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).

Manuscript received June 9, 2019; revised August 29, 2019 and October 12, 2019; accepted November 2, 2019. Date of publication November 13, 2019; date of current version January 10, 2020. *(Corresponding author: Michal Podpora.)*

Y. Sovyn is with the Institute of Computer Technologies, Automation and Metrology, Lviv Polytechnic National University, 79000 Lviv, Ukraine.

V. Khoma is with the Institute of Computer Technologies, Automation and Metrology, Lviv Polytechnic National University, 79000 Lviv, Ukraine, and also with the Faculty of Electrical Engineering, Automatic Control and Informatics, Opole University of Technology, 45-271 Opole, Poland.

M. Podpora is with the Faculty of Electrical Engineering, Automatic Control and Informatics, Opole University of Technology, 45-271 Opole, Poland (e-mail: m.podpora@po.opole.pl).

Digital Object Identifier 10.1109/JIOT.2019.2953230

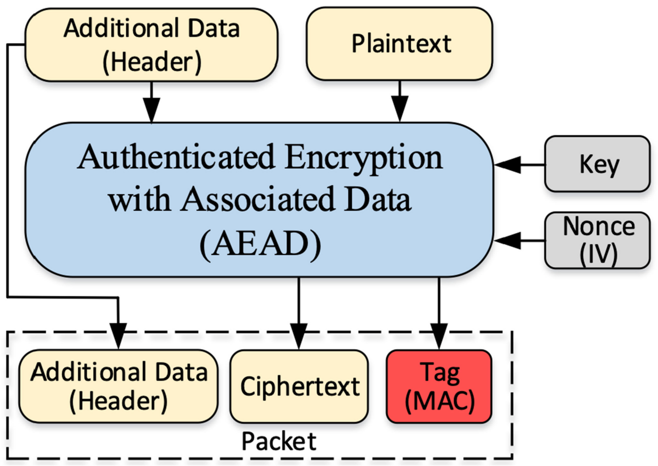


Fig. 1. Principle of operation of AEAD algorithms.

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.

TABLE I
Evolution of Cryptographic Instructions in Intel and AMD Processors Generations [17]

		PCLMULQDQ		AESENC/AESDEC	
Year	CPU	Latency	Reciprocal throughput	Latency	Reciprocal throughput
2010	Intel Westmere	12	8	5	2
2011	Intel Sandy Bridge	14	8	8	4
2012	Intel Ivy Bridge	14	8	4	1
2013	Intel Haswell	7	2	7	1
2014	Intel Broadwell	5	1	7	1
2015	Intel Skylake	7	1	4	1
2011	AMD Bulldozer	12	7	5	2
2012	AMD Piledriver	12	7	5	2
2014	AMD Steamroller	11	7	5	1
2013	AMD Jaguar	3	1	5	1
2017	AMD Ryzen	4	2	4	0.5

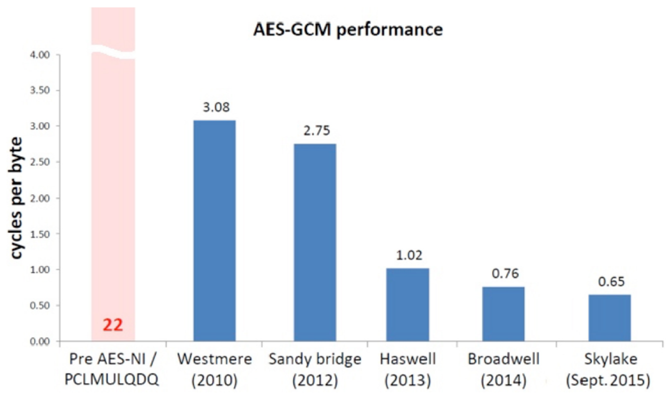


Fig. 2. Influence of the evolution of cryptographic instructions in the Intel and AMD processors generations on AES-GCM performance [18].

