Comparison of Three CPU-Core Families for IoT Applications in Terms of Security and Performance of AES-GCM

Yaroslav Sovyn, Volodymyr Khoma, and Michal Podpora

Abstract—This article describes the implementation of the AES-GCM for IoT-oriented low-end 8/16/32-bit general-purpose processors. Although various aspects of implementations of the AES-GCM for high-end processors and hardware were examined in detail, the low-end processors to a lesser extent. This article estimates the speed and memory demand for various approaches to ensuring resistance to attacks, such as timing analysis and simple power analysis by ensuring the constant algorithm execution time. A particular attention is paid to the low-level multiplication implementation in GF(2^{128}) for each architecture as a key galois/counter mode operation, because low-end processors do not have ready-made instructions for carry-less multiplication. For each AVR/MSP430/ARM Cortex-M3 processor core, a constant time implementation of carry-less multiplication is proposed, the performance of which approaches the Not Constant Time algorithm.

Index Terms—AES-GCM, authenticated encryption with associated data (AEAD), carry-less multiplication, embedded systems, GF(2^{128}) multiplication, IoT, performance, side-channel attack, simple power analysis, timing analysis.

I. INTRODUCTION

According to the IoT concept, the IoT devices not only need to be able to communicate by sending and receiving information, but they are also expected to use a secure communication method or protocol in a public network [1], [2]. Of course, various aspects of this interaction must be reliably protected. This is especially important in the case of critical infrastructure, which determines the life and health of people. IoT devices can be perceived as interconnected Embedded Systems, and their design usually includes microcontrollers (MCUs), MCUs usually offer very limited computing power, and they have relatively little ROM and RAM memory. Therefore, the cryptographic algorithms implemented on embedded systems must be efficient (use minimum resources) and be resistant to a wide range of attacks including side-channel attacks. The main challenge of the implementation of cryptographic algorithms for embedded systems is the fulfillment of seemingly conflicting requirements of the level of security, performance, and price of the final device.

Almost all cryptographic algorithms have been designed to be implemented in computer systems that use universal processors providing sufficient CPU power. Therefore, the software implementation of encryption or hashing on 8/16/32-bit MCUs is slow and energy-intensive and requires quite a lot of memory. The search for new and adaptation of existing algorithms on platforms with limited computing power is covered by lightweight cryptography [3], [4]. Another approach to solving the problem of the complexity of encryption algorithms assumes the use of cryptographic accelerators to speed up calculations hundreds of times [5]–[7].

IoT applications are operating using standard or custom protocols, sending data in the form of packets. These packages can contain both: confidential information, which should be protected from unauthorized reading, modification, and falsification, as well as nonconfidential data. The nonconfidential data include packet header (which may be tampered or replaced), including addresses, port numbers, protocol versions, and other information needed to handle the packet. The header must be authenticated while also it should remain unencrypted so that network devices can read it.

Cryptographic primitives to comprehensively solve similar issues are covered by the term authenticated encryption with associated data (AEAD). In the AEAD algorithms, a part of the message is encrypted and the other part remains in the open form, but the entire packet is authenticated. AEAD encryption—it is a mode of symmetric encryption of data transmitted in the form of packets, which simultaneously provides both: confidentiality and data authentication, using a single software interface.

AEAD algorithms are more efficient and simpler than using the separate algorithms for encryption and authentication, and thus require fewer resources. It also allows to avoid critical errors when combining encryption and authentication, which led to a series of practical attacks on protocols and applications, including SSL/TLS [8].

A typical software interface implementing the AEAD method provides the following functions.

1) Encryption:
   Input: key, initial vector, plaintext, and optional header;
   Output: ciphertext and authentication tag (Fig. 1).
2) **Decryption**:

   **Input**: key, initial vector, ciphertext, authentication tag, and optional header;
   
   **Output**: Plaintext or error if the calculated tag does not match the specified one.

In IoT applications and protocols, security is crucial [9], [10], which is why there is an objective need to use AEAD algorithms.

### II. Analysis of Recent Research and Publications

The AEAD algorithms are used in known protocols such as TLS 1.2 and in its newer version TLS 1.3 for securing data transmission in computer networks. The TLS 1.2 protocol has been implemented in many libraries, including lightweight, oriented toward embedded systems, and IoT devices, among which the most well-known are WolfSSL, GUARD TLS Tiny, mbed TLS, and CycloneSSL. However, the TLS protocol is quite “heavy” for IoT, so as an alternative, the relatively new noise protocol can be considered, which reduces the complexity and computational power requirements of TLS.

Taking this into account, one of the algorithms most commonly used today was selected for the research: AES-GCM. This algorithm is supported by TLS 1.2, 1.3, and noise protocols. Moreover, what is very important, it has been standardized in NIST SP 800-38D and RFC5116 [11], [12].

The need for authenticated encryption algorithms confirmed the CAESAR 2013 competition (Competition for Authenticated Encryption: Security, Applicability, and Robustness) [13] to select algorithms intended to be more efficient than AES-GCM and provide resistance to reuse/misuse attacks. Six winners for various uses of ciphers were listed in February 2019.

Since the approval of the AES-GCM standard in 2007, even systems built upon high-performance general-purpose processors with a high clock frequency, large RAM, and cache capacity and a powerful instruction set encounter a problem of insufficient performance for its implementation. The AES-GCM algorithm includes two operations: 1) AES data encryption in the counter mode and 2) Galois hash (GHASH) hashing based on multiplication in the Galois field with modular reduction.

