Low Power Link Layer Security for IoT: Implementation and Performance Analysis

Diego Altolini*, Vishwas Lakkundi†, Nicola Bui*†, Cristiano Tapparello‡ and Michele Rossi*‡
*Consorzio Ferrara Ricerche, Ferrara, Italy, †Patavina Technologies, Padova, Italy, ‡DEI, University of Padova, Italy
Email: diego.altolini@gmail.com, vishwaskl@ieee.org, nicolabui@gmail.com, {tappare1, rossi}@dei.unipd.it

***Abstract*—In this paper, we present the implementation and performance evaluation of security functionalities at the link layer of IEEE 802.15.4-compliant IoT devices. Specifically, we implement the required encryption and authentication mechanisms entirely in software and as well exploit the hardware ciphers that are made available by our IoT platform. Moreover, we present quantitative results on the memory footprint, the execution time and the energy consumption of selected implementation modes and discuss some relevant tradeoffs. As expected, we find that hardware-based implementations are not only much faster, leading to latencies shorter than two orders of magnitude compared to software-based security suites, but also provide substantial savings in terms of ROM memory occupation, i.e. up to six times, and energy consumption. Furthermore, the addition of hardware-based security support at the link layer only marginally impacts the network lifetime metric, leading to worst-case reductions of just 2% compared to the case where no security is employed. This is due to the fact that energy consumption is dominated by other factors, including the transmission and reception of data packets and the control traffic that is required to maintain the network structures for routing and data collection. On the other hand, entirely software-based implementations are to be avoided as the network lifetime reduction in this case can be as high as 25%.**

***Index Terms*—Information Security; Data Link Layer Security; AES-CCM; Internet of Things; M2M Communication; Performance Analysis.**

I. Introduction

The concept of Internet of Things (IoT) [1] has been proposed and studied as a means to provide wireless communication functionalities to different types of physical objects that support our daily activities. Due to the pervasive nature of these objects, sensitive data can be collected and transmitted for both public and private use from different sources. Consequently, integrity and confidentiality of transmitted data as well as the authentication of the elements involved in the communications are crucial.

There is an extensive literature that presents different solutions to provide security functionality to wireless data networks [2], but the inherently limited processing and communication capabilities of IoT devices prevent the use of full-fledged security suites. IoT devices are resource constrained and, in turn, we either need to craft dedicated security mechanisms or adapt existing schemes.

Security is a vast research topic, covering pretty much all layers of the protocol stack. Security requirements range from confidentially (information can only be understood by the intended receiver), integrity (a message will not be corrupted along the path connecting its source to its destination), authentication (the identity of the sender is certified and verified at the receiver), non-repudiation (the sender of a message cannot later deny having sent the message) and robustness against various types of attacks (man in the middle, replay attacks, denial-of-service, etc.) [3]. Several approaches are currently being investigated to provide security at the network and transport layers through lightweight key exchange techniques and simplified cipher suite negotiation procedures, as illustrated in [4, 5]. Other contributions focus on lightweight crypto-primitives, such as Elliptic Cryptography [6]. A substantial amount of work has been done to devise effective key management protocols as explained in [3] and the references therein.

In this paper, we focus on data link layer security and specifically on the security mechanisms that are specified by the IEEE 802.15.4 MAC standard. Hence, we explore different implementation modes, studying the impact of implementing encryption and authentication primitives in software or exploiting the hardware ciphers that are offered by most contemporary IoT platforms. We carry out an experimental campaign focusing on performance metrics such as latency, memory occupation and energy consumption and discuss relevant tradeoffs. Our results indicate that hardware-based implementations lead to latencies shorter than two orders of magnitude compared to software-based security suites, at the same time providing substantial savings in terms of ROM memory occupation (up to six times) and reduced energy consumption. Remarkably, the addition of hardware-based security support at the link layer only marginally impacts the network lifetime metric, leading to worst-case reductions of just 2% (relative to the case where no security is accounted for at the link layer). This is probably the main result of this paper and also basically tells us that hardware-based implementations of link layer security are a viable approach. Software-based implementations are rather less attractive as the network lifetime reduction in this case can be as high as 25%.

This paper is organized as follows: our protocol stack architecture, link layer security aspects, pertinent authentication and encryption algorithms are described in Section II; Section III explains how the security block is implemented within such architecture, defines its structure, specifies the algorithms employed and also describes the experimental setup used during the trials. Section IV depicts the results obtained and provides their performance analysis and finally and conclusions are drawn in Section V.

