \to \{0,1\}^{*}$, $(K, N, P) \mapsto C$ encrypts the plaintext P into a ciphertext C using a 32-byte secret key K and a 12-byte nonce N (cf. Algorithm 1). Internally, it uses the $\mathsf{ChaCha20} : \{0,1\}^{256} \times \{0,1\}^{96} \times \{0,1\}^{32} \to \{0,1\}^{512}$, $(K, N, \gamma) \mapsto k_{\gamma}^{N}$ block function in a way reminiscent of a block cipher's counter mode: the plaintext is encrypted by simply chopping P into 64-byte blocks $(p_i)_{0 \leq i < 2^{32}}$ and XOR-ing them with the 64-byte output blocks k_i^N of the $\mathsf{ChaCha20}$ block function, i.e. $c_i = p_i \oplus k_i^N$. Conversely, the decryption is obtained in a simple backward manner by $p_i = c_i \oplus k_i^N$. Note that the $\mathsf{ChaCha20}$ block function takes an additional a 4-byte counter γ as input, which starts at 0 and it is increased for each encrypted block. The $\mathsf{ChaCha20}$ block function works internally by iterating the quarter round function $\mathsf{QR} : (\{0,1\}^{32})^4 \to (\{0,1\}^{32})^4$ over a 64-byte internal state S. The state S is represented as a 4×4 matrix, where each element represents a 32-bit word, and the element at position (i, j) is denoted by $\mathsf{S}[x]$, with $x = 4i + j$ for $0 \leq i, j \leq 3$. The initial state of S is obtained as follows: the first row is filled with the constants `0x61707865`, `0x3320646e`, `0x79622d32`, and `0x6b206574`, the second and third rows are filled with the 32-byte secret key K in little-endian order, the word element at position 12 is filled with the block counter γ, and the remaining three word elements are filled with the nonce N. The quarter function $(i, j, k, l) = \mathsf{QR}(a, b, c, d)$ acts on the state as follows:

$$\begin{cases} e = a \boxplus b & ; \quad h = (d \oplus e) \lll 16 \\ g = c \boxplus h & ; \quad f = (b \oplus g) \lll 12 \\ i = e \boxplus f & ; \quad l = (h \oplus i) \lll 8 \\ k = g \boxplus l & ; \quad j = (f \oplus k) \lll 7 \end{cases},$$

where $\boxplus$ denotes integer addition modulo 2^{32}, $\oplus$ denotes a XOR operation, and $\lll n$ denotes a rotation of n bits to the left. The ChaCha20 block function consists of 10 "double rounds" which alternately execute a "column round" (4 quarter rounds operating on the columns of S) and a "diagonal round" (4 quarter rounds operating on the diagonals of S), thus resulting in a total of 20 round iterations (or 80 quarter rounds). Note that the invocation of the ChaCha20 block function using a single combination of key and nonce (and increasing the block counter) limits the encryption to 256 GB of data, which is a reasonable amount for most applications. Also note that it must be ensured that the nonce is only used once for the same key.