The last operation is much slower than encryption. Therefore, in 2010, AES-NI encryption instructions were added to x86 processors (Table I), as well as a special PCLMULQDQ (Carry-Less Multiplication) instruction for multiplication within Galois Field $GF(2^{128})$ [14], [15]. This, after using various techniques, allowed to optimize calculations (especially modular reduction) [16] to significantly increase the efficiency of AES-GCM (Fig. 2).

With the advent of NEON SIMD-instructions in ARMv7 processors, Carry-Less Multiplication can be accelerated using the VMULL.P8 command, which performs simultaneous (at the same time) polynomial multiplication of $8 \times 8$ bits for an 8-byte vector [19]. By combining this instruction with other SIMD-instructions of the NEON unit, a carry-less product calculation of $128 \times 128$ bits is built.

In the ARMv8 processors, four special SIMD instructions have been added to support AES-GCM, including encryption, AES (AESE/AESD—AES single-round encryption/decryption, AESIMC—AES inverse mixed columns, and AESMC—AES mixed columns), as well as $64 \times 64$ multiplication PMULL in Galois field [20]. In [21], it has been shown that their use increases the efficiency of AES-GCM by nine times compared to ARMv7.

Even more noticeable is the lack of hardware support for the implementation of AES-GCM in MCUs. Only some 8/16-bit MCU models are equipped with the AES-128 algorithm cryptographic accelerators, for example, 8-bit AVR encrypts one block for 375 cycles and 16-bit MSP430 for 167 cycles.
More often, the AES crypto-modules are found in 32-bit MCUs with ARM Cortex-M core, which, depending on the type of the crypto-module, perform block encryption between 12 and 168 cycles [7]. However, all MCUs lack any hardware support for carrying out the carry-less multiplication necessary to calculate GHASH (which is 2–4 times more complex than AES encryption). In addition, GHASH enumeration must be performed more often because it also applies to the attached packet data that is not encrypted. That is why in this article special attention was given to the optimization of GHASH calculation, taking into account MCUs architectures and their command sets.

As for the implementation of the AES algorithm, there are well-known efficient computational techniques based on the use of so-called T-tables [22]. These are four precomputed T0–T3 look-up tables with a size of 1 kB (256 × 32 bits)× (at the same time) secure against timing analysis and simple power analysis low-level multiplication implementation in GF (2128) for each architecture as a key GCM operation, because low-end processors do not offer built-in carry-less multiplication instructions.

IV. ANALYSIS OF THE 8/16/32-BIT MICROCONTROLLER ARCHITECTURES IN TERMS OF AES-GCM OPERATIONS

Considering that there is no dominant platform in IoT, it is important to estimate the implementation of the AES-GCM algorithm on different market segments of embedded systems' processors: low-end (8/16 bits) and high-end (32 bits). One typical architecture from each 8-, 16-, and 32-bit MCU architectures was selected for the study.

A. AVR Microcontrollers (8 Bits)

As an 8-bit platform, the AVR MCUs family was chosen. This choice is a result of a rich instruction set designed for maximum effectiveness of programs written in high-level languages.

It is worth noting that among the features of the AVR core, important in the context of cryptography, is Harvard memory architecture with separate memory—8-bit data memory (using SRAM) and 16-bit program memory (using Flash), which increases efficiency. The entire family of AVR MCUs is based on the typical RISC architecture. The register system holds 32 8-bit general-purpose registers directly connected to the ALU, performing arithmetic, logic, and bit operations. The instruction set is sufficiently developed and contains over 130 instructions, most of which are performed in one cycle, thanks to the use of a two-level pipeline. The core contains 8 × 8 multiplication unit, executing the MUL instruction in 2 cycles [26].

AVR MCUs support simple, direct, and indirect addressing modes. The availability of predecrementing, post-incrementation, and the offset modes in indirect addressing enables efficient processing of data sets in the process of cryptographic algorithm execution, generating a compact program code. For access to data in Flash memory (S-box and look-up table), indirect addressing is used. Access to the SRAM takes place in two cycles, three cycles are required for Flash.

B. MSP430 Microcontrollers (16 Bits)

The MSP430 family gained popularity in the IoT technology, especially in wireless sensor networks, due to very low energy consumption.

The compact 16-bit RISC MSP430 core is built according to Princeton architecture and contains 16 registers, of which

### Table II

| Parameter                  | MSP430 (16 bit) | ARM Cortex-M3 (32 bit) |
|----------------------------|-----------------|------------------------|
| Encryption/Decryption (CPB)| 863/862         | 2769/-                 |
| ROM/RAM [Bytes]            | 7166/884        | 2644/812               |
| Plaintext/Header [Bytes]   | 16/0            | 16/16                  |

This article aims to explore ways of effective software implementation of the AES-GCM AEAD algorithm, often used in IoT applications and to estimate and compare the demand for resources for typical low-end 8/16/32-bit processors, because this issue is not highlighted enough in publications known to authors. This will give us a basis for further comparison with the light finalists of the CAESAR competition—Ascon and ACORN algorithms, to estimate the progress achieved. In addition, the priority is to achieve maximum AES-GCM performance, because according to the CAESAR competition requirements, the winners must overtake it.

