On the Cybersecurity of m-Health IoT Systems with LED Bitslice Implementation

AbdelRahman Eldosouky and Walid Saad
Wireless@VT, Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA
Emails:{iv727,walids}@vt.edu

***Abstract*—The Internet of Things (IoT) will provide a large-scale infrastructure that can support a plethora of new networked services. One critical IoT application pertains to m-Health services which allow monitoring the health status of patients while providing the ability for a rapid response in emergency cases. Connecting healthcare services to the IoT brings forward new security threats and vulnerabilities that can jeopardize the patients' private data. In this paper, a novel security framework for m-Health IoT security is proposed using the concept of moving target defense (MTD). MTD allows the m-Health system to dynamically change its cryptographic keys to increase uncertainty on an attacker and secure the data. In the proposed scheme, the devices update their keys locally to eliminate the risk of revealing new keys while they are being shared with a gateway. A practical implementation is proposed based on bitslicing LED, a lightweight encryption cipher, to improve the performance of decrypting multiple packets at the same time. LED bitsliced implementation was tested on an ARM Cortex-A53 and was shown to consume half of the processor's instructions compared to the conventional implementation. The effect of applying MTD on the number of processor's instructions is evaluated and shown to be bounded.**

I. Introduction

The Internet of Things (IoT) is seen as a large-scale ecosystem that will integrate a heterogeneous mix of devices, sensors, and wearable devices. The IoT will be a major enablers for a variety of smart services that range from large-scale sensing to smart transportation [1] and healthcare [2]. M-Health systems that are wireless-enabled healthcare systems will be one of the primary services supported by the IoT system that will provide them with pervasive Internet access [3]. As discussed in [2], m-Health IoT systems include a number of smart devices and sensors that monitor a patient's medical conditions such as blood pressure, pulse or body temperature. The measured data is then sent to remote physicians via an access point or a gateway [4]. This gateway is responsible for collecting and sending the data [5] as well as providing wireless connectivity, via multiple networking interfaces, to the m-Health IoT devices.

This pervasive wireless connectivity for small, m-Health IoT devices, will bring forward new security challenges and vulnerabilities. Malicious attacks can now leverage the connectivity of these devices to launch remote attacks and potentially access the patients' critical data that is being transmitted by the m-Health devices. Taking these attacks into consideration, security and privacy constitute key concerns in all IoT systems. The work in [6] highlights the main security issues in the IoT while outlining the main existing solutions that have been developed to maintain the confidentiality, authenticity, and integrity of data in IoT. The authors discuss security features that need to be applied in a security architecture of four levels distributed between the devices and the cloud. Device authentication, data encryption, and key agreement are highlighted as the most critical security requirements that need to be addressed at the devices side. Note that some wireless security approaches, e.g., physical layer security [7] cannot be used with the IoT due to its heterogeneous nature.

Recently, such security requirements received significant attention in the literature due to the specific nature of the IoT. The huge number of heterogeneous limited-resources devices in the IoT complicate the security mechanisms. The limited resources make it hard for the IoT devices to run complex security algorithms. Hence, lightweight encryption techniques are seen as the cornerstone of IoT security. In [8], the authors present a lightweight encryption method to authenticate RFID tags at the readers. The work in [9] evaluates two major types of attribute-based encryption on different IoT devices. This work shows that the performance of attribute-based encryption cannot be readily deployed in small IoT devices, due to resource constraints. A more recent work in [10] proposes a lightweight attribute-based encryption scheme based on elliptic curve cryptography. The scheme is shown to have low communication overhead provided that the number of attributes remains small.

While data encryption is not a sufficient security mechanism for the IoT [11] as it does not protect against insider attacks, that is not the case in m-Health IoT. In the IoT, both data encryption and device authentication [12] are taken into consideration. However, as the devices in an m-Health system are usually operated around the patient and known to the gateway, security mechanisms are oriented more towards data encryption for enhancing data privacy. In [4], the authors provide a prototype for applying asymmetric, public key, encryption in an m-Health IoT system. Due to the high computational power of public key encryption, it is applied at the level of the gateway. The more recent work in [13] demonstrates the benefits of applying cloud computing in an m-Health system by using hybrid encryption schemes. In hybrid schemes, symmetric secret key encryption, which is known to have low computational power, is used between devices and the gateway while a public key, which consumes significant power but provides more security, is used between the gateway and the cloud. However, in all these systems, using secret key encryption can be problematic if the key was revealed by an attacker through any of the known attacks.