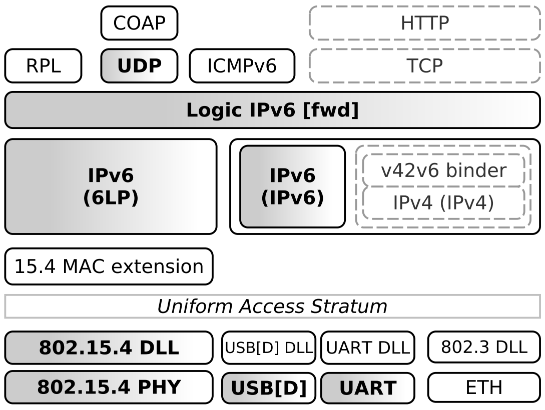


Figure 1: Communication stack architecture.

II. MAC Layer Security

A. Protocol stack architecture and link layer procedures

The architecture of our communication stack is depicted in Fig. 1. It illustrates the implementation of an IPv6/6LoWPAN stack [7, 8] running on IEEE 802.15.4 radio technology. This stack provides an abstraction of all the existing physical communication interfaces of a sensor node and allows seamless data transmission, reception and forwarding. In our implementation, the shaded modules are mandatory to enable basic connectivity (providing IPv6 support, physical and link layer technology), while the unshaded modules are optional, and provide more advanced functionalities such as CoAP [9] to handle application layer messages and RPL [10] to permit routing of packets over multi-hop low power and lossy IPv6 networks. The modules with dashed borders, instead, represent the logical components of the communication stack that provide the basic gateway functionalities and currently reside outside the node. These are needed at the border gateway, which connects the wireless sensor network (WSN) domain to the Internet. Further details on IEEE 802.15.4 data link (DLL) and physical (PHY) layers are available in the IEEE 802.15.4 standard [11]. For more details on our protocol architecture, see [5, 12].

At the DLL, we have implemented the Low Power Listening (LPL) protocol [13, 14] that implements a medium access control (MAC) strategy specifically designed to enable longer battery lifetimes of WSN nodes. The main idea is to let the network nodes sleep for most of the time and wake them up periodically for only a fraction of the sleep period. During this time, nodes sense the channel to detect data transmissions. If a transmission is detected, the node continues to receive the whole data packet, otherwise it switches back to the sleep mode. On the other hand, the transmitting node, in order to maximize the probability of being heard by the addressed L2 destination, repeats the transmission of the packet for the whole duration of the sleep period. This protocol provides great advantages especially when the transmissions are very sporadic. The LPL protocol takes care of all the major MAC functionalities including transmission and reception of frames, radio power efficiency control, CSMA-CA channel access management, backoff congestion control and retransmissions, duplicate filtering, broadcast support, neighbor cache maintenance and so on. The LPL layer implementation resides between the 6LoWPAN adaptation layer and the IEEE 802.15.4 PHY layer.

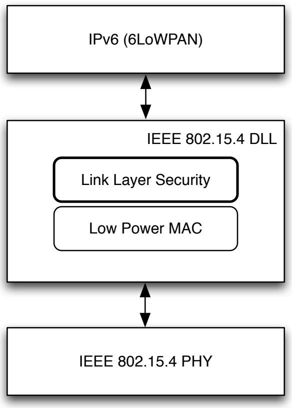


Figure 2: Link layer security module.

Our link layer security module is implemented within the LPL block as shown in Fig. 2. Further details are given below in Section III.

B. IEEE 802.15.4 MAC security description

The IEEE 802.15.4 specification [11] defines the PHY and MAC layers for low rate wireless personal area networks (LR-WPAN), that is used for short range communications and requires low power consumption and low cost devices. Because of their nature, there are additional constraints that make it difficult to secure these networks. In particular, the choices related to cryptographic algorithms are strongly influenced by the limited capabilities available on IoT nodes.

This specification provides some security services as *data confidentiality*, *data authenticity* and *protection against replay attacks*. Data confidentiality is required in order to protect the information included in the data frames from unauthorized access and is achieved through message encryption, whereas data authenticity is used to prevent unauthorized content changes and to verify the identity of the source of the transmitted information at the receiver. Replay protection, when required, detects the fraudulent repetition of messages to avoid replay attacks and is obtained through a frame counter, which is incremented each time a secure frame is transmitted until it reaches a given threshold. Obviously, as security is not mandatory, the unsecured mode is also supported.

For secured mode, the IEEE 802.15.4 standard makes use of symmetric-key cryptography with keys provided by the higher layers[1], combined together with some operation modes to achieve both frame encryption and integrity. While the former is accomplished by using a symmetric block cipher, such as the Advanced Encryption Standard (AES), the latter is fulfilled through a message authentication code (MAC), also called

[1]To this end, pre-shared keys may be used or the network administrator should rely on a key distribution mechanism. The investigation of these matters is outside the scope of this paper.

authentication tag or message integrity code (MIC), that is a variable-length tag used to authenticate and to provide integrity assurance on the message. Counter (CTR) mode, cipher block chaining with message authentication code (CBC-MAC) mode and counter with CBC-MAC (CCM) mode are the supported modes of operation. When both encryption and authentication are provided, the operation mode used is CCM; otherwise, when only one is required, CTR and CBC-MAC are used for providing encryption and authentication only, respectively. AES is the block cipher used for all the operation modes. AES, CTR, CBC-MAC and CCM are explained in greater details in the following sections.