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 8×8 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 128×128 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 64×64 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.

TABLE II
PARAMETERS FOR THE SOFTWARE IMPLEMENTATION OF THE AES 128-GCM ON MCUS

CPU model	MSP430 (16 bit), [23]	ARM Cortex-M3 (32 bit), [24]
Encr./Decr. [CPB]	863/862	2769/-
ROM/RAM [Bytes]	7166/884	2644/812
Plaintext/Header [Bytes]	16/0	16/16

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) containing a 32-bit result of the *SubBytes*(), *ShiftRows*(), and *MixColumns*() operations for a given byte of the state array. Thus, AES encryption consists of finding the right value in the table and adding modulo 2 to the round key.

At present, there are almost no publications on how to efficiently implement AEAD ciphers for low-end MCU-based embedded systems. Only articles [23], [24] present the results of research on the impact of architecture and methods for the optimization of calculations in the AES-GCM algorithm (Table II).

The parameters given in Table II concern:

1) row 1—efficiency of encryption/decryption (in cycles per byte–cpb);
2) row 2—size of the memory required to perform calculations;
3) row 3—size of the encrypted and authenticated data.

In [23], it was shown that the use of the built-in accelerator allowed to increase the AES-128 encryption rate by 2.2 times, however, the estimation of the number of cycles did not take into account the operation of generating key-dependent variables. Data used in [24] were taken from the Cifra cryptographic primitive library [25], created for embedded systems, so its priorities include simplicity, moderate requirements for code size and data size, and resistance to side-channel attacks, which results in lowering speed.

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 (2^{128}) 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, postincrementation, 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 III
BASIC PARAMETERS OF THE MCUS TESTED

MCU	Data bus width [bits]	General registers count	Multiplication unit
AVR	8	32	8 × 8 (core)
MSP 430	16	12	16 × 16 (module)
ARM Cortex M3	32	13	32 × 32 (core)

TABLE IV
COMPLEXITY OF LOGICAL AND BIT OPERATIONS (>> AND << OPERATORS—LEFT AND RIGHT SHIFT, >>> AND <<< OPERATORS—LEFT AND RIGHT ROTATION, ...—CYCLIC SHIFT)

MCU	Logical and bitwise operations (cycles per operation)
AVR	AND (1), OR (1), XOR (1), NOT (1), >> 1 (1), 1 << (1), >>>1 (1), <<< 1 (1)
MSP 430	AND (1), OR (1), XOR (1), NOT (1), >> 1 (1), 1 << (1), >>>1 (1), <<< 1 (1)
ARM Cortex M3	AND (1), OR (1), XOR (1), NOT (1) >> 1...32 (1), 1...31 << (1), >>>1...31 (1)

twelve (R4-R15) are general-purpose registers. Registers R0-R3 perform special functions (Program Counter, Stack Pointer, Status Register, and Constant Generator). The instruction set is very simple and contains 27 original and 24 emulated instructions that are optimized for the efficient use of high-level programming languages. All commands are 16-bit and can support both 8-bit and 16-bit operands. There is also a 16x16 multiplication module. Seven addressing modes are supported. The number of cycles to perform an instruction depends on the command format and the addressing mode and can range from 1 to 6 [27].

Due to the orthogonal architecture and registry operations performed in one cycle, high efficiency and code density are ensured. From the point of view of cryptography, an important feature of the MSP430 processor is the direct exchange of data between memory cells (omitting registers).

C. ARM Cortex-M3 Microcontrollers (32 Bits)

ARM cores dominate the 32-bit RISC MCU market and are currently approaching 8-bit models for power consumption and price, making them a serious competitor in their traditional applications. Therefore, the research on the implementation of the AES-GCM algorithm was carried out using the ARM Cortex-M3 MCU. It is a 32-bit processor based on Harvard architecture with a three-stage pipeline providing support for the Thumb and Thumb-2 instruction set.