Algorithm 1 ChaCha20 Stream Cipher:
$C = \mathsf{ChaCha20\text{-}SC}(K, N, P)$.

```
Input:   $K \in \{0,1\}^{256}$, $N \in \{0,1\}^{96}$, $P \in \{0,1\}^*$
Output:  $C \in \{0,1\}^{|P|}$
for $i \leftarrow 0$ to $\lceil |P|/512 \rceil - 1$ do
    /* Init State */
    $\mathsf{S}[0] \leftarrow$ 0x61707865, $\mathsf{S}[1] \leftarrow$ 0x3320646e
    $\mathsf{S}[2] \leftarrow$ 0x79622d32, $\mathsf{S}[3] \leftarrow$ 0x6b206574
    $\mathsf{S}[4..11] \leftarrow K$                               {Set Key}
    $\mathsf{S}[12] \leftarrow i$                                  {Set Counter}
    $\mathsf{S}[13..15] \leftarrow N$                              {Set Nonce}
    $\mathsf{S}' \leftarrow \mathsf{S}$                            {Save Initial State}
    for $n \leftarrow 0$ to 9 do                                   {10 Double Rounds}
        /* Column Round */
        $\mathsf{S}[0,4,\ 8,12] \leftarrow \mathsf{QR}(\mathsf{S}[0],\mathsf{S}[4],\mathsf{S}[\ 8],\mathsf{S}[12])$
        $\mathsf{S}[1,5,\ 9,13] \leftarrow \mathsf{QR}(\mathsf{S}[1],\mathsf{S}[5],\mathsf{S}[\ 9],\mathsf{S}[13])$
        $\mathsf{S}[2,6,10,14] \leftarrow \mathsf{QR}(\mathsf{S}[2],\mathsf{S}[6],\mathsf{S}[10],\mathsf{S}[14])$
        $\mathsf{S}[3,7,11,15] \leftarrow \mathsf{QR}(\mathsf{S}[3],\mathsf{S}[7],\mathsf{S}[11],\mathsf{S}[15])$
        /* Diagonal Round */
        $\mathsf{S}[0,5,10,15] \leftarrow \mathsf{QR}(\mathsf{S}[0],\mathsf{S}[5],\mathsf{S}[10],\mathsf{S}[15])$
        $\mathsf{S}[1,6,11,12] \leftarrow \mathsf{QR}(\mathsf{S}[1],\mathsf{S}[6],\mathsf{S}[11],\mathsf{S}[12])$
        $\mathsf{S}[2,7,\ 8,13] \leftarrow \mathsf{QR}(\mathsf{S}[2],\mathsf{S}[7],\mathsf{S}[\ 8],\mathsf{S}[13])$
        $\mathsf{S}[3,4,\ 9,14] \leftarrow \mathsf{QR}(\mathsf{S}[3],\mathsf{S}[4],\mathsf{S}[\ 9],\mathsf{S}[14])$
    end for
    $k_i^N \leftarrow \mathsf{S} \boxplus \mathsf{S}'$              {$\forall 0 \le x \le 15 : \mathsf{S}[x] \boxplus \mathsf{S}'[x]$}
    $c_i \leftarrow p_i \oplus k_i^N$                              {Encrypt}
end for
return $C$
```

IV. Poly1305 Authenticator

Poly1305 is a message authentication code based on a polynomial evaluation over the prime field $2^{130} - 5$. It takes as input a *one-time* 32-byte secret key (r, s) and a ℓ-byte message M, and returns a 16-byte authentication tag T:

$$\begin{aligned}\mathsf{Poly1305} \quad : \quad & \{0,1\}^{128} \times \{0,1\}^{128} \times \{0,1\}^{8\ell} \to \{0,1\}^{128}, \\ & (r, s, M) \mapsto T.\end{aligned}$$

The authentication tag $T = \mathsf{Poly1305}(r, s, M)$ is computed as follows: first, the function $\mathsf{Pad1305} : \{0,1\}^{8\ell} \to \{0,1\}^{136q}$, $M \mapsto (m_i)_{1 \le i \le q}$ chops the input message M into $q = \lceil \ell/16 \rceil$ 17-byte chunks $(m_i)_{1 \le i \le q}$, where each 16-byte block is padded with the byte 0x01 and the last block is padded with the byte 0x01 and $(16 - \ell) \bmod 16$ zero bytes. Then, the function $\mathsf{Clamp} : \{0,1\}^{128} \to \{0,1\}^{128}, r \mapsto \bar{r}$ clears some bits of r, such that $\bar{r} = r_0 + r_1 + r_2 + r_3$, where $r_0 \in \{0, 1, 2, 3, ..., 2^{28} - 1\}$, $r_1/2^{32} \in \{0, 4, 8, 12, ..., 2^{28} - 4\}$, $r_2/2^{64} \in \{0, 4, 8, 12, ..., 2^{28} - 4\}$, and $r_3/2^{96} \in \{0, 4, 8, 12, ..., 2^{28} - 4\}$. This allows saving costly carry propagations, at the (marginal) loss of a few bits of security. Finally, the authentication tag T is generated by evaluating the polynomial defined by the coefficients $(m_i)_{1 \le i \le q}$ at $\bar{r}$, adding the key s, and truncating the results to 128-bit:

$$T = \left(\sum_{i=1}^{q} m_i \bar{r}^{q-i+1} \bmod 2^{130} - 5 \right) + s \bmod 2^{128}. \quad (1)$$