The IEEE 802.15.4 specification allows flexibility for security support providing optional data confidentiality, and different levels of data authenticity in order to minimize the total overhead. The security options and the corresponding services offered by the specification, together with the authentication tag length, are shown in Table I.

TABLE I
SECURITY OPTIONS AND SERVICES PROVIDED BY THE IEEE 802.15.4 SPEC.

Security name	Confidentiality	Authenticity	Auth. tag (bits)
None	✗	✗	0
MIC-32	✗	✓	32
MIC-64	✗	✓	64
MIC-128	✗	✓	128
ENC	✓	✗	0
ENC-MIC-32	✓	✓	32
ENC-MIC-64	✓	✓	64
ENC-MIC-128	✓	✓	128

C. Advanced Encryption Standard (AES)

AES is a cryptographic algorithm defined by the US National Institute of Standards and Technology (NIST) in 2001, as Federal Information Processing Standard (FIPS) Publication 197 [15] and used by US Federal departments and agencies or non-Federal Government organizations to protect sensitive and unclassified electronic data.

It is a symmetric block cipher used to encrypt and decrypt information with the same key, based on the Rijndael algorithm. Unlike Rijndael that handles more block sizes and key lengths, the AES is designed to encrypt and decrypt data blocks of 128 bits using cipher keys of 128, 192, or 256 bits. It has supplanted the Data Encryption Standard (DES) in many cryptography applications.

D. Counter mode (CTR)

CTR [16] is a mode of operation that is used in conjunction with a symmetric key block cipher algorithm, such as AES, to obtain confidentiality over several blocks using the same cipher key, and to avoid pattern recognition by a possible intruder. For each block, a counter block, that must be different among the blocks for all messages under a given key (also called *nonce*), is encrypted using a block cipher with a given key to produce the *keystream*. The resulting output block is then XORed with the corresponding *plaintext* block to obtain the *ciphertext* block and vice versa. For the last block, which may be a partial block, only the most significant bits related to the partial block length are used whereas the remaining bits are discarded. For both CTR encryption and decryption, the forward block cipher functions can be performed in parallel. The CTR mode of operation is shown in Fig. 3, for both encryption and decryption, where K is the symmetric cipher key.

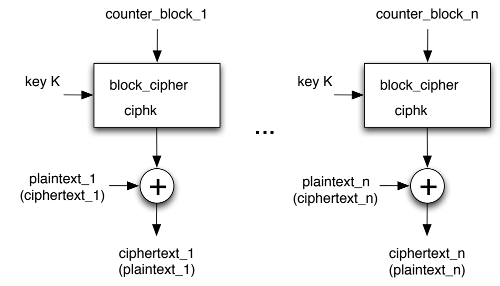


Figure 3: Diagram of the CTR encryption (decryption) mode.

E. CBC-MAC mode

The cipher block chaining with message authentication code is a procedure to generate a message authentication code using a block cipher in CBC [16] operation mode.

The message to be authenticated is encrypted in CBC mode with a deterministic initialization vector consisting of an all-zero block. The sequence to be encrypted must be padded to a multiple of the cipher block size (*block size*), simply by adding zeroes at the end of the message. Further, the padded data is divided into blocks, each matching the block size. The first message block is XORed with the initialization vector before being encrypted using a symmetric-key block cipher with the same key. The result is then XORed with the second message block and encrypted again. The last message block is XORed with the previous block and given as input to the block cipher. The resulting block, or part of it (leftmost bits), represents the authentication tag. Fig. 4 shows the CBC-MAC mode, where *plaintext i* represents the *i-th* message block.

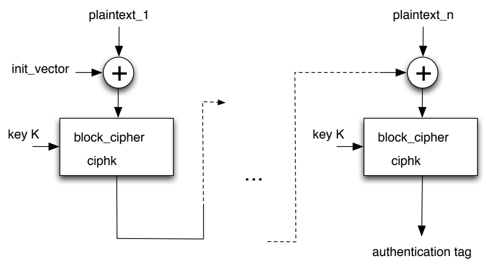


Figure 4: Diagram of the CBC-MAC mode of operation.

As CBC-MAC is not secure for variable-length messages, usually the length of the message is included in the first block. Since the input block to each encryption operation depends on the result of the previous one, it cannot be run in parallel. At the receiver side, the message authentication code is computed and compared with the received one to verify authentication and integrity of the received data message.