The Cortex-M3 core contains 16 registers R0–R15, of which the R0–R12 registers are general-purpose registers. The ALU 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. A one-cycle 32 × 32 multiply unit is also available [28].

Tables III and IV show the basic (in the context of cryptographic calculations) properties of embedded processors.

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

$$GCM\text{-}AE_K(IV, A, P) = (C, T) \tag{1}$$

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

$$GCM\text{-}AD_K(IV, A, C, T) = (P \text{ or FAIL}) \tag{2}$$

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 $P1, P2, \ldots, Pn$ of plaintext (Fig. 3).

To generate a key stream, $Y0, Y1, Y2, \ldots, Yn$ data blocks (parameterized with the secret key K) are passed to the input of the AES algorithm. The first block $Y0$ 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

$$Y0 = CTR = IV \| 0^{31} 1. \tag{3}$$

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_H ({}, IV) 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.

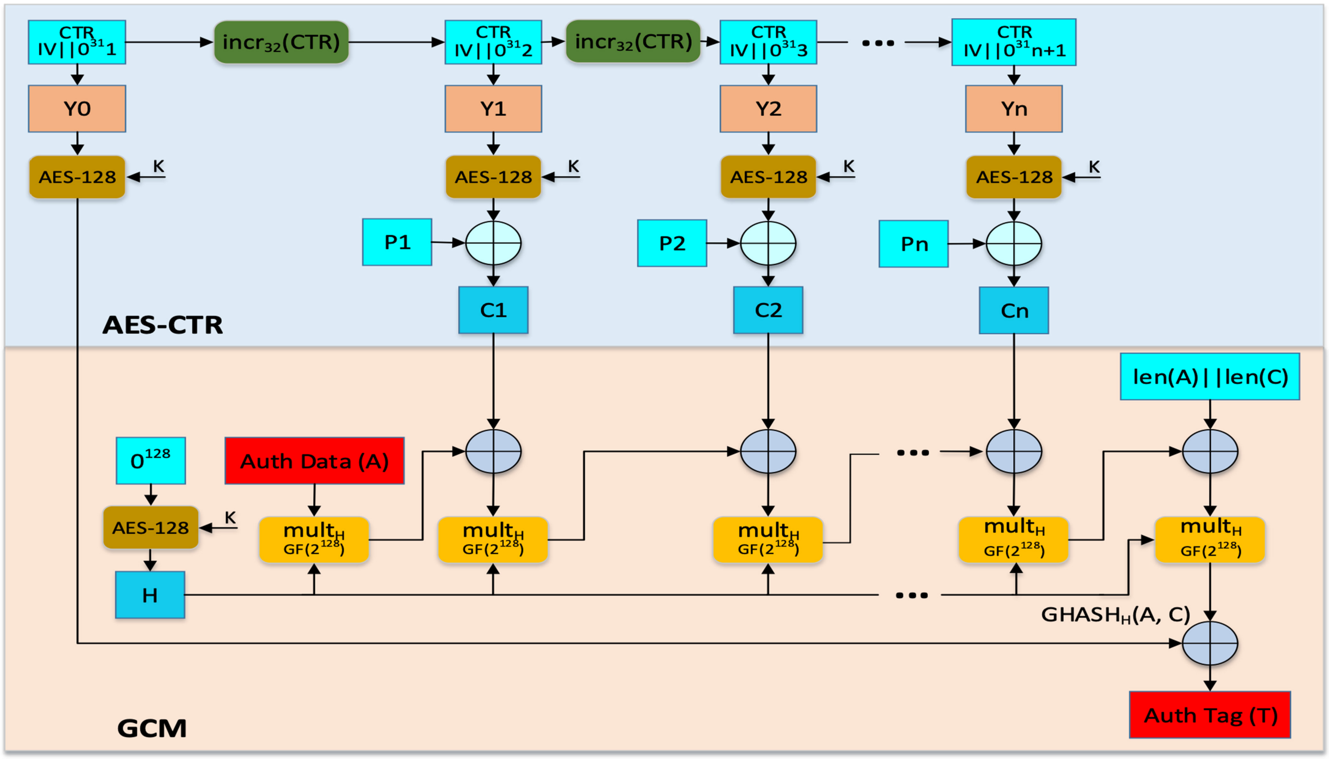


Fig. 3. Structure of the GCM-AE$_K$ algorithm $(IV, A, P) = (C, T)$.