Algorithm 2 Poly1305-ChaCha20 Authenticator:
$T = \mathsf{Poly1305\text{-}ChaCha20}(K, N, M)$.

```
Input:   $K \in \{0,1\}^{256}$, $N \in \{0,1\}^{96}$, $M \in \{0,1\}^{8\ell}$
Output:  $T \in \{0,1\}^{128}$
/* ChaCha20 */
$(r, s) \leftarrow \mathsf{ChaCha20}(K, N, 0)$   {Compute one-time key}
/* Poly1305 */
$(m_1, ..., m_q) \leftarrow \mathsf{Pad1305}(M)$   {Chop $M$ into $q$ chunks}
$\bar{r} \leftarrow \mathsf{Clamp}(r)$                       {Clear bits of $r$}
$h \leftarrow 0$
for $i \leftarrow 0$ to $(q - 1)$ do                {Evaluate Polynomial}
    $h \leftarrow h + m_{i+1} \bar{r}^{q-i} \bmod 2^{130} - 5$
end for
$T \leftarrow h + s \bmod 2^{128}$                          {Generate Tag}
return $T$
```

In the original paper [4], AES is used to generate the one-time 16-byte secret key s, using a 16-byte nonce n, and a 16-byte secret key k, as $s = \mathsf{AES}_k(n)$. In this case, the authenticator was named Poly1305-AES. Similarly, the authenticator Poly1305-ChaCha20 (cf. Algorithm 2) uses the ChaCha20 stream cipher to generate the one-time 32-byte secret key (r, s) using a 32-byte secret key K and a 12-byte nonce N, as $r||s = \mathsf{ChaCha20}(K, N, 0)$. Note that only 32 bytes of $\mathsf{ChaCha20}$ output are used in this case.

V. ChaCha20-Poly1305 Authenticated Encryption with Associated Data (AEAD)

The ChaCha20 stream cipher and the Poly1305 authenticator can be combined together to realize the ChaCha20-Poly1305 Authenticated Encryption with Associated Data (AEAD) scheme, as specified in the RFC7539 [5]. The IND-CPA and INT-CTXT security of this scheme has been proven in the random oracle model, assuming that ChaCha20 is a PRF and Poly1305 is an ϵ-almost-Δ-universal hash function [8]. The ChaCha20-Poly1305 AEAD takes as input a 32-byte secret key K, a 12-byte nonce N, a variable-length plaintext P, and a variable-length associate data AD, and returns a ciphertext C and a 16-byte authentication tag T:

$$\mathsf{CP} : \{0,1\}^{256} \times \{0,1\}^{96} \times (\{0,1\}^*)^2 \to \{0,1\}^* \times \{0,1\}^{128},$$

such that $(K, N, P, AD) \mapsto (C, T)$. The scheme is illustrated in Algorithm 3, where the function $\mathsf{L} : \{0,1\}^* \to \{0,1\}^{64}$ returns the length of the input as 64-bit little-endian integers, and the function $\mathsf{Pad} : \{0,1\}^{8\ell} \to \{0,1\}^{8(\ell+\delta)}$ returns the ℓ-byte input padded with $\delta = (16 - \ell) \bmod 16$ zero bytes. Note that the encryption is performed using the ChaCha20 stream cipher with $\gamma \ge 1$, because $\gamma = 0$ is used for generating the one-time key (r, s) and therefore it can not be re-used for encryption. Decryption works in the similar way, where the role of the plaintext and the ciphertext is inverted and the generated tag must be bitwise compared with the received tag in order to verify the authenticity of data. Note that the Poly1305-ChaCha20 authenticator takes the ciphertext as input for both ChaCha20-Poly1305 encryptions and decryptions.

Algorithm 3 ChaCha20-Poly1305 Authenticated Encryption Scheme.

```
Input: $K \in \{0,1\}^{256}$, $N \in \{0,1\}^{96}$, $P \in \{0,1\}^{*}$, $AD \in \{0,1\}^{*}$
Output: $C \in \{0,1\}^{|P|}, T \in \{0,1\}^{128}$
$C \leftarrow \mathsf{ChaCha20\text{-}SC}(K,N,P)$                          {Encrypt}
$M \leftarrow \mathsf{Pad}(AD)||\mathsf{Pad}(C)||\mathsf{L}(AD)||\mathsf{L}(C)$   {Pad Message}
$T \leftarrow \mathsf{Poly1305\text{-}ChaCha20}(K,N,M)$                  {Authenticate}
return $(C,T)$
```

VI. Implementation Details

In this section, assembly level optimizations of ChaCha20 and Poly1305 for ARM Cortex-M4 processors, aimed at both high-speed and compact code-size, are presented.