F. Counter with CBC-MAC (CCM) mode

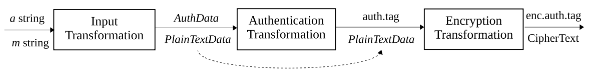
The CCM mode [17, 18] provides both authentication and confidentiality of data by combining the techniques of CTR and CBC-MAC modes. Further details on CCM follow in Section III.

III. Implementation Details

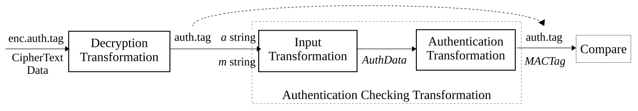
CCM is a generic combined encryption and authentication block cipher mode that is also specified by the IEEE 802.11 MAC standard. Specifically, CCM* coincides with the original specification of CCM [17] for messages that require authentication and encryption, but it additionally supports messages that only require encryption. Moreover, it can also be employed where the use of variable-length authentication tags is preferable (rather than fixed-length authentication tags). Like CCM, CCM* also requires a single secret key [11].

The block cipher used in the implementation of CCM* is AES-128 as specified in [15]. This block cipher is used with symmetric keys with the same size as that of the block cipher, which in our implementation has been set to 128 bits.

The implementation of CCM* includes two procedures: forward transformation (Fig. 5(a)), executed at the transmitter side to encrypt and authenticate link layer packets before their transmission over the channel and inverse transformation (Fig. 5(b)), executed at the receiver side to decrypt and validate the data. The forward transformation involves the execution of: an input transformation (CCM-1), an authentication procedure (CCM-2) and an encryption transformation (CCM-3), as shown in Fig. 5(a). CCM-1 involves the transformation of the two CCM* input strings *m* (the original message) and *a* (an associated data sequence) onto the strings *PlainTextData* and *AuthData*, to be used by the encryption transformation and the authentication procedure, respectively. Thus, the authentication transformation step tags the *AuthData* using the tagging transformation and produces an authentication tag. Finally, the *PlaintextData* and the authentication tag formed earlier are encrypted using the encryption transformation step resulting in an encrypted message called *Ciphertext* and an encrypted authentication tag.



(a) CCM* forward transformation



(b) CCM* inverse transformation

Figure 5: Block diagram of CCM* mode of operation.

The CCM* mode inverse transformation involves the execution of a decryption transformation and an authentication checking transformation as shown in Fig. 5(b). The decryption transformation is similar to the encryption transformation explained above, resulting in a decrypted authentication tag and the *m* string. Further, the authentication checking transformation step uses the input transformation explained above to form *AuthData* by using as inputs the *a* string and the *m* string established during decryption transformation. Furthermore, it employs authentication transformation, with *AuthData* as input, to form *MACTag* and compares it with the authentication tag established during decryption transformation. If both the tags are equal, the authentication is valid; otherwise the resultant *a* and *m* strings are rejected thus completing the overall CCM* procedure. Further details about all the CCM* implementation steps described above are available in [11, 18] and are beyond the scope of this paper.

In order to determine the best possible solution in terms of efficiency and complexity, we have considered several combinations of both software and hardware implementations of our security module and its constituents. While the AES block cipher is implemented in both hardware and software, the CBC-MAC and the CCM* engine were only implemented in software, as the corresponding functionalities are not provided by our hardware architecture. In greater detail, the hardware platform for which we have implemented our protocol stack features an AVR XMEGA AU micro-controller [19], that is combined with an AVR AT86RF231 radio transceiver [20]. The XMEGA AU micro-controllers is a family of low-power, high-performance, and peripheral-rich CMOS 8/16-bit micro-controllers based on the AVR enhanced RISC architecture, whereas the AT86RF231 is a low-power 2.4 GHz radio transceiver designed for industrial and consumer IEEE 802.15.4, ZigBee, 6LoWPAN, RF4CE, SP100 and WirelessHART.

The experimental setup used during our measurement trials included the aforementioned hardware modules along with a digital storage oscilloscope and a computer running on a 3.4 GHz Intel Core i7 CPU under the UNIX operating environment.

IV. Results and Analysis

The performance of our DLL security suite has been evaluated in terms of parameters such as execution time, memory usage and energy consumption. In what follows, we use these metrics to compare hardware and software implementations of the AES block cipher and to analyze the overall CCM* mode of operation. For all of them we considered both optimized and unoptimized versions. Optimization can be achieved through different criteria such as the minimization of the code size, the memory usage or the execution time without compromising on the code correctness. Since our tightest requirement is related to the code size, we opted for the *space* option that tries to reduce the image size of the compiled code as much as possible.