One promising technique to improve a system's security is the so-called moving target defense (MTD) [14]. MTD is the concept of continuously randomizing a system's configuration in order to increase the uncertainty and cost of an attack. In the IoT, the system's configuration can essentially include encryption keys, network parameters or IP addresses. While applying MTD improves a system's security, it can also incur some costs that

This research was supported by the U.S. National Science Foundation under Grants CNS-1524634, OAC-1541105, and OAC-1638283.

reduce the overall performance. The authors in [15] applied MTD by frequently changing the IP address of IoT devices to increase the security. Security improvement and network latency were studied for implementing MTD over low-powered personal area networks. The work in [16] applies MTD using a stochastic game between an attacker and a base station acting as a defender. Multiple encryption techniques with multiple shared secret keys are implemented at the nodes. The security benefit as well as the MTD costs are studied in this scenario. However, existing MTD works such as [15] and [16] are not designed for m-Health IoT systems and do not provide specific implementations of the encryption system.

The main contribution of this paper is an MTD security framework tailored to the unique nature of m-Health IoT systems. The framework uses a hybrid encryption scheme in which secret keys are used to encrypt the data sent from the devices to the gateway, and a public key to encrypt data from the gateway over the Internet. The proposed MTD scheme is applied by frequently changing the secret keys used in the communication between the devices and the gateway. The gateway takes the decision to update all the keys in the network hence allowing each device to calculate its new key and start using it. The new encryption key is generated by encrypting the old key using another pre-shared key. Hence, only two secret keys need to be pre-shared between each device and the gateway. A case study is provided to study the effect of applying MTD on an enhanced (in terms of performance) real system. In this system, a lightweight encryption technique, LED [17], is used for encrypting the data. As gateways typically apply performance improvement techniques to speed up the process of decrypting the collected data, we propose a new bitslice implementation for LED that can be used at the gateway. To the best of our knowledge, this is the first 64-bit bitslice implementation for LED algorithm. We also provide a modified 32-bit version suitable for 32 bit registers. The system is tested on a virtual 64-bit ARM Cortex-A processor and the results show that the bitslice implementation consumes half of the processor's instructions compared to the original LED implementation. Results also show that using MTD and bitslice does not yield any significant degradation in the system's performance when some packets are missed compared to the original implementation.

The rest of the paper is organized as follows. Section II presents the proposed security mechanism and the MTD scheme. In Section III, the bitslice implementation is presented in detail, and the metrics used to measure the performance improvement are discussed. Performance evaluation using a real-world implementation are presented in Section IV. Finally, conclusions are drawn in Section V.

II. Encryption model in M-Health IoT system using Moving Target Defense

Consider an m-Health network consisting of a number of smart devices and sensors, referred to as nodes, that monitor a patient's medical condition and send the measured readings to a gateway. The gateway will send the collected data over the Internet to a remote hospital or a clinic. Unless there is an emergency that needs to be reported, we assume all the devices are synchronized to send frequent updates about what they sense or measure. The frequency of sending the updates depends on the medical situation and the criticality of the patient's health status.

All the data sent from the nodes is encrypted at each node before it is sent to the gateway. The gateway decrypts the received data and re-encrypts it using a more powerful encryption algorithm to be sent over the Internet. Due to the resource limitations of the IoT nodes, a lightweight encryption technique should be used to encrypt the data at every node. Typically, symmetric algorithms, which use a pre-shared secret key, are less power demanding than asymmetric algorithms, which use two different keys known as public and private keys. Therefore, symmetric lightweight algorithms are more suitable for IoT nodes. A secret key must be shared between every node and the gateway prior to connecting to the Internet. At a given node i, a plaintext P is encrypted using node i's secret key K_i to get the ciphertext $C = E_{K_i}(P)$. The gateway, which is a computationally capable device, will then use an asymmetric algorithm to decrypt the data and send it over the Internet. This makes the m-Health encryption system, a hybrid system combining both symmetric and asymmetric encryption algorithms.

Symmetric encryption algorithms can be vulnerable to some attacks like brute force attacks, known plaintext attacks, chosen plaintext attacks, and differential cryptanalysis attacks. The goal of all such attacks is to reveal the secret key used in the encryption allowing the adversary to access and read the private data or even send fake data impersonating another node by using its key. To mitigate the effect of a successful attack and make the system more resilient, we use MTD by frequently changing the secret key used in the communication between every node and the gateway. The gateway decides to update the keys and informs the devices which should start using the new keys immediately. The time needed to apply MTD, i.e., initiating new keys is decided by the gateway depending on the frequency of sending new packets from the devices. This potential time delay yields a trade-off between increasing the attacker's chance to perform a successful attack and incurring more cost by frequently changing the keys as discussed in Section III.