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

$$g = x^{128} + x^7 + x^2 + x^1 + 1. \tag{4}$$

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:*

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

2) *Reduction:*

$$\{Z\} \bmod \left(x^{128} + x^7 + x^2 + x^1 + 1\right), \text{ (256 bits} \rightarrow \text{128 bits).} \tag{6}$$

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

$$H = \text{AES}_K\left(0^{128}\right). \tag{7}$$

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

$$T = \text{GHASH}_H(A, C, \text{len}(A)_{64} \| \text{len}(C)_{64}) \oplus \text{AES}_K(Y_0). \tag{8}$$

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 × 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, $T0[0] = 0xc66363a5$, where $0x63 = 0xc6 \otimes 0xa5$. 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, $T0-T3$ tables elements with converged indexes contain equal bytes, but in a different order ($T0, [0] = 0xc66363a5, T1[0] = 0xa5c66363$, etc.). This is also taken into account during the software byte reading, so only one table *T0* 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 $c = a \otimes b$, 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
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          $V \leftarrow$ rightshift($V$)
8      else
9          $V \leftarrow$ rightshift($V$)$\oplus R$
10     end if
11 end for
12 return $Z$
```

Fig. 4. Algorithm for calculating the value of $Z = D \otimes H \bmod g$, where D, H, and $Z \in GF(2^{128})$, $R = 0xe1$.

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

NCT	CT
if (a & 0x01)	mask = 0x00 − (a & 0x01)
{$b = b\ c$; }	$b = b$ (c & mask).

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 $s \otimes H$. The look-up tables and hashing key H are generated initially (during the offline stage). In the stage, the calculation of $D \otimes H$ 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_i (for each byte D), where each table contains 256 elements $T_i[j]$ (all possible values of byte D). 128-bit values $T_i[j]$ are calculated as follows:

$$T_i[j] = \left(\text{Hash Key} \otimes \left(j << 2^{8i}\right)\right) \bmod g \quad \text{for } j = 0, 1, \ldots, 255 \text{ and } i = 0, 1, \ldots, 15. \tag{9}$$

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.

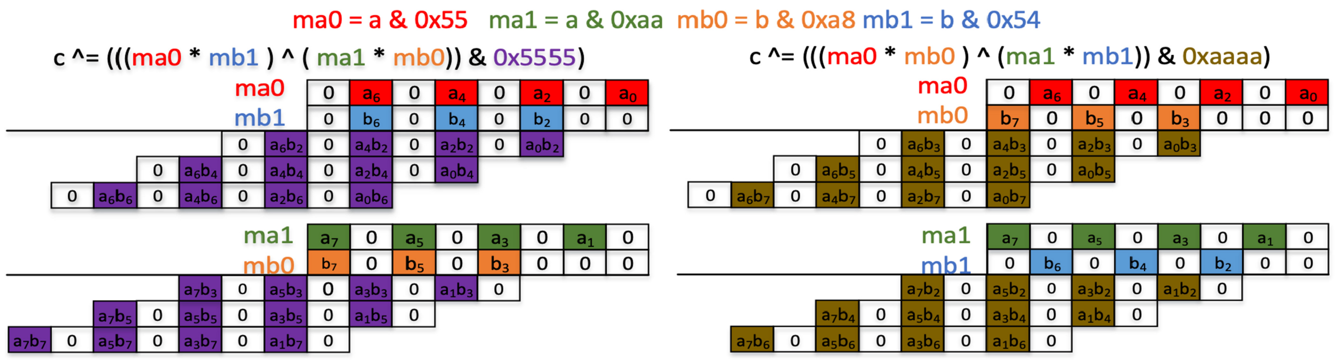
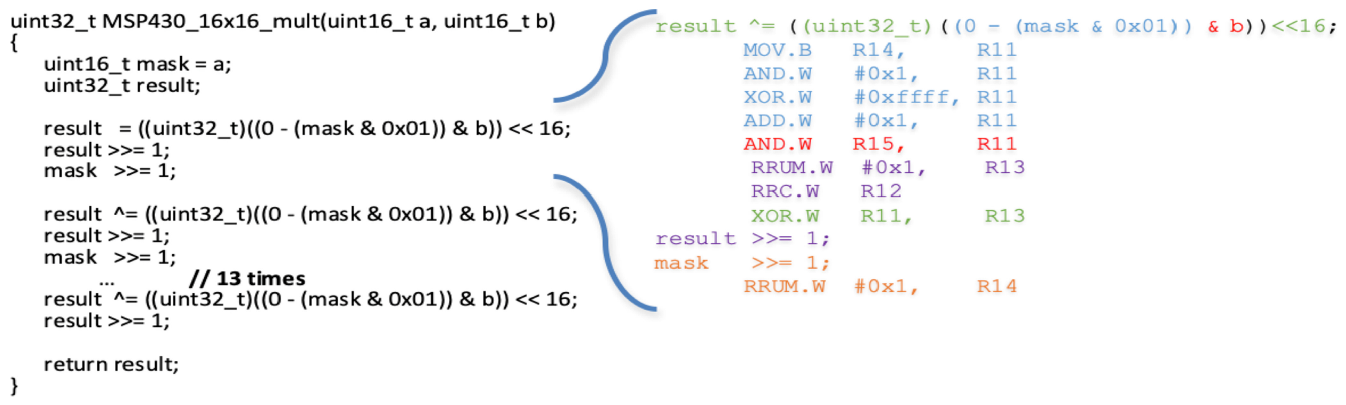

ma0 = a & 0x55   ma1 = a & 0xaa   mb0 = b & 0xa8   mb1 = b & 0x54
c ^= (((ma0 * mb1) ^ (ma1 * mb0)) & 0x5555)
c ^= (((ma0 * mb0) ^ (ma1 * mb1)) & 0xaaaa)
ma0
mb1
mb0
ma1

Fig. 5. Process of performing the 8 × 8 CTP-multiplication operation.