A. Time and memory requirements

The first set of measurements pertaining to the time required by different security operations is analyzed below.

According to Atmel's application notes [19, 21], the AES crypto module requires 375 clock cycles to execute one encryption/decryption operation, when key and data are loaded and the mode of operation is selected. Considering the AVR XMEGA A3 micro-controller's reference clock frequency of 32 MHz [19], the time required to complete a single encryption(decryption) operation is given by:

$$t_{\text{enc/dec}} = t_{\text{clk}} \cdot N_{\text{clock\_cycles}} = \frac{1}{32\text{ MHz}} \cdot 375 \simeq 11.719\ \mu\text{s}.$$

The corresponding experimental measurements, obtained for optimized and unoptimized code, are given in Table II. The execution times are also expressed in terms of clock *ticks* alongside, by again taking into consideration the micro-controller's clock frequency of 32 MHz. Our experimental measurements closely match those predicted by the Atmel application notes.

TABLE II
AES HARDWARE ENCRYPTION/DECRYPTION.

	Time Required	Ticks
Optimized Code	12.00 μs	384
Unoptimized Code	12.80 μs	410

The delay performance of encryption and decryption operations for a block of 128 bits (according to AES-128) are given in Table III, where the delay is expressed in terms of seconds and clock ticks (note that these measurements also include the time required to load the secret key and the data block to be encrypted).

TABLE III
AES ENCRYPTION/DECRYPTION TIME FOR A BLOCK OF 128 BITS.

	Optimized Code		Unoptimized Code	
	Time	*Ticks*	*Time*	*Ticks*
AES Hardware	41.60 μs	1331	160.00 μs	5120
AES Software	1.44 ms	46080	2.94 ms	94080

As shown in Table III, the software implementation of AES is much slower than its hardware counterpart. This was expected, and is due to the fact that the former requires several accesses to the flash memory in order to perform the required operations, and every memory access involves a significant amount of time. In fact, we implemented all the encryption constants as look-up tables in flash memory in order to save as much SRAM as possible. In our architecture, this is achieved using specific compiler attributes provided by the compiler, e.g., `PROGMEM`. The values of time required to read either a byte or a double word (4 bytes) from the flash memory are listed in Table IV.

TABLE IV
TIME REQUIRED FOR EACH ACCESS TO THE FLASH MEMORY.

	Optimized Code		Unoptimized Code	
	Time	*Ticks*	*Time*	*Ticks*
Byte	8.20 μs	262	15.40 μs	493
Double Word	11.40 μs	365	24.00 μs	768

Regarding the CCM* mode of operation, we recall here that CCM*'s encryption consists of three phases CCM-1, CCM-2 and CCM-3 as explained in Section III and, in addition to these, CCM*'s decryption also requires the validation of the decrypted data. In Table V, we show the performance of the complete CCM* forward (i.e. encryption and authentication at the transmitter) and CCM* inverse (i.e. decryption and validation at the receiver) procedures in terms of execution time and consumed energy, considering a link layer packet payload size of 95 bytes, which is the maximum possible packet payload size for the security level considered in our implementation (i.e. ENC-MIC-128) considering the values of MAC header (14 bytes), FCS (2 bytes), MIC (16 bytes) and the maximum PSDU size (127 bytes) as defined in [11]. These results include the time necessary to execute the three phases CCM-1, CCM-2 and CCM-3 and also that required to create the *nonce* and the associated *data* sequences, as required by the algorithms. The performance related to each of the constituent operations (CCM-1, CCM-2 and CCM3) is also given in the table.

It should be noted that while the AES block cipher can be implemented in either hardware or software, as indicated in Table V, the CBC-MAC and the CCM* engine are implemented in software.

The total amount of memory consumed by each of the four security implementations that we consider here is given in Table VI. Note that the overhead pertaining to accessing the flash memory (ROM) during the software mode of operation is quite high even in terms of memory consumption. Specifically, the ROM memory footprint of the hardware-based implementation is about six times smaller than that of the software-based implementation. However, there was no significant difference between the four modes with respect to RAM consumption.

TABLE VI
OVERALL MEMORY CONSUMPTION IN BYTES.

Security Mode	ROM	RAM
Hardware Optimized	4495	23
Hardware Unoptimized	7041	23
Software Optimized	21459	37
Software Unoptimized	26183	37

B. Energy efficiency and implementation complexity

The contour plots in Fig. 6 and Fig. 7 depict the impact of introducing DLL security within the IoT protocol stack of Section II-A. We measure the energy efficiency of our DLL security implementations in terms of their effect on the overall lifetime of the network as illustrated in these contour plots. The x-axes represent the number of relay nodes in the network, whereas the y-axes show the inter-packet transmission intervals. The numerical values expressed as percentages shown above the contour lines represent the reduction in lifetime between a network setup where the nodes do not implement any DLL security feature and the same network scenario where all nodes implement DLL security. These figures have been obtained considering a network setup where a number of nodes (shown on the abscissa) have been deployed according to a tree topology, 6LoWPAN has been used as the addressing technology and RPL [10] has been exploited to route data to a data collection node located at the root of the tree.