New secret keys are calculated by encrypting the old keys, within each node, using another pre-shared secret key referred to as the MTD key. Given a key K_i used by a node i, the new key will be given by $K_{i_{\text{new}}} = E_{K_{\text{MTD}}}(K_i)$, where K_{MTD} is the pre-shared MTD key. Consequently, both the gateway and the device can get the new key without having to share any additional keys. Fig. 1 shows the proposed model for m-Health security mechanism. Note that each node can use a different MTD key.

The use of MTD in this mechanism will increase the uncertainty on any attacker, thus improving the security of the system. This is due to the fact that there is no fixed key, no fixed time to change the key, and the new keys are generated locally to eliminate the possibility of being intercepted. Even if the attacker was able to reveal one or more keys, it will not be able to reveal the new keys as they are generated using the MTD key which is stored locally at each node. Therefore, the attacker will lose any privilege once the keys are updated and will have to start a new attack.

Finally, the security mechanism proposed here does not require

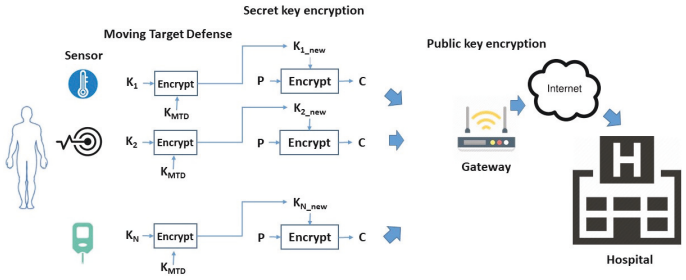

Fig. 1. MTD security mechanism for m-Health IoT system.

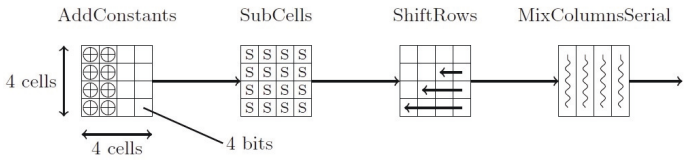

Fig. 2. The four operations in a single LED round [17].

any device-level hardware modification. It only requires a small software modification to add the pre-shared keys and to allow the nodes to respond to the gateway signals of changing the key. Next, we present the practical implementation of the proposed mechanism. We define a performance improvement technique to be used by the gateway in decrypting a number of packets at once and then study the effect of applying MTD on the performance.

III. Case Study: LED Bitsliced Implementation

A. LED Block Cipher

Lightweight encryption techniques are designed for resource-constrained devices. Some techniques target the hardware such as area on the chip, power, or energy consumption while others provide light software such as low memory and small code size. In this case study, we choose to implement LED block cipher [17]. LED is hardware-oriented which provides the smallest silicon footprint in its class of block ciphers with a reasonable performance. LED was chosen for this case study as hardware consumption is more critical because the software performance can be improved by some techniques as shown later.

In terms of design, LED's design is similar to the design of advanced encryption standard (AES) schemes. The main difference between LED and AES is that LED uses no key schedule and the same key is applied every round. The user-provided key can range from 64-bit to 128-bit. Increasing the key length will increase the security and the power consumption as well. In this work, since we adopt MTD and we depend on the key change as a defense, no need to consider longer keys which consume more computational power and hence a 64-bit key will be suitable. LED applies rounds like AES. In each round, four operations are applied to the state, which are: AddConstants, SubCells, ShiftRows, and MixColumnsSerial. The state refers to the current input to each round, which is initially the plaintext. Fig. 2 shows the four operations in each round.

In AddConstants, some predefined constants are combined with each state's bits. SubCells is used to replace the bytes of each cell in the state using an S-Box. ShiftRows is used to

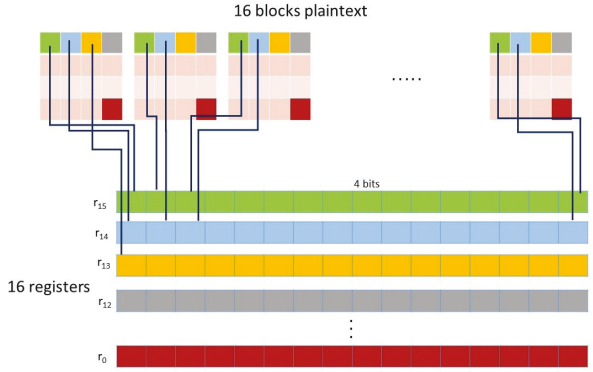

Fig. 3. Bitsliced representation of 16 plaintext blocks into 16 64-bits processor's registers. Colors represent data that is stored in the same register.

rotate the cells to the left a number of times depending on their row. Finally, MixColumnsSerial is used to multiply the cells by another predefined matrix and the multiplication is done over a defined Galois field. LED applies this round four times to the same state before adding the key. This process is repeated eight times, i.e., in total 32 rounds are applied to the state.