Given that AES implementations are well developed for low-end processors, the main emphasis is on efficient and (at the same time) secure against timing analysis and simple power analysis low-level multiplication implementation in GF (2128) for each architecture as a key GCM operation, because low-end processors do not offer built-in carry-less multiplication instructions.

III. PURPOSE OF THIS ARTICLE

This article aims to explore ways of effective software implementation of the AES-GCM AEAD algorithm, often used in IoT applications and to estimate and compare the demand for resources for typical low-end 8/16/32-bit processors, because this issue is not highlighted enough in publications known to authors. This will give us a basis for further comparison with the light finalists of the CAESAR competition—Ascon and ACORN algorithms, to estimate the progress achieved. In addition, the priority is to achieve maximum AES-GCM performance, because according to the CAESAR competition requirements, the winners must overtake it.

Given that AES implementations are well developed for low-end processors, the main emphasis is on efficient and (at the same time) secure against timing analysis and simple power analysis low-level multiplication implementation in GF (2128) for each architecture as a key GCM operation, because low-end processors do not offer built-in carry-less multiplication instructions.
processors.

available. When the instruction has a 32-bit shift block that allows one of the operands to be moved by a specific number of bits when the instruction is executed, the Thumb and Thumb-2 instruction set.

The Cortex-M3 core contains 16 registers R0–R15, of which R0–R12 registers are general-purpose registers. The ALU architecture with a three-stage pipeline providing support for calculations in modern microprocessors and the possibility of pipelining and parallel computing. GCM computes encryption and MAC blocks in parallel. GCM uses 128-bit CIPH block cipher, in which AES is often used (CIPH = AES). In the following, under the abbreviation AES-GCM will be understood variant AES-128 GCM, as it is the most economical for IoT applications.

Authenticated encryption function

\[ GCM-AE_K(IV, A, P) = (C, T) \]  

encrypts sensitive data and calculates the authentication tag for both confidential data and added nonconfidential data. The input data for AE encryption include: key K; plaintext P; additional authenticated data (AAD) A; and initialization vector IV. The output data contain: ciphertext C and authentication tag (Tag) T. The tag length can be 128/120/112/104/96 bits. The authors chose the most popular version of the tag for the research—128 bits.

Authenticated decryption function

\[ GCM-AD_K(IV, A, C, T) = (P \text{ or } \text{FAIL}) \]  

decrypts confidential data if the verification of the authentication tag succeeded. The GCM-AD takes K, IV, A, C, and T values as inputs and returns Plaintext P (if the received authentication tag T corresponds to the calculated T*) or the special error code FAIL otherwise.

The AES-GCM algorithm consists of two parts (Fig. 3).

1) AES encryption in AES counter mode (CTR).
2) GHASH authentication to calculate the authentication tag.

A. AES CTR

The algorithm uses AES in the counter mode, generating a key stream at the output of the encryption function, and the cryptogram itself results from the XOR operation of this stream with individual blocks \( P_1, P_2, \ldots, P_n \) of plaintext (Fig. 3).

To generate a key stream, \( Y_0, Y_1, Y_2, \ldots, Y_n \) data blocks are passed to the input of the AES algorithm. The first block \( Y_0 \) is formed as a result of concatenation, i.e., chain coupling of bits of the initial vector IV with the initial content of the counter

\[ Y_0 = CTR = IV ||_0^{31} 1 \]  

The next blocks are calculated in a similar way, but each time the value of the counter is incremented by one incremental function incr_32 (Fig. 3).

The IV initialization vector can be any length, but 96-bit length is recommended (also assumed in this article), due to simplicity and compatibility, otherwise the GHASH function is performed. Considering that AES CTR is a stream cipher, the requirement of IV uniqueness as well as the secret key K is essential for security.

V. STRUCTURE OF THE AES-GCM CRYPTOGRAPHIC ALGORITHM

The Galois/counter mode (GCM) is the most popular standard encryption scheme by NIST, which is used in a number of TLS, Noise, IPSec, SSH, and other protocols. GCM popularity is the result of a lack of patenting, as well as hardware support for calculations in modern microprocessors and the possibility of pipelining and parallel computing. GCM uses 128-bit CIPH block cipher, in which AES is often used (CIPH = AES). In the following, under the abbreviation AES-GCM will be understood variant AES-128 GCM, as it is the most economical for IoT applications.

Authenticated encryption function

\[ GCM-AE_K(IV, A, P) = (C, T) \]  

encrypts sensitive data and calculates the authentication tag for both confidential data and added nonconfidential data. The input data for AE encryption include: key K; plaintext P; additional authenticated data (AAD) A; and initialization vector IV. The output data contain: ciphertext C and authentication tag (Tag) T. The tag length can be 128/120/112/104/96 bits. The authors chose the most popular version of the tag for the research—128 bits.

Authenticated decryption function

\[ GCM-AD_K(IV, A, C, T) = (P \text{ or } \text{FAIL}) \]  

decrypts confidential data if the verification of the authentication tag succeeded. The GCM-AD takes K, IV, A, C, and T values as inputs and returns Plaintext P (if the received authentication tag T corresponds to the calculated T*) or the special error code FAIL otherwise.

The AES-GCM algorithm consists of two parts (Fig. 3).

1) AES encryption in AES counter mode (CTR).
2) GHASH authentication to calculate the authentication tag.