Remarkably, the impact of adding our DLL security suite is negligible when we utilize a hardware-based optimized implementation as can be seen in Fig. 6. In this case, the decrease in the nodes' lifespan is less than 2% under all network configurations (i.e. varying number of nodes and inter-packet transmission times). This is a very good result which makes DLL security a viable option even for resource constrained IoT nodes. Note however that the impact of adding DLL security

TABLE V
EXECUTION TIME AND ENERGY CONSUMPTION PERFORMANCE OF THE HARDWARE- AND SOFTWARE-BASED IMPLEMENTATIONS OF THE DLL SECURITY SUITE. RESULTS ARE SHOWN FOR BOTH OPTIMIZED AND UNOPTIMIZED CODE FOR A PAYLOAD SIZE OF 95 BYTES (ENERGY PERFORMANCE OF UNOPTIMIZED CODE HAS BEEN OMITTED DUE TO SPACE CONSTRAINTS).

Function	Optimized code						Unoptimized code			
	AES Hardware			AES Software			AES Hardware		AES Software	
	Time	*Ticks*	*Energy*	*Time*	*Ticks*	*Energy*	*Time*	*Ticks*	*Time*	*Ticks*
CCM* forward	1.44 ms	46080	114.05 μJ	28.80 ms	921600	2280.96 μJ	4.80 ms	153600	64.80 ms	2073600
CCM* inverse	1.50 ms	48000	118.80 μJ	29.20 ms	934400	2312.64 μJ	5.20 ms	166400	64.80 ms	2073600
CCM-1) Input	78.00 μs	2496	6.17 μJ	78.00 μs	2496	6.17 μJ	102.40 μs	3277	102.40 μs	3277
CCM-2) Authentication	700.00 μs	22400	55.44 μJ	15.00 ms	480000	1188.00 μJ	2.56 ms	81920	33.60 ms	1075200
CCM-3) Encryption	576.00 μs	18432	45.62 μJ	13.80 ms	441600	1092.96 μJ	1.92 ms	61440	30.40 ms	972800

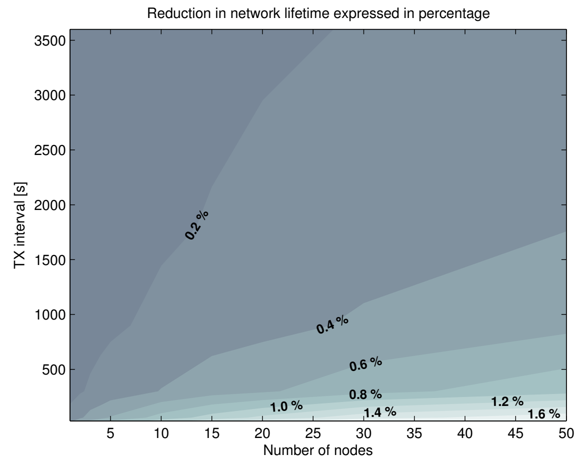


Figure 6: Energy consumption comparison: no security vs. optimized hardware-based security implementation.

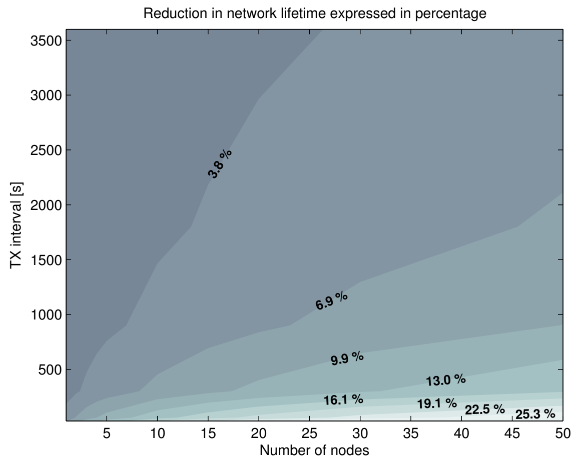


Figure 7: Energy consumption comparison: no security vs. optimized software-based security implementation.

is much more pronounced for software-based implementations (see Fig. 7). In this case, when the number of nodes is high (in the range $30 - 50$) the lifetime can reduce by about 25% for inter-packet transmission times shorter than 500 seconds.