B. LED Bitsliced Implementation

Bitslicing is the process of slicing the data into its bit level and performing the required operations on these bits. Bitslicing is designed for a specific processor size, e.g., a 32-bit or a 64-bit processor, which essentially maps to the size of the data types that the processor can handle. In this case study, we design a bitslicing scheme suitable for a 64-bit processor which can typically be found in gateways and modern mobile devices. Although bitslicing is not an optimization technique, it can offer a great flexibility to improve the performance if it is used appropriately.

LED encrypts a 64-bit plaintext. These 64-bits are organized in a state as a 4×4 matrix of nibbles and each nibble consists of 4-bits. In the design of bitslicing implementation, we take every nibble, 4-bits, to be the minimum chunk that will be processed. Every nibble will be stored in a different processor's register, hence 16 blocks of plaintext should be processed at the same time to make use of the 64-bits registers. In an m-Health IoT network, as the gateway receives data from multiple devices, it is very likely to have 16 or more plaintexts at a given time. Processing data in such different arrangements, requires modifying all the operations of the original LED. In addition to the four operations of LED and the key adding step that must be modified, an initialization operation need to be executed to transform data blocks into the desired arrangement in processor's registers. Next, each modified operation is discussed in detail.

- *BitTranspose*: The 16 blocks of plaintext as well as their corresponding 16 64-bit keys are transposed first to a form suitable for bitslicing. Sixteen 64-bits registers are needed. Fig. 3 shows our bitslice implementation arrangement. Every first nibble in each plaintext is stored in the first register, i.e., (r_{15}) at consecutive locations. The next nibbles are stored in the second register r_{14} and so on to fill all the remaining registers.
- *AddRoundKey*: The encryption key is added first before applying other operations. In AddRoundKey, every plaintext

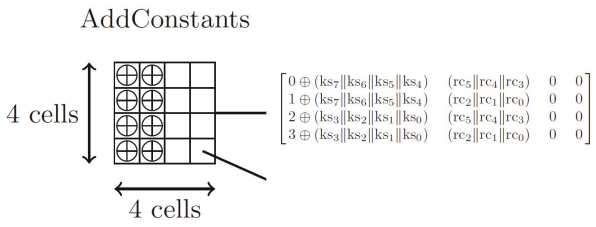

Fig. 4. AddConstants operation of LED.

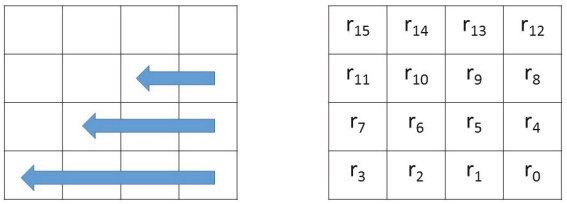

Fig. 5. Registers considered for swapping in ShiftRows operation of LED.

nibble is XORed with the corresponding nibble in the key. As the plaintext and the key are transposed using the same mechanism, every two nibbles need to be XORed will be in the same locations of the transposed plaintext and the transposed key. Therefore, a direct XOR operation can be applied to every pair of transposed registers which gives a total of 16 processor instructions to apply round keys. The original algorithm deals with separate nibbles and needs to XOR every nibble separately, which requires 16 instructions for every plaintext and, thus, 256 instructions for the 16 plaintext blocks. This process will be repeated 32 times before each round, thus the transposed representation reduces significantly the number of instructions required to apply round keys.

• *AddConstants*: In AddConstants, half of the nibbles are modified as shown in Fig. 4. Therefore, only eight out of the sixteen registers need to be updated. Instead of using the original constants provided by LED, new constants suitable for the bitslice representation need to be calculated from the original constants. These new constants will be stored and used directly each round. Each nibble of the first column, in each state, is XORed with one of four different values. These values are constant and, hence, can be computed in advance. As each register holds the same nibble in different plaintext blocks, each of these four values is concatenated sixteen times to fit all of the nibbles. For example, register r_{15} will be XORed with the new value 4 concatenated 16 times. The same is applied to registers r_3, r_7, and r_{11} which hold the nibbles of the first columns in all plaintext blocks. Nibbles in the second column, in each state, are XORed with specific three bits of the round constants. The values of these three bits, concatenated sixteen times, are stored in advance and, hence, can be used directly in the bitslicing implementation. This modification can save only a few processor instructions but is necessary for the bitslicing implementation.