A. AES CTR

The algorithm uses AES in the counter mode, generating a key stream at the output of the encryption function, and the cryptogram itself results from the XOR operation of this stream with individual blocks \( P_1, P_2, \ldots, P_n \) of plaintext (Fig. 3).

To generate a key stream, \( Y_0, Y_1, Y_2, \ldots, Y_n \) data blocks are passed to the input of the AES algorithm. The first block \( Y_0 \) is formed as a result of concatenation, i.e., chain coupling of bits of the initial vector IV with the initial content of the counter

\[ Y_0 = CTR = IV ||_0^{31} 1 \]  

The next blocks are calculated in a similar way, but each time the value of the counter is incremented by one incremental function incr_32 (Fig. 3).

The IV initialization vector can be any length, but 96-bit length is recommended (also assumed in this article), due to simplicity and compatibility, otherwise the GHASH function is performed. Considering that AES CTR is a stream cipher, the requirement of IV uniqueness as well as the secret key K is essential for security.

V. STRUCTURE OF THE AES-GCM CRYPTOGRAPHIC ALGORITHM

The Galois/counter mode (GCM) is the most popular standard encryption scheme by NIST, which is used in a number of TLS, Noise, IPSec, SSH, and other protocols. GCM popularity is the result of a lack of patenting, as well as hardware support for calculations in modern microprocessors and the possibility of pipelining and parallel computing. GCM uses 128-bit CIPH block cipher, in which AES is often used (CIPH = AES). In the following, under the abbreviation AES-GCM will be understood variant AES-128 GCM, as it is the most economical for IoT applications.

Authenticated encryption function

\[ GCM-AE_K(IV, A, P) = (C, T) \]  

encrypts sensitive data and calculates the authentication tag for both confidential data and added nonconfidential data. The input data for AE encryption include: key K; plaintext P; additional authenticated data (AAD) A; and initialization vector IV. The output data contain: ciphertext C and authentication tag (Tag) T. The tag length can be 128/120/112/104/96 bits. The authors chose the most popular version of the tag for the research—128 bits.

Authenticated decryption function

\[ GCM-AD_K(IV, A, C, T) = (P \text{ or } \text{FAIL}) \]  

decrypts confidential data if the verification of the authentication tag succeeded. The GCM-AD takes K, IV, A, C, and T values as inputs and returns Plaintext P (if the received authentication tag T corresponds to the calculated T*) or the special error code FAIL otherwise.

The AES-GCM algorithm consists of two parts (Fig. 3).

1) AES encryption in AES counter mode (CTR).
2) GHASH authentication to calculate the authentication tag.

A. AES CTR

The algorithm uses AES in the counter mode, generating a key stream at the output of the encryption function, and the cryptogram itself results from the XOR operation of this stream with individual blocks \( P_1, P_2, \ldots, P_n \) of plaintext (Fig. 3).

To generate a key stream, \( Y_0, Y_1, Y_2, \ldots, Y_n \) data blocks are passed to the input of the AES algorithm. The first block \( Y_0 \) is formed as a result of concatenation, i.e., chain coupling of bits of the initial vector IV with the initial content of the counter

\[ Y_0 = CTR = IV ||_0^{31} 1 \]  

The next blocks are calculated in a similar way, but each time the value of the counter is incremented by one incremental function incr_32 (Fig. 3).

The IV initialization vector can be any length, but 96-bit length is recommended (also assumed in this article), due to simplicity and compatibility, otherwise the GHASH function is performed. Considering that AES CTR is a stream cipher, the requirement of IV uniqueness as well as the secret key K is essential for security.
B. GHASH

The GHASH algorithm compresses the attached AAD data and ciphertext in one block, which is then encrypted to calculate an authentication tag. To form the HMAC function, in $GF(2^{128})$, GHASH uses the modulo calculation of an irreducible polynomial

$$g = x^{128} + x^7 + x^2 + x^1 + 1.$$  (4)

The data to be authenticated (AAD and ciphertext) are hashed with 128-bit blocks using the $mult$ function. This function multiplies the 128-bit data block $D$ by the 128-bit hash $H$ by the modulo $g$.

The $mult$ function includes two steps:

1) **Polynomial Carry-Less Multiplication:**

$$Z = D \odot H, \ (128 \text{ bits} \odot 128 \text{ bits} \rightarrow 256 \text{ bits}). \quad (5)$$

2) **Reduction:**

$$[Z \text{ mod } (x^{128} + x^7 + x^2 + x^1 + 1)], \ (256 \text{ bits} \rightarrow 128 \text{ bits}).$$ \quad (6)

The last AAD block and the last block of ciphertext are padded by zeroes to 16 bytes, if necessary.

The $H$ hash keying subkey is generated by encrypting the 128-bit AES zero block using the $K$ key

$$H = AES_K(0^{128}). \quad (7)$$

The result calculated by the GHASH function is added modulo 2 to the Y0 block cipher, which gives the value of the authentication tag

$$T = GHASH_H(A, C, len(A)_{64} \| len(C)_{64}) \oplus AES_K(Y_0). \quad (8)$$

VI. RESEARCH METHODOLOGY

For each selected AVR MCU family (8 bits), MSP430 (16 bits), ARM Cortex-M3 (32 bits), and AEAD algorithms were implemented in C language using the embedded workbench for AVR (v7.10.5), IAR embedded workbench for MSP430 (v7.11.2) and IAR Embedded Workbench for ARM (v8.22.1) integrated development environments, respectively. The number of cycles and the size of the code have also been calculated in these environments.