To better put these percentages into perspective, we note that lifetime differences of $1-2\%$ roughly correspond to $5-8$ hours of reduced operational time. This is the price we have to pay for the addition of DLL security features, which we deem acceptable considering that typical values of the WSN lifetime span between three months and two years (depending on the selected inter-packet transmission time).

These fine results are due to the fact that the energy spent for the processing required by the DLL security algorithms is orders of magnitude smaller than that required by the radio for transmission and reception activities of data and control packets (as those required to maintain the network structures for routing and data collection). These facts dominate the overall energy budget (compare the energy figures in Table V with respect to the energy drained by the transmission of a packet, which takes about 480 μJ). For the software-based implementation, instead, the security overhead impacts more on the lifetime, as shown in Fig. 7 and this is due to their much longer processing times, which entail a much higher energy expenditure.

In Fig. 8 we focus on the implementation complexity, which is the amount of time taken by each of the constituent blocks of the CCM* security module vs. the link layer packet payload size (reported on the abscissa). In this figure, we show results for both the hardware- (HW) and software-based (SW) implementations, by specifying the results for the complete CCM* security procedure (i.e. including CCM-1, CCM-2 and CCM-3 and referred to in the figure as CCM_AES) as well as the time taken by the execution of the sole authentication procedure (CCM-2, referred to in the figure as AUTH_AES) and by the encryption transformation (CCM-3, referred to in the figure as ENC_AES). Due to space limitations we only show the results for the optimized (opt) HW implementation and the unoptimized (unopt) SW implementation, which can be considered as the best and worst cases respectively in terms of performance. Clearly, the authentication procedure (AUTH_AES_HW) is computationally more intensive than encryption transformation (ENC_AES_SW and ENC_AES_HW) for both software- and hardware-based implementations. In addition, HW implementations outperform SW ones by about two orders of magnitude.

Fig. 9 shows the total time taken by the entire CCM* procedure (including CCM-1, CCM-2 and CCM-3) as a function of the link layer packet size. We hereby provide curves for HW and SW implementations for both the optimized and unoptimized code. As expected, the hardware optimized (HW_opt) security mode performs best, whereas the software unoptimized (SW_unopt) security mode fares the worst. Note that the results

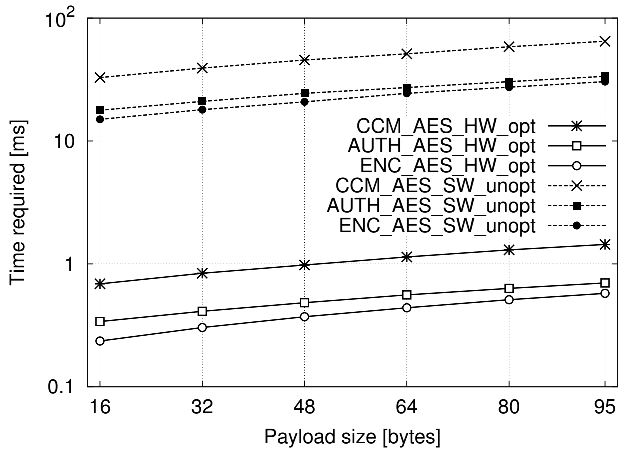


Figure 8: Time required by each operation for selected CCM* security modes (best and worst methods only).

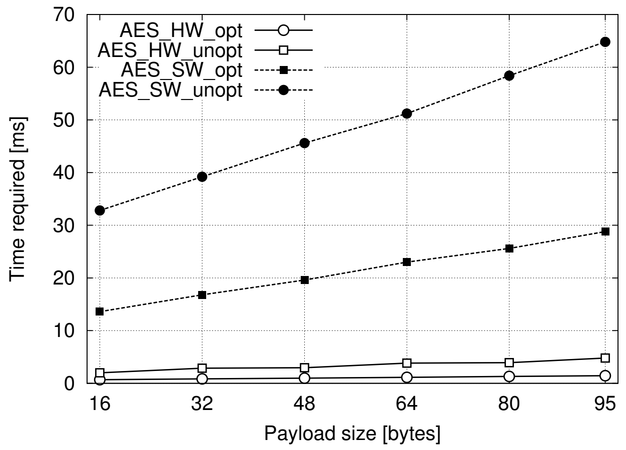


Figure 9: Total time required by CCM* security (all methods).

in this figure, for a payload size of 95 bytes, correspond to those shown in Table V. Also, the performance of all modes varies linearly with the packet payload size.