A. ChaCha20

We optimized the 20 rounds of ChaCha20 in two ways: (1) we optimized the double rounds by rearranging the order of the quarter rounds and by minimizing the amount of memory operations; (2) we optimized the quarter rounds by exploiting the "flexible second operand" feature (cf. Section II).

Double rounds: At most 14 words of the ChaCha20 state can be held in the registers at the same time. We load the words $\mathsf{S}[0]$ to $\mathsf{S}[11]$ into the registers and we keep them for all 20 rounds, while we use the remaining 2 registers to hold alternately the words $\mathsf{S}[12]$ to $\mathsf{S}[15]$, and the loop iterator. Each quarter round operates on only one of the four words $\mathsf{S}[12]$ to $\mathsf{S}[15]$. Note, that the loop iterator could be removed through loop unrolling. However, we decided against it, to preserve a reasonably small code size. At the beginning of each double round, the loop iterator and $\mathsf{S}[12]$ are in the 2 registers. The remaining 3 words are located on the stack. We swap the loop iterator with $\mathsf{S}[13]$. Then, the quarter rounds $\mathsf{QR}(\mathsf{S}[0],\mathsf{S}[4],\mathsf{S}[8],\mathsf{S}[12])$ and $\mathsf{QR}(\mathsf{S}[1],\mathsf{S}[5],\mathsf{S}[9],\mathsf{S}[13])$ are performed. Then, we swap $\mathsf{S}[12]$ and $\mathsf{S}[13]$ with $\mathsf{S}[14]$ and $\mathsf{S}[15]$. All column rounds are independent from each other, and all diagonal rounds are independent from each other, because they operate on different words of the state. Therefore, we can freely change the order of the column rounds and the order of the diagonal rounds. We exploit this fact to perform the column rounds $\mathsf{QR}(\mathsf{S}[2],\mathsf{S}[6],\mathsf{S}[10],\mathsf{S}[14])$, $\mathsf{QR}(\mathsf{S}[3],\mathsf{S}[7],\mathsf{S}[11],\mathsf{S}[15])$, and the diagonal rounds $\mathsf{QR}(\mathsf{S}[0],\mathsf{S}[5],\mathsf{S}[10],\mathsf{S}[15])$, $\mathsf{QR}(\mathsf{S}[3],\mathsf{S}[4],\mathsf{S}[9],\mathsf{S}[14])$ without swapping any more words. We then swap $\mathsf{S}[14]$ and $\mathsf{S}[15]$ with $\mathsf{S}[12]$ and $\mathsf{S}[13]$, and perform the remaining two diagonal rounds $\mathsf{QR}(\mathsf{S}[1],\mathsf{S}[6],\mathsf{S}[11],\mathsf{S}[12])$ and $\mathsf{QR}(\mathsf{S}[2],\mathsf{S}[7],\mathsf{S}[8],\mathsf{S}[13])$. To complete the loop iteration, we swap $\mathsf{S}[13]$ with the loop iterator. Combined, we use only 6 swapping operations per double round.

Quarter rounds: We slightly modify the definition of the QR function from $(i,j,k,l) = \mathsf{QR}(a,b,c,d)$ to $(i,j',k,l') = \mathsf{QR}(a,b',c,d')$, where $b' = b \ggg 7$,$d' = d \ggg 8$, $j' = j \ggg 7$, $l' = l \ggg 8$, in order to effectively exploit the "flexible second operand" feature. Let the words in the 2^{nd} and the 4^{th} rows be denoted as b-words and d-words, respectively. Instead of calculating the rotations with a ROR instruction, we skip this instruction and fix the missing rotations by rotating registers directly, when they are added or XOR-ed, using the "flexible second operand" feature:

$$\begin{cases} e &= a \boxplus (b' \lll \ 7) \quad ; \quad h' = e \oplus (d' \lll \ 8) \\ g &= c \boxplus (h' \lll 16) \quad ; \quad f' = g \oplus (b' \lll \ 7) \\ i &= e \boxplus (f' \lll 12) \quad ; \quad l' = i \oplus (h' \lll 16) \\ k &= g \boxplus (l' \lll \ 8) \quad ; \quad j' = k \oplus (f' \lll 12) \end{cases},$$

This requires the b-words $\mathsf{S}[1]$, $\mathsf{S}[5]$, $\mathsf{S}[9]$, $\mathsf{S}[13]$ to be rotated by 7 bits to the right, and the d-words $\mathsf{S}[3]$, $\mathsf{S}[7]$, $\mathsf{S}[11]$, $\mathsf{S}[15]$ to be rotated by 8 bits to the right, before the first quarter round. After the last quarter round of the 20 rounds, the b-words are rotated by 7 bits to the left and the d-words are rotated by 8 bits to the left. In between any quarter rounds, no rotations have to be done, as the outputs j' and l' of one quarter round function are the inputs b' and d' of later quarter round functions. This trick reduces the amount of instructions from 960 to 656.