In this article, the maximum speed was generally considered the priority, which is why the compilation was carried out with the optimization level -O hs (High, favoring speed).

The parameters that were measured as follows:

1) encryption/decryption speed expressed in cycles per byte (cpb);
2) amount of permanent memory (ROM), which consists of the size of the program code and tables placed in the flash memory;
3) operational memory (RAM) represented by tables in the SRAM and the desired stack size.

VII. IMPLEMENTATION FEATURES OF THE AES ENCRYPTION ALGORITHM

For all types of MCUs, AES-implementation, based on the T-table, was used to secure the highest performance at a reasonable cost of memory. This requires 4 T0-T3 tables with a size of 1 KB ($256 \times 32$ bits), where each 1024-byte lookup table stores the results of the operations $SubBytes()$, $ShiftRows()$, and $MixColumns()$. 
Due to the fact that in the case of 8-bit AVR MCUs, where memory reads are made bytes, the size of the tables and their number can be optimized. For this purpose, a feature is used that each 32-bit value in the table consists of two identical bytes, which are a linear combination of two other bytes. For example, \( T[0] = 0xc66363a5 \), where \( 0x63 = 0xc6 \oplus 0xa5 \). In this way, you only need to store two different bytes instead of four and calculate the required bytes in the encryption process, which reduces the table size to 512 bytes and reduces the number of cycles. In addition, \( T_0-T_3 \) tables elements with converged indexes contain equal bytes, but in a different order (\( T_0 = 0xc66363a5, T_1 = 0xa5c66363, \) etc.). This is also taken into account during the software byte reading, so only one table \( T_0 \) has to be stored. Thus, the size of the \( T \) table in the case of AVR-MCU is 512 bytes.

For 16- and 32-bit processors, reading 32-bit values provides better performance than byte operations, which is why a traditional approach based on four 1024-byte tables is used.

VIII. IMPLEMENTATION FEATURES OF THE GCM AUTHENTICATION ALGORITHM

In terms of performance, the most critical GCM operation is the carry-less multiplication of two 128-bit operands in a GHASH operation, because the embedded processors do not support the multiplication instruction in Galois Field \( GF(2^{128}) \).

In order to implement GHASH, several approaches that differ in both speed and security can be used. For the implementation of software cryptographic algorithms, it is important to provide some level of protection against side-channel attacks. In case of GHASH, the basic level of protection implies resistance to timing attacks as well as to simple attacks on energy consumption (simple power analysis), which is achievable by the lack of such loops, operations, and conditional branches in the program code, whose execution time depends on the value secret data. Such implementations are performed in a constant time [constant-time (CT)] regardless of input data, but at the expense of lowering the performance. The CT requirement is almost a mandatory parameter for cryptographic libraries, however, this option can be disabled if necessary. This will result in increasing efficiency, but at the same time, it will reduce the level of security, mainly because of the NCT of the GHASH operation.

In order to meet the conflicting requirements of maintaining both: safety and efficiency, the authors proposed their own solution, ensuring a quick carry-less multiplication at a constant time, therefore it is marked CTP (CT & Performance). Therefore, this article presents the results of research covering three different ways of implementing carry-less multiplication:

1) GHASH-NCT—fast but vulnerable to time attacks;
2) GHASH-CT—safe, but slower;
3) GHASH-CTP—safe and efficient.

In the case of GHASH-NCT, the direct method of Shift-XOR multiplication or the so-called Schoolbook method was used [29]. In this method, for every multiplication \( c = a \oplus b \), every bit \( b \) is extracted, and if it is equal to 1, the XOR operation is performed over the content of accumulator’s \( c \) and the value of the operand \( a \) shifted by \( i \) bits (\( a << i \)).

Algorithm

| Line | Description |
|------|-------------|
| 1    | \( Z \leftarrow 0, V \leftarrow D \) |
| 2    | for \( i = 0 \) to \( 127 \) do |
| 3    | if \( H_i = 1 \) then |
| 4    | \( Z \leftarrow Z \oplus V \) |
| 5    | end if |
| 6    | if \( V_{127} = 0 \) then |
| 7    | \( V \leftarrow \text{rightshift}(V) \) |
| 8    | else |
| 9    | \( V \leftarrow \text{rightshift}(V) \oplus R \) |
| 10   | end if |
| 11   | end for |
| 12   | return \( Z \) |

Fig. 4. Algorithm for calculating the value of \( Z = D \oplus H \mod g \), where \( D \), \( H \), and \( Z \in GF(2^{128}) \), \( R = 0xel \).

The main advantage of this method is its simplicity and the possibility of reducing the modulo of the polynomial \( g \) at the time of multiplication. As can be seen in lines 3 and 4 of the algorithm (Fig. 4), the multiplication time depends on the value of \( H_i \) bits of one of the operands.