```
uint32_t MSP430_16x16_mult(uint16_t a, uint16_t b)
{
    uint16_t mask = a;
    uint32_t result;

    result  = ((uint32_t)((0 - (mask & 0x01)) & b)) << 16;
    result >>= 1;
    mask   >>= 1;

    result  ^= ((uint32_t)((0 - (mask & 0x01)) & b)) << 16;
    result >>= 1;
    mask   >>= 1;
        ...        // 13 times
    result  ^= ((uint32_t)((0 - (mask & 0x01)) & b)) << 16;
    result >>= 1;

    return result;
}
```

```
result ^= ((uint32_t)((0 - (mask & 0x01)) & b))<<16;
    MOV.B   R14,     R11
    AND.W   #0x1,    R11
    XOR.W   #0xffff, R11
    ADD.W   #0x1,    R11
    AND.W   R15,     R11
     RRUM.W  #0x1,    R13
     RRC.W   R12
     XOR.W   R11,     R13
result >>= 1;
mask   >>= 1;
    RRUM.W  #0x1,    R14
```

Fig. 6. Process of performing the 16×16 CTP-multiplication operation.

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 × 8, 16 × 16, and 32 × 32. Next, based on these operations, the result of multiplication 128 × 128 → 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

$$c = a \otimes b(8 \times 8 \rightarrow 16) \tag{10}$$

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

$$c = ((0x00 - (b \ \& \ 0x01)) \ \& \ a) \wedge (a * (b \ \& \ 0x02)). \tag{11}$$

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

$$c = a \wedge (a * 2b_1), \ \text{ if } b_0 = 1$$
$$\text{and } c = a * 2b_1, \ \text{ if } b_0 = 0. \tag{12}$$

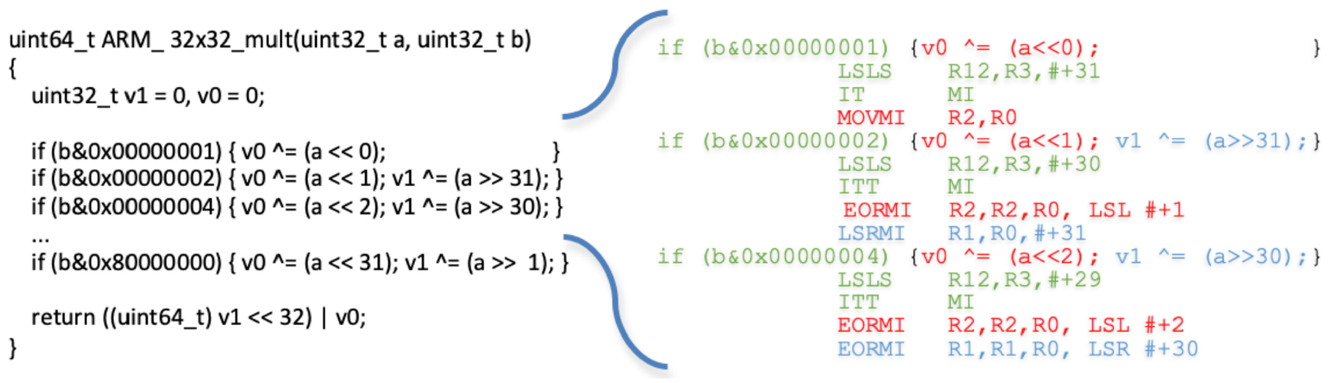
Bits b_1b_0 will still not affect the result and will be masked during the subsequent calculations.

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 8×8 multiplication in 41 cycles.

B. MSP430

Although these MCUs have a 16 × 16 → 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



```
uint64_t ARM_ 32x32_mult(uint32_t a, uint32_t b)
{
  uint32_t v1 = 0, v0 = 0;

  if (b&0x00000001) { v0 ^= (a << 0);                  }
  if (b&0x00000002) { v0 ^= (a << 1); v1 ^= (a >> 31); }
  if (b&0x00000004) { v0 ^= (a << 2); v1 ^= (a >> 30); }
  ...
  if (b&0x80000000) { v0 ^= (a << 31); v1 ^= (a >>  1); }