B. Poly1305

Using the Horner's method, the inner part of Equation (1) can be efficiently computed using only additions and multiplications by $\bar{r}$:

$$h = ((...(m_1\bar{r} + m_2)\bar{r} + ... + m_{q-1})\bar{r} + m_q)\bar{r} \bmod 2^{130} - 5.$$

Then, the authentication tag $T = h + s \bmod 2^{128}$ is obtained by a regular addition by s and a truncation to 128-bit. Note that no pre-computations are reasonable here as $\bar{r}$ is a one-time key. The pseudo-code is shown in Algorithm 4. We optimized Poly1305 in three ways: (1) we handle the complete chunks $(m_1, ..., m_{q-1})$ and the last (possibly incomplete) chunk m_q separately. In fact, the complete chunks always consist of 16 bytes of the input and one byte of padding. As the padding byte corresponds to the fixed value 0x01 at a fixed position 17^{th}, we don't store it anywhere and exploit this information directly in the code. In case the input's length is a multiple of 16, then the last chunk m_q is complete and it is also handled this way. However, in case the input's length is *not* a multiple of 16, then the last chunk m_q is handled separately, because the location of the padding byte 0x01 is not at a fixed position. If the last chunk is incomplete, we apply the padding and iterate over it separately, i.e. it is copied to a separate array along with the padding bytes. This trick allows for a big speed-up of the overall code. Also note that during the looping a copy of the key $\bar{r}$ is kept in 4 registers, as to avoid the costs associated with repeatedly loading the key in each loop iteration. (2) We exploit the fact that $\bar{r}$ is clamped to realize a very fast modular multiplier in radix-2^{32}. (3) We exploit fast and lazy reduction techniques to speed up the computation of modular operations. That is, in each iteration we load a new 16-byte block, add it to the current state, multiply the result by $\bar{r}$, and apply a fast reduction to reduce the value to $< 2^{130}+2^{129}$. A full reduction modulo $2^{130} - 5$ is performed only at the end, when all the chunks have been processed and before the authentication tag T is computed.

Algorithm 4 Poly1305 Evaluation.

```
Input: $\bar{r}$ and $(m_i)_{1 \leq i \leq q}$.
Output: $h \in \mathbb{F}_{2^{130}-5}$.
$h_0 \leftarrow 0$
for $i \leftarrow 1$ to $q$ do
    $h_i \leftarrow h_{i-1} + m_i$                                {Add}
    $h_i \leftarrow h_i \bar{r}$                {Multiply and Fast Reduction}
end for
$h \leftarrow h_q \bmod 2^{130} - 5$                       {Final Reduction}
return $h$
```

Addition: The addition is performed straightforwardly using 5 addition instructions. The two inputs are the output of the previous reduction step $h_i \bar{r} < 2^{130} + 2^{129}$ and the next chunk to be processed $m_{i+1} < 2^{129}$. After the addition the result is $h_i \bar{r} + m_{i+1} < 2^{131}$.