In order to keep the Shift-XOR algorithm’ execution time fixed, the authors propose to replace conditional operators dependent on steps three and six with unconditional operations using masks. This variant of the GHASH-CT algorithm is more efficient in terms of execution time. The following fragment shows the basic idea of achieving a fixed execution time based on masks depending on the value of the least significant bit

\[
\begin{align*}
\text{NCT} & \quad \text{CT} \\
\text{if } (a \& 0x01) & \quad \text{mask} = 0x00 - (a \& 0x01) \\
{b = b \& c;} & \quad b = b \& (c \& \text{mask}).
\end{align*}
\]

Typical libraries that implement software multiplication take advantage of the fact that one of the multipliers of the hash function \( H \) has a constant value, which allows to build different (in size) look-up tables, depending on the desired speed [27]. The basic idea of these algorithms is to divide factor \( D \) with an NCT into parts \( s \) (usually 8 or 4 bits) and use them as indexes of tables with precalculated partial products \( s \oplus H \). The look-up tables and hashing key \( H \) are generated initially (during the offline stage). In the stage, the calculation of \( D \oplus H \) is replaced by searching in tables and executing the XOR logical operation of the read elements for forming the result.

The fastest variant of this method is to use 64 kilobytes to store 16 tables \( T_j \) (for each byte \( D \)), where each table contains 256 elements \( T_j[i] \) (all possible values of byte \( D \)). 128-bit values \( T_j[i] \) are calculated as follows:

\[
T_j[i] = (\text{Hash Key} \oplus (i << 2^k)) \mod g
\]

for \( j = 0, 1, \ldots, 255 \) and \( i = 0, 1, \ldots, 15 \). (9)

Multiplication in this variant requires 16 readings of 128-bit values from tables and 16 128-bit XOR operations.

It is possible to use smaller tables (8 kB or 4 kB), but this causes a significant reduction in the effectiveness of this method, as the number of operations of address calculations and memory readings increases.

In this article, a table-based approach was not considered, taking into account several issues.
1) The implementation of this method requires a large amount of RAM because the tables are calculated dynamically during the execution of the cryptographic algorithm. However, the demand for 64 KB of RAM is critical for embedded processors, especially, 8/16 bits.

2) Calculating the tables takes a lot of time, and extends the response time after replacing the key.

3) From points 1 and 2 it follows that for IoT applications that have several keys and these keys are often changed, this approach is impractical or even impossible.

In this article, for GHASH-CT, depending on the architecture and width of the processor data bus, three approaches are used that provide multiplication of $8 \times 8$, $16 \times 16$, and $32 \times 32$. Next, based on these operations, the result of multiplication $128 \times 128 \rightarrow 256$ is created hierarchically using the Karatsuba algorithm, which allows to reduce the number of multiplications in each level of the hierarchy from four to three.

The essence of new techniques for efficiently performing a constant time CTP multiplication operation in the Galois field is presented below.

A. AVR

To implement the basic function

$$c = a \otimes b (8 \times 8 \rightarrow 16)$$

the embedded hardware two-cycle multiplication unit (i.e., the MUL command) was used. The basic idea is to divide multipliers into smaller parts so that during the multiplication operation, carry bits would not affect other result bits. For this purpose, first we perform CT operand multiplications and the two least significant bits $b_1b_0$

$$c = ((0x00 - (b & 0x01)) & a) \land (a * (b & 0x02)). \quad (11)$$

The expression $0x00 - (b\&0x01)$ generates the value-mask $0x00$, if $b_0 = 0$, and $0xff$, if $b_0 = 1$. What results in

$$c = a \land (a * 2b_1), \quad b_0 = 1$$

and

$$c = a * 2b_1, \quad b_0 = 0. \quad (12)$$

Bits $b_1b_0$ will still not affect the result and will be masked during the subsequent calculations.

Then, four multiplications are performed, each of which has a maximum number of elements equal to 3, and the gap between them is 1 bit, which guarantees no distortion caused by carry (Fig. 5). The arithmetic sum of elements in the carry-excluding column is equal to their sum modulo 2, and carry never affects adjacent nonzero elements. Using the appropriate masks, proper bits are selected and combined into the final result.

As shown by the experiments, the presented method is faster than Shift-XOR and provides $8 \times 8$ multiplication in 41 cycles.

B. MSP430

Although these MCUs have a $16 \times 16 \rightarrow 32$ multiplication unit, it does not belong to the processor core and it works as a peripheral module. Accordingly, to access the input and output data and to control the operating mode, there are special registers that must be configured from the main program. This
requires a significant number of clock cycles and makes it unprofitable to use this unit for multiplication.

Therefore, in this article, the previously presented method was used to perform CTP multiplication. In particular, the value of \( a_i \) is bit-scanned using the word \((\text{mask} \& 0x01) = 0b0 \cdots 0b0a_i\). The word \( 0x00 \) — (mask&0x01) takes the values \( 0x0000 \) or \( 0xffff \), serving as the mask for the multiplier \( b \). The resulting partial product overlaps the result with the corresponding offset. High-level \( C \) code is translated by the compiler into compact deterministic assembler instructions, as shown in Fig. 6.

The 16\( \times \)16 multiplication takes 180 cycles.

The GCM algorithm adopted the big-endian format for the order of bits in bytes. This requires, during the GHASH operation, to reverse the order of bits in bytes. In the case of AVR- and MSP430-MCU, inversion is performed using an array method (256 \( \times \)8-bit array), and for ARM an intrinsic function is used to generate an RBIT instruction that performs a bit reverse in a 32-bit register.

In order to implement the modular reduction, the method described in [14] was used.

**IX. PERFORMANCE EVALUATION AND COMPARISON OF RESULTS**

Measured parameters after the implementation of the AES-GCM algorithm on different types of MCUs are summarized in Tables V–VII.