  return ((uint64_t) v1 << 32) | v0;
}
```

```
if (b&0x00000001) {v0 ^= (a<<0);                }
          LSLS     R12,R3,#+31
          IT       MI
          MOVMI    R2,R0
if (b&0x00000002) {v0 ^= (a<<1); v1 ^= (a>>31);}
          LSLS     R12,R3,#+30
          ITT      MI
          EORMI    R2,R2,R0, LSL #+1
          LSRMI    R1,R0,#+31
if (b&0x00000004) {v0 ^= (a<<2); v1 ^= (a>>30);}
          LSLS     R12,R3,#+29
          ITT      MI
          EORMI    R2,R2,R0, LSL #+2
          EORMI    R1,R1,R0, LSR #+30
```

Fig. 7. Process of performing the 32 × 32 CTP-multiplication operation.

TABLE V
Parameters of the AES-GCM Algorithm on an 8-bit AVR MCU (Length of Attached AAD Data = 16 bytes)

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/ 4345	2100/ 2107	1411/ 1415	1073/ 1075	886/ 887	802/ 803	758/ 759	733/ 733	724/ 724	9844	1083
AES-GCM CT	9046/ 9069	4529/ 4537	3003/ 3007	2240/ 2242	1859/ 1860	1668/ 1669	1573/ 1573	1525/ 1525	1501/ 1501	9642	1092
AES-GCM CTP	4358/ 4381	2185/ 2193	1440/ 1444	1067/ 1069	880/ 881	787/ 788	741/ 741	717/ 717	706/ 706	11012	1514

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 (mask & $0x01$) = $0b0 \cdots 00a_i$. 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 16×16 multiplication takes 180 cycles.

C. ARM Cortex-M3

The Cortex-M3 core is equipped with a $32 \times 32 \rightarrow 64$ one-cycle multiplication unit; nevertheless, the Shift-XOR method proves its effectiveness due to the ability to combine the shift of one of the operands (by a certain number of bits) and XOR operations in one instruction. Although the C language Shift-XOR method contains conditional operators (therefore the computation time is variable), but the machine code generated by the compiler is a constant time code. This is achieved by using the *IT* (IF-THEN) compiler command, which allows conditional execution of a small code fragment (up to 4 instructions). At the same time, there is no drop in productivity due to the pipeline stall, because the order in which the code is run does not change. Depending on the condition, instructions following *IT* are performed or not (they do not change the state of registers and flags), but in both cases, they pass through the pipeline. This ensures a fixed execution time of the code. Fig. 7 shows the *C* code and its mapping to assembler, showing the lack of conditional transitions.

The 32 × 32 multiplication takes 140 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 × 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

TABLE VI
PARAMETERS OF THE AES-GCM ALGORITHM ON A 16-bit MSP430 MCU (LENGTH OF ATTACHED AAD DATA = 16 bytes)

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	3058/ 3062	1492/ 1496	994/ 996	749/ 750	616/ 616	555/ 555	523/ 523	506/ 506	499/ 499	11134	350
AES-GCM CT	5765/ 5769	2882/ 2886	1906/ 1908	1417/ 1418	1173/ 1174	1051/ 1051	990/ 990	960/ 960	944/ 944	11630	360
AES-GCM CTP	3638/ 3642	1819/ 1823	1195/ 1197	883/ 884	728/ 728	650/ 650	611/ 611	591/ 591	581/ 581	11568	522

TABLE VII
PARAMETERS OF THE AES-GCM ALGORITHM ON THE 32-bit ARM CORTEX-M3 MCU (LENGTH OF ATTACHED AAD DATA = 16 bytes)

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	981/ 985	486/ 488	318/ 319	234/ 234	192/ 192	171/ 171	160/ 160	155/ 155	153/ 153	9738	452
AES-GCM CT	1535/ 1540	763/ 765	503/ 504	373/ 373	307/ 308	275/ 275	259/ 259	250/ 250	246/ 246	10508	372
AES-GCM CTP	1015/ 1020	503/ 505	329/ 330	243/ 243	199/ 199	177/ 177	166/ 167	161/ 161	158/ 158	9108	432

TABLE VIII
COMPARISON OF THE OBTAINED RESULTS, WITH THE LIGHTWEIGHT FINALISTS OF THE CAESAR COMPETITION
(LENGTH OF ATTACHED AAD DATA = 16 bytes)

Algorithm	CPU	Encryption/decryption speed [cycles/Bytes]		ROM [Bytes]	RAM [Bytes]
		m = 16	m = 256		
ACORN-128 [30]		17999	-	3700	263
Ascon-128 [30]	AVR	11941	-	6140	268