Multiplication: We implement a 131×124-bit multiplication using product scanning as shown in Listing 1, where a is the output of the previous addition step h_i and b is the clamped key $\bar{r}$. Note that the following relations are always true due to the properties of the key $\bar{r}$: $a_4 < 2^3$, $(b_j)_{0 \leq j \leq 3} < 2^{28}$, and $a_4 b_i < 2^{31}$. The multiplication works as follows: first, the value $a_0 b_0$ is computed using the UMULL instruction and the 60-bit result is stored into two registers $(p_0, p_1) \leftarrow a_0 b_0$. Note that the register p_1 contains a value $< 2^{28}$. Then, a register p_2 is set to zero and the products $a_0 b_1$ and $a_1 b_0$ are accumulated into (p_1, p_2) using the UMLAL instruction. After the accumulations, p_1 contains the final result. The process continues till p_7 is computed. We have enough registers to carry out the complete multiplication. Note that all intermediate products $a_i b_j$ can be accumulated into p_{i+j} and p_{i+j+1} without worrying about overflows. The worst case happens for p_3 and p_4 where 4 accumulations are necessary. Note that the registers p_5, p_6, p_7 are actually the same registers as a_0, a_1, a_2. Their content is overwritten by an intermediate result as soon as the parts of a are no longer needed. We accumulate the result by using the UMLAL instruction. Whenever we add a new register to hold the accumulation, we need to make sure, that is set to zero. This can be bypassed by inserting a result of one of the multiplications, that affects only this register. Using this trick, we managed to avoid all set-to-zero instructions except for one by using the instructions UMULL p_4, p_2, a_4, b_0 MUL p_5, a_4, b_1 MUL p_6, a_4, b_2 MUL p_7, a_4, b_3. This was possible, because the result of $a_4 \times b_i$ is always smaller than 2^{31}. The instruction UMULL p_4, p_2, a_4, b_0 uses UMULL instead of MUL, in order to write 0 into the register p_2. Apart from the necessary $5 \times 4 = 20$ multiplication instructions, and one instruction to set a register to zero, no other instructions are used in our code. Most importantly no carries have to be propagated and no memory instructions for storing intermediate results have to be used as enough registers are there. Our 131×124-bit multiplication costs only 21 instructions. Due to the size of the operands, the particular properties of $\bar{r}$, and the availability of very fast and powerful instructions, such as UMLAL and UMULL instructions, we speculate that other approaches to implement such a multiplication on ARM Cortex-M4 processors (e.g. Karatsuba) will perform similar to, or even worse than, our product scanning based multiplication.

Listing 1 131×124-bit Product Scanning Multiplication.

```
Input:   $a = (a_0, a_1, a_2, a_3, a_4)$ and $b = (b_0, b_1, b_2, b_3)$.
Output:  $p = (p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7)$ s.t. $p = ab$.
SUB     $p_3, p_3$                      {Set $p_3$ to 0}
UMULL   $p_0, p_1, a_0, b_0$              {$[p_0, p_1] = a_0 b_0$}
UMULL   $p_4, p_2, a_4, b_0$              {$p_4 = a_4 b_0$, $p_2 = 0$}
UMLAL   $p_1, p_2, a_0, b_1$              {$[p_1, p_2] += a_0 b_1$}
UMLAL   $p_1, p_2, a_1, b_0$              {$[p_1, p_2] += a_1 b_0$}
UMLAL   $p_2, p_3, a_0, b_2$              {$[p_2, p_3] += a_0 b_2$}
UMLAL   $p_2, p_3, a_1, b_1$              {$[p_2, p_3] += a_1 b_1$}
UMLAL   $p_2, p_3, a_2, b_0$              {$[p_2, p_3] += a_2 b_0$}
UMLAL   $p_3, p_4, a_0, b_3$              {$[p_3, p_4] += a_0 b_3$}
UMLAL   $p_3, p_4, a_1, b_2$              {$[p_3, p_4] += a_1 b_2$}
UMLAL   $p_3, p_4, a_2, b_1$              {$[p_3, p_4] += a_2 b_1$}
UMLAL   $p_3, p_4, a_3, b_0$              {$[p_3, p_4] += a_3 b_0$}
MUL     $p_5, a_4, b_1$                   {$p_5 = a_4 b_1$}
UMLAL   $p_4, p_5, a_1, b_3$              {$[p_4, p_5] += a_1 b_3$}
UMLAL   $p_4, p_5, a_2, b_2$              {$[p_4, p_5] += a_2 b_2$}
UMLAL   $p_4, p_5, a_3, b_1$              {$[p_4, p_5] += a_3 b_1$}
MUL     $p_6, a_4, b_2$                   {$p_6 = a_4 b_2$}
UMLAL   $p_5, p_6, a_2, b_3$              {$[p_5, p_6] += a_2 b_3$}
UMLAL   $p_5, p_6, a_3, b_2$              {$[p_5, p_6] += a_3 b_2$}
MUL     $p_7, a_4, b_3$                   {$p_7 = a_4 b_3$}
UMLAL   $p_6, p_7, a_3, b_3$              {$[p_6, p_7] += a_3 b_3$}
```