The results of the research show some trends depending on the size of the message and the width of the MCU data bus. The application of the CTP methods of multiplication in the AES-GCM algorithm, proposed by the authors, ensuring the maintenance of the assumed level of security, allowed to avoid significant efficiency losses compared to the NCT approach.

1) On an 8-bit AVR MCU, for small packet lengths (from 8 to 256 bytes), the encryption/decryption speed in the CTP mode gets closer to the NCT mode, and for longer lengths (from 512 to 2048 bytes) is even higher than in NCT. This requires an increase (about 12%) of the required ROM and an almost 1.4-fold increase in RAM.

2) On a 16-bit MSP430 MCU, for a packet length of 8 bytes, the encryption/decryption rate in the CTP mode is close to 20% lower than in the NCT mode, but as the packet length increases, the difference in speed gradually decreases. Implementation of the method requires

| Algorithm     | Encryption/decryption speed [cycles/Bytes] | Package length mlen [Bytes] | ROM [Bytes] | RAM [Bytes] |
|---------------|-------------------------------------------|-----------------------------|-------------|-------------|
|               | 8             | 16             | 32        | 64        | 128        | 256        | 512        | 1024       | 2048       |       |       |       |
| AES-GCM NCT   | 4322/         | 2100/         | 1413/     | 1076/     | 886/       | 802/       | 758/       | 733/       | 724/       | 9844  | 1038  |       |
|               | 4345          | 2107          | 1415      | 1077      | 887       | 803        | 759        | 733        | 724        |       |       |       |
| AES-GCM CT    | 9046          | 4529          | 3003      | 2240      | 1859      | 1668       | 1573       | 1525       | 1501       | 9642  | 1092  |       |
|               | 9007          | 4537          | 3007      | 2240      | 1859      | 1668       | 1573       | 1525       | 1501       |       |       |       |
| AES-GCM CTP   | 4358          | 2185          | 1440      | 1067      | 880       | 787        | 741        | 717        | 706        | 11012 | 1514  |       |
|               | 4381          | 2185          | 1444      | 1069      | 881       | 798        | 741        | 717        | 706        |       |       |       |
increasing the amount of ROM only by 4%, and RAM by 50%.

3) On the 32-bit ARM Cortex-M3 MCU, the encryption/decryption speeds in the CTP and NCT modes are comparable, but the size of the required memory decreases slightly—for ROM by 6% and for RAM by 4%. The decrease can be explained by the fact that the IT instruction allows a concise description of the low-level $32 \times 32$ multiplication, so to get the product using the Karatsuba algorithm, only one high-level $64 \times 64$ multiplication function call need to be used.

To evaluate the progress made, it is very interesting to compare the results obtained by the authors with the lightweight CAESAR finalists. Table VIII presents the comparison of the results of implementation of the best algorithms of the CEASAR competition (i.e., Ascon and ACORN).

For comparison with the results obtained in this article, the values presented in [30] and [31] were recalculated from the number of cycles $clk$ (needed to encrypt messages of a given length $m$), into the number of cycles per byte $cpb$, in a following way: $cpb = clk/m$. Also from all implementations presented in [30], we focused on the fastest one for each type of processor. The unmasked ACORN and Ascon implementations parameters were approximated in accordance with the result graphs presented in [31].

It can be seen that our implementation of AES-GCM CTP significantly wins in performance for small messages. As message size increases, AES-GCM CTP overtakes Ascon and loses to ACORN a bit. However, ACORN and Ascon generally require significantly less memory, especially ACORN. Therefore, for low-end processors compared to the optimized versions of AES-GCM, the ACORN and Ascon algorithms are actually light in terms of resources and to a lesser extent in terms of speed. However, according to the authors, there is certainly significant room to optimize the speed of ACORN and Ascon, for this class of processors.

### X. Summary

This article presented implementations of the AES-GCM authentication encryption algorithm, which were promising...
for IoT protocols/standards, using typical 8/16/32-bit MCUs. The main goal of the proposed methods was to achieve the maximum speed, as well as to ensure a constant time of the algorithm implementation during encryption and decryption as the basic factor in the protection against side-channel attacks. The presented results allow making a conscious choice of the proper authentication cipher based on the traffic analysis and available processor resources for a specific IoT application.