AES-GCM CTP [this work]		2185/2193	-	11012	1514
ACORN-128 [30]		23856	-	2326	218
Ascon-128 [30]	MSP430	34005	-	8358	290
AES-GCM CTP [this work]		1819/1823	-	11568	522
ACORN-128 [30]		2806	-	2364	344
Ascon-128 [30]		639	-	16072	240
ACORN-128 [31]	ARM Cortex-M3	≈1500	≈156	≈1700	-
Ascon-128 [31]		≈875	≈238	≈1900	-
AES-GCM CTP [this work]		503/505	177	9108	432

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 32×32 multiplication, so to get the product using the Karatsuba algorithm, only one high-level 64×64 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_0411FINAL.pdf

[3] A. Biryukov and L. Perrin, "State of the art in lightweight symmetric cryptography," Cryptol. ePrint Archive, Rep. 2017/511, 2017. Accessed: Nov. 27, 2019. [Online]. Available: https://eprint.iacr.org/2017/511.pdf

[4] S. Panasenko and S. Smagin, "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/m/d/4/1/d/8/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. Opirskyy, 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. Eldewahi, T. M. H. Sharf, A. A. Mansor, N. A. F. Mohamed, and S. M. H. Alwahbani, "SSL/TLS attacks: Analysis and evaluation," in *Proc. Int. Conf. Comput. Control Netw. Electron. Embedded Syst. Eng. (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. Dworkin, "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 and its usage for computing the GCM mode," 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.hjp.at/doc/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 Intelligence Informatics 2013* (LNCS 8128), A. Cuzzocrea, C. Kittl, D. E. Simos, E. Weippl, and L. Xu, Eds. Berlin, Germany: Springer, 2013, pp. 137–154. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-40588-4_10#citeas

[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/ddi0487/ea/DDI0487E_a_armv8_arm.pdf

[21] C. Gouvêa and J. López, "Implementing GCM on ARMv8," in *Topics in Cryptology, CT-RSA 2015* (LNCS 9048), K. Nyberg, Ed. Cham, Switzerland: Springer, 2015, pp. 167–180. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-16715-2_9#citeas

[22] J. Daemen and V. Rijmen, *The Design of Rijndael*. New York, NY, USA: Springer-Verlag, 2002.

[23] C. P. L. Gouvea and J. Lopez, "High speed implementation of authenticated encryption for the MSP430X microcontroller," in *Progress in Cryptology LATINCRYPT 2012* (LNCS 7533), A. Hevia and G. Neven, Eds, Heidelberg, Germany: Springer, 2012, pp. 288–304. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-16715-2_9#citeas

[24] F. De Santis, A. Schauer, and G. Sigl, "ChaCha20-poly1305 authenticated encryption for high-speed embedded IoT applications," in *Proc. Design Autom. Test Eur. Conf. Exhibit. (DATE)*, Lausanne, Switzerland, 2017, pp. 692–697.

[25] (Feb. 2017). *The Cifra Project. A Collection of Cryptographic Primitives Targeted at Embedded Use*. [Online]. Available: https://github.com/ctz/cifra

[26] *8-Bit AVR Microcontroller With 8/16K Bytes of ISP Flash and USB Controller*, Atmel Corporat., San Jose, CA, USA, 2008.

[27] *User's Guide. MSP430FR58xx/59xx/68xx, and MSP430FR69xx Family*, Texas Instrum., Dallas, TX, USA, 2015.

[28] *ARM and Thumb-2 Instruction Set*, ARM, Cambridge, U.K., 2016.

[29] D. A. McGrew and J. Viega, *The Galois/Counter Mode of Operation (GCM)*, NIST, Gaithersburg, MA, USA, 2005.

[30] A. Adomnicai *et al.*, "Lilliput-AE: A new lightweight tweakable block cipher for authenticated encryption with associated data," NIST, Gaithersburg, MA, USA, 2019.

[31] A. Adomnicai, J. Fournier, and L. Masson, "Masking the lightweight authenticated ciphers ACORN and Ascon in software," Cryptol. ePrint Archive, Rep. 2018/708, 2018. Accessed: Nov. 27, 2019. [Online]. Available: https://eprint.iacr.org/2018/708.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, cybersecurity 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.