Fast Reduction: We represent the result of the multiplication as $p = d_0 + d_1 2^{130}$, where $d_0 < 2^{130}$ and $d_1 < 2^{126}$. Hence, the fast reduction is applied as $d = d_0 + 5d_1 < 2^{130} + 2^{129} < 2^{131}$. Using such a reduction would require several shifts to extract d_1 and to add $5d_1$ back. However, we exploit the "flexible second operand" again to shift, multiply by 5 and accumulate at the same time. Initially, the upper two bits of d_0 and the lower 30 bits of d_1 share a register. They are split apart, using an additional register. The result is d_0 covering 5 registers and d_1, shifted by 2 bits to the left, covering 4 registers. A shift by two to the left is equal to a multiplication by 4. We add $4d_1$ to d_0 with 5 ADD/ADC instructions. Then, we add $4d_1$, shifted by two to the right, to d_0 with 10 ADD/ADC instructions with shifted second operands. Shifting $4d_1$ by two to the right is equal to dividing it by 4. Effectively, we compute $d = d_0 + 4d_1 + (4d_1)/4 = d_0 + 5d_1$.

Final Reduction: As a fast reduction has just taken place, then h_q is guaranteed to be $< 2^{130} + 2^{129}$. Hence, we need to compute $h_q + 5 - 2^{130}$ if $h_q \geq 2^{130} - 5$ to finally reduce the result. Our implementation is constant-time and works as follows: first, $g \leftarrow h_q + 5$ is computed and stored in 5 free registers. The register that includes the 130^{th}-bit of g is copied to another register. Then, the 130^{th}-bit of g is set to zero, which is equal to $g \leftarrow g - 2^{130}$ if the 130^{th}-bit of g was set. Depending on the 130^{th}-bit of g (before was set to zero), one of two results is returned.

TABLE I
ENCRYPTION OF 64-BYTE INPUT DATA. MAC OF 128-BYTE INPUT DATA. AEAD OF 16-BYTE INPUT DATA AND 16-BYTE ASSOCIATED DATA.

	Platform	Algorithm	Runtime [Cycles]	Speed‡ [Cycles/Byte]	Size [Byte]	Stack [Byte]
Encryption	8-bit AVR ATmega [9]	Salsa20	17,787	268.0	–	268
	32-bit ARM Cortex-M4 [10]	Salsa20	3,311	–	1,272	552
	32-bit ARM Cortex-M0 [11]	ChaCha20	–	39.9	–	–
	32-bit ARM Cortex-M4 [10]	ChaCha20	3,468	–	1,328	544
	32-bit ARM Cortex-M4 [12]†	ChaCha20	1,287	17.6	3,174	228
	32-bit ARM Cortex-M4 (This Work)	**ChaCha20**	**1,487**	**20.6**	**734**	**232**
MAC	32-bit ARM Cortex-M0 [13]	Chaskey	–	18.3	1,308	–
	32-bit ARM Cortex-M4 [13]	Chaskey	–	7.0	908	–
	8-bit AVR ATmega [9]	Poly1305	–	195.0	–	148
	32-bit ARM Cortex-M4 (This Work)	**Poly1305**	**747**	**3.6**	**744**	**120**
	32-bit ARM Cortex-M4 (This Work)	**Poly1305-ChaCha20**	**1,945**	**3.6**	**1,322**	**224**
AEAD	32-bit ARM Cortex-M4 [10]	AES128-GCM	43,657	–	2,644	812
	32-bit ARM Cortex-M4 [10]	AES128-EAX	32,159	–	2,780	932
	32-bit ARM Cortex-M4 [10]	AES128-CCM	23,949	–	2,256	780
	32-bit ARM Cortex-M4 [10]	NORX32	6,855	–	1,820	320
	32-bit ARM Cortex-M4 (This Work)	**ChaCha20-Poly1305**	**3,364**	**28.4**	**1,946**	**332**

† These results were obtained by including the π-ChaCha20 function (cf. https://github.com/joostrijneveld/chacha-arm-cortex-m/) in our code.
‡ Asymptotic values.