REFERENCES

[1] “Overview of the Internet of Things,” ITU, Geneva, Switzerland, ITU-Recommendation Y.2060, 2012.
[2] D. Evans. The Internet of Things: How the Next Evolution of the Internet is Changing Everything. Accessed: Jun. 1, 2019. [Online]. Available: https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411_FINAL.pdf
[3] A. Biryukov and L. Perrin, “State of the art in lightweight symmetric cryptography,” Cryptol. ePrint Archive, Rep. 2017/511, Accessed: Nov. 27, 2019. [Online]. Available: https://eprint.iacr.org/2017/511.pdf
[4] S. Pannek, S. Sinanovic, “Lightweight cryptography: Underlying principles and approaches,” Int. J. Comput. Theory Eng., vol. 3, no. 4, pp. 516–520, 2011.
[5] S. Gueron, Intel Advanced Encryption Standard (AES) Instructions Set. Accessed: Jun. 1, 2019. [Online]. Available: https://software.intel.com/sites/default/files/mcl/4/1/18/AES_WP_Rev_03_final_2010_01_26.pdf
[6] S. Gueron, AES-GCM for Efficient Authenticated Encryption Ending the Reign of HMAC-SHA-1. Accessed: Jun. 1, 2019. [Online]. Available: https://crypto.stanford.edu/RealWorldCrypto/slides/gueron.pdf
[7] Y. Sovyn, Y. Nakonechny, I. Opirsky, and M. Stakhiv, “Analysis of hardware support of cryptography in Internet of Things devices,” Sci. J. Inf. Security, vol. 24, no. 1, pp. 36–48, 2018.
[8] A. E. W. Eldewhari, T. M. H. Sharaf, A. A. Mansor, N. A. F. Mohamed, and S. M. H. Alsahabi, “SSL/TLS attacks: Analysis and evaluation,” in Proc. Int. Conf. Comput. Control Netw. Electron. Embedded Syst. (ICCNEEE), Khartoum, Sudan, 2015, pp. 203–208.
[9] P. Schaumont, “Security in the Internet of Things: A challenge of scale,” in Proc. Design Autom. Test Eur. Conf. Exhibit. (DATE), Lausanne, Switzerland, 2017, pp. 674–679.
[10] Y. Yang, L. Wu, G. Yin, L. Li, and H. Zhao, “A survey on security and privacy issues in Internet-of-Things,” IEEE Internet Things J., vol. 4, no. 5, pp. 1250–1258, Oct. 2017.
[11] M. Deo and S. Roy, “Recommendation for block cipher modes of operation: Galois/counter mode (GCM) for confidentiality and authentication,” document SP 800–38D, NIST, Gaithersburg, MA, USA, Nov. 2007.
[12] D. McGrew, “An interface and algorithms for authenticated encryption,” IETF, RFC 5116, 2008.
[13] Competition for Authenticated Encryption: Security, Applicability, and Robustness, CAESAR, Las Vegas, NV, USA, 2012.
[14] Intel Corporation. (2018). Intel Architecture Instruction Set Extensions and Future Features Programming Reference. Accessed: Jan. 1, 2019. [Online]. Available: https://software.intel.com/en-us/download/intel-architecture-instruction-set-extensions-and-future-features-programming-reference
[15] S. Gueron, “Intel advanced encryption standard (AES) new instructions set,” Santa Clara, CA, USA, Intel, White Paper, 2012.
[16] S. Gueron and M. E. Kounavis, “Intel carry-less multiplication instruction,” Santa Clara, CA, USA, Intel, White Paper, 2014.
[17] A. Fog. (2018). Instruction Tables. Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD and VIA CPUs. Accessed: Jun. 1, 2019. [Online]. Available: https://www.agner.org/optimize/instruction_tables.pdf
[18] S. Gueron, A. Langley, and Y. Lindell, “AES-GCM-SIV nonce misuse-resistant authenticated encryption,” Internet Res. Task Force, RFC 8452, Apr. 2019. Accessed: Nov. 28, 2019. [Online]. Available: http://www.ietf.org/rfc/rfc8452.html
[19] D. Câmara, C. Gouvêa, J. López, and R. Dahab, “Fast software polynomial multiplication on ARM processors using the NEON engine,” in Security Engineering and Information Informatics 2013 (LNCS 8128), A. Cuzzocrea, C. Kittl, D. E. Simos, E. Weippl, and L. X. Eus. Berlin, Germany: Springer, 2013, pp. 137–154. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-40588-4_10?ie=taes
[20] ARM Limited. (2017). ARM Architecture Reference Manual. ARMv8, for ARMv8-A Architecture Profile. Accessed: Jan. 1, 2019. [Online]. Available: https://static.docs.arm.com/ddi0487a/DDI0487E_a_armv8_arm.pdf

Yaroslav Sovyn was born in 1979. He received the M.Sc. degree in computer systems, automation and control and the Ph.D. degree in biological and medical systems from Lviv Polytechnic National University, Lviv, Ukraine, in 2001 and 2007, respectively. He currently employed as an Assistant Professor with the Department of Information Security, Lviv Polytechnic National University. His scientific interests include: security of embedded systems, hardware cryptography, light cryptography, effective implementation of cryptographic algorithms in embedded systems and IoT with increased resistance to physical attacks and side channel attacks.

Volodymyr Khoma was born in 1959. He received the M.Sc. degree in automation and telemechanics, the Ph.D. and the Habilitation degrees in electrical and electronic engineering from Lviv Polytechnic National University, Lviv, Ukraine, in 1981, 1990, and 2001, respectively. He received the Title of Professor in the field of control engineering in 2003. He currently employed as a Professor with the Institute of Control Engineering, Opole University of Technology, Opole, Poland, and the Department of Information Security, Lviv Polytechnic National University. His scientific interests include: principles and methods of using artificial intelligence in cybersecurity, cyber-security of embedded systems and IoT, methods and algorithms of digital measurement and signal processing.

Michal Podpora was born in 1979. He received the B.Sc. and M.Sc. degrees in computer engineering and the Ph.D. degree in control engineering and robotics from the Opole University of Technology, Opole, Poland, in 2002, 2004, and 2012, respectively. Since 2010, he has been a research department manager and a project manager of research projects for industry. He currently employed as a Humanoid Robot Developer with Research Department, Weegree, Opole, and as an Assistant Professor with the Institute of Computer Science, Opole University of Technology. His main research interests include artificial intelligence in robotics, machine vision, cognitive systems, robotic systems, smart infrastructure, embedded systems, IoT, cybersecurity, and forensic science.