V. Conclusions

In this paper we have presented the implementation of IEEE 802.15.4-compliant link layer security procedures. To begin with, we briefly describe our Internet of Things protocol stack architecture, then delve onto the description of the security functionalities that are to be implemented at the data link layer for complete IEEE 802.15.4 compliance. Thus, we have detailed all the aspects of our implementation and have finally presented our experimental results to quantify the performance of the corresponding encryption and authentication routines for the selected implementation modes. From our experimental analysis, we conclude that hardware implementations of AES ciphers are preferred as they provide substantial savings in terms of memory (up to six times) and, at the same time, in terms of delay reduction (up to two orders of magnitude) compared to their software-based counterparts. Also, the addition of hardware-based link layer security features has no significant impact on the network lifetime performance. Specifically, compared to network scenarios where the data link layer does not include any security feature, hardware-based security leads to wort-case lifetime reductions of just 2%. On the other hand, software-based implementations may have detrimental effects on the lifetime performance, leading to reductions as high as 25% in extreme cases.

Acknowledgment

This work has been supported in part by the European Commission through the FP7 EU project titled "Internet of Things–Architecture (IoT-A)" (G.A. no. 257521) and by the FP7 EU project "Symbiotic Wireless Autonomous Powered system (SWAP)" (G.A. no. 251557).

References

[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A Survey," *Elsevier Computer Networks*, vol. 54, no. 15, Oct. 2010.

[2] H. Suo, J. Wan, C. Zou, and J. Liu, "Security in the Internet of Things: A Review," in *IEEE ICCSEE*, Hangzhou, China, Mar. 2012.

[3] Y. Wang, G. Attebury, and B. Ramamurthy, "A Survey of Security Issues in Wireless Sensor Networks," *IEEE Communications Survey*, vol. 8, no. 2, 2nd Quarter 2006.

[4] G. Bianchi, A. T. Capossele, A. Mei, and C. Petrioli, "Flexible Key Exchange Negotiation for Wireless Sensor Networks," in *ACM WiNTECH*, Chicago, IL, US, Sep. 2010.

[5] R. Bonetto, N. Bui, V. Lakkundi, A. Olivereau, A. Serbanati, and M. Rossi, "Secure Communication for Smart IoT Objects: Protocol Stacks, Use Cases and Practical Examples," in *IEEE IoT-SoS*, San Francisco, CA, US, Jun. 2012.

[6] L. Batina, N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede, "Low-Cost elliptic curve cryptography for wireless sensor networks," in *ACM ESAS*, Hamburg, Germany, Sep. 2006.

[7] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "RFC4944: Transmission of IPv6 Packets over IEEE 802.15.4 Networks," IETF Request For Comments, Sep. 2007.

[8] Z. Shelby and C. Borman, *6LoWPAN: The Wireless Embedded Internet*. Wiley, Nov. 2009.

[9] Z. Shelby, K. Hartke, C. Bormann, and B. Frank, "Constrained Application Protocol (CoAP)," IETF Internet Draft, draft-ietf-core-coap-13, Dec. 2012.

[10] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander, "RFC6550: RPL – IPv6 Routing Protocol for Low-Power and Lossy Networks," IETF Request For Comments, Mar. 2012.

[11] IEEE 802.15.4, "Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)," http://www.ieee802.org/15/pub/TG4.html, Sep. 2006.

[12] A. P. Castellani, M. Gheda, N. Bui, M. Rossi, and M. Zorzi, "Web Services for the Internet of Things through CoAP and EXI," in *Proceedings of IEEE ICC*, Kyoto, Japan, Jun. 2011.

[13] D. C. J. Polastre, J. Hill, "Versatile Low Power Media Access for Wireless Sensors Networks," in *ACM SenSys*, Baltimore, MD, USA, Nov. 2004.

[14] M. Buettner, G. V. Yee, E. Anderson, and R. Han, "X-MAC: a Short Preamble MAC Protocol for Duty-Cycled Wireless Sensor Networks," in *ACM SenSys*, Boulder, CO, USA, Oct. 2006.

[15] FIPS Publication 197, "Advanced Encryption Standard (AES)," U.S. DoC/NIST, 2001.

[16] NIST Special Publication 800-38A, "Recommendation for Block Cipher Modes of Operation," U.S. DoC/NIST, 2001.

[17] D. Whiting, R. Housley, and N. Ferguson, "RFC3610: Counter with CBC-MAC (CCM)," IETF Request For Comments, Sep. 2003.

[18] NIST Special Publication 800-38C, "Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality," U.S. DoC/NIST, 2004.

[19] "Atmel AVR XMEGA AU 8-bit Microcontroller Manual," http://www.atmel.com/Images/Atmel-8331-8-and-16-bit-AVR-Microcontroller-XMEGA-AU_Manual.pdf, 2013.

[20] "AT86RF231: AVR Low Power 2.4 GHz Transceiver," http://www.atmel.com/Images/doc8111.pdf, 2009.

[21] "AVR1318: Using the XMEGA built-in AES accelerator," http://www.atmel.com/Images/doc8106.pdf, 2008.