VII. PERFORMANCE EVALUATION AND COMPARISON

The software was cross-compiled using the GNU Compiler Collection for ARM Embedded Processors version `4.9.3` release `20150529` with the options `-O2 -mthumb -mcpu=cortex-m4` and tested on a `STM32F411RE` board [14]. The code size was measured summing up the size of the `.text`, `.bss` and `.data` segments, while the runtime was measured using the internal clock cycle counter (CYCCNT). The speed is expressed in cycles/byte and represents the asymptotic speed which is achieved when the length of the messages is large. Note that the reported speed and runtime include the overheads for calling/returning from the considered functions, while all input data was assumed to be already word aligned. The stack usage was estimated by filling the memory with a canary and then checking how many words were changed. Table I provides a comparison overview of the implementation results. Our ChaCha20 implementation is slower than [12], but it is only $1/4$ of the size. This allows us to use dedicated functions of ChaCha20 to speed up Poly1305-ChaCha20 and ChaCha20-Poly1305 while retaining a small code size. Due to the lack of publicly available and directly comparable results of AEAD and MAC algorithms on ARM Cortex-M processors, our comparison extends to other algorithms and low-cost processors. In particular, our MAC and AEAD results are compared to Chaskey and NORX32, respectively. NORX is a third-round AEAD candidate running in the CAESAR competition, whose design is based on ChaCha and Keccak. NORX32 is an instance of the NORX family using 32-bit words, hence providing a good fit for the comparison. Chaskey is a MAC designed for high performance on 32-bit microcontrollers [13]. Finally, we considered the results from the open source Cifra project [10], though not assembly optimized nor peer reviewed. Note that, while our results refer to constant time executions, this might not be the case for the other entries in the table.

VIII. CONCLUSION

In this work, we presented high-speed, constant-time, and compact implementations of the ChaCha20 stream cipher, Poly1305-ChaCha20 authenticator, and ChaCha20-Poly1305 authenticated encryption scheme on ARM Cortex-M4 processors. Our results put the ChaCha20 and Poly1305 ciphers among the most promising candidates to secure emerging IoT applications with tight speed and space constraints, and to support TLS secured communication on ARM Cortex-M4 devices, according to the RFC7905 and RFC7539. The software is available at https://gitlab.lrz.de/tueisec/ChaCha20-Poly1305-ARM-Cortex-M4.

REFERENCES

[1] D. J. Bernstein, “Failures of Secret-Key Cryptography,” Invited Talk at FSE 2013, https://cr.yp.to/talks/2013.03.12/slides.pdf, FSE 2013.

[2] “CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness,” http://competitions.cr.yp.to/caesar.html, 2012.

[3] D. J. Bernstein, “ChaCha, a variant of Salsa20,” January 2008.

[4] ——, “The Poly1305-AES Message-Authentication Code,” in *FSE 2015*. LNCS 3557, 2005, pp. 32–49.

[5] Y. Nir and A. Langley, “ChaCha20 and Poly1305 for IETF Protocols,” RFC 7539, https://rfc-editor.org/rfc/rfc7539.txt, 2015.

[6] A. Langley, W.-T. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson, “ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS),” RFC 7905, https://rfc-editor.org/rfc/rfc7905.txt, 2016.

[7] ARM®, “ARM® and Thumb®-2 Instruction Set,” http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf, 2016.

[8] G. Procter, “A Security Analysis of the Composition of ChaCha20 and Poly1305,” Cryptology ePrint Archive, Report 2014/613, 2014.

[9] M. Hutter and P. Schwabe, “NaCl on 8-Bit AVR Microcontrollers.” in *AFRICACRYPT 2013*. LNCS 7918, 2013, pp. 156–172.

[10] “The Cifra Project,” https://github.com/ctz/cifra, September 2016.

[11] M. Neikes and N. Samwel, “ARM Cortex-M0 implementation of ChaCha20,” https://gitlab.science.ru.nl/mneikes/arm-chacha20, 2016.

[12] A. Hülsing, J. Rijneveld, and P. Schwabe, “ARMed SPHINCS – Computing a 41 KB Signature in 16 KB of RAM,” in *PKC 2016*. LNCS 9614, 2016, pp. 446–470.

[13] N. Mouha, B. Mennink, A. Van Herrewege, D. Watanabe, B. Preneel, and I. Verbauwhede, “Chaskey: An Efficient MAC Algorithm for 32-bit Microcontrollers,” in *SAC 2014*. LNCS 8781, 2014, pp. 306–323.

[14] STMicroelectronics, “STM32F411RE High-performance access line,” http://www2.st.com/resource/en/datasheet/stm32f411re.pdf, 2016.