• *SubCells*: In SubCells, each nibble is replaced by a corresponding value from the S-Box. As we still deal with a whole nibble, no modification need to be applied to the S-Box. Each nibble was separated from the register and then substituted from the S-Box which requires twice the processor instructions used in the original implementation.

• *ShiftRows*: In ShiftRows, each row of the state is shifted to the left by a multiple of four bits as shown in Fig. 5. The figure also shows which register is used to store each nibble in the state. We made use of the fact that each register in this implementation holds nibbles from the same location in each state. As such, all the nibbles in the register need to be shifted by the same amount. Therefore, instead of doing actual shifting, we need to just swap the registers. For example, register r_{10} is placed in r_{11} then r_9 is placed in r_{10} then r_8 is placed in r_9 and, finally, r_{11} is placed in r_8 and so on for similar registers. We need one temporary register for the swapping process, and five assignment instructions. This can save a lot of instructions from the original operations where nibbles were considered separately.

• *MixColumnSerial*: In this operation, a constant matrix is multiplied by the state matrix. Each row in this matrix is multiplied by a column in the state matrix to update one nibble in the state matrix. As multiplication is done nibble by nibble, we had to define a new MixColumn operation that extracts nibbles from the registers and use them in the multiplication process. Even though a number of extra instructions are needed for separation, the updated nibbles can be calculated for the all the 16 registers at one iteration. This parallel calculation allows, in total, saving a significant amount of instructions when compared to the original case in which each plaintext is processed separately. For example, nibbles from registers r_{15}, r_{11}, r_7, and r_4 are processed together in the multiplication process which is different from the original implementation in which multiple iterations are needed. The rest of the columns are processed in the same way.

Finally, another version of this bitslicing was designed to suit 32-bit processors. A 32-bit version is obtained by decrypting 8 data blocks instead of 16. The 8 blocks are stored in 16 32-bit registers in the same way discussed in the 64-bit version. Similarly, every nibble from the plaintext is stored in a different register. The rest of the operations will follow as the 64-bit version but dealing with a smaller size of input. This implementation could be used either for 32-bit processors or 64-bit processors that support 32-bit registers. This implementation is beneficial for the gateway when applying MTD as shown in Section IV.

C. Performance Metrics

The performance of bitslicing is maximized when the total number of data blocks is available at the same time, this is sixteen plaintext blocks in our implementation. Bitslicing uses a constant number of processor's instructions whether 16 blocks are available or not. Here, we calculate the average number of instructions needed to decrypt a plaintext block, a as follows:

$$a = \frac{N}{b},$$

where N is the total processor instructions and b is the actual number of blocks that were encrypted and is bounded by the maximum number of blocks allowed by the design which is 16 in our implementation. Clearly, if we have fewer than sixteen blocks, the average number of instructions per block will increase and degrade the performance.

Another metric that we consider is *the cost of applying MTD* in m-Health IoT systems that consist of heterogeneous devices

differing in their computational capabilities. Here, when the gateway asks the devices to update their keys, they can have different response times. Hence, they may encrypt new packets using the old encryption key while the gateway is expecting data encrypted with new keys. This incurs a processing cost at the gateway, which is the wasted processor's instructions to decrypt data with wrong keys and the extra instructions required to decrypt again using the old keys. We measure the wasted instructions as the difference between the number of instructions used to decrypt the maximum number of packets that was expected and the number of instructions used for the correctly decrypted packets. Th number of instructions needed to re-decrypt packets using old keys will differ according to the number of missed packets. The gateway is given the option to re-decrypt the missed packets using either the original LED implementation, the 32-bit slicing version, or the 64-bit bitslicing version. This choice depends on the implementation that will use the least number of instructions to re-decrypt the missed packets. The choice of the re-decryption technique and the mathematical formulation for the cost are discussed, in detail, in Section IV.

IV. Evaluating LED Bitsliced Implementation

For our evaluation, we use an ARM Cortex-A 64-bit processor as the target processor to evaluate our implementation. ARM 64-bit processors such as Cortex-A53 and Cortex-A57 can be found in many mobile devices. Evaluating the code on a real processor is challenging as other operations can affect the measurements. Therefore, we use the ARM development studio (DS-5) [18] which gives the ability to create a virtual processor emulator for a specific ARM processor then run the code on it. We use Cortex-A53 as our implementation processor, and we adjust the compiler optimization flags to the maximum performance in all the next experiments. In Cortex-A53, we can use both 64-bit registers or 32-bit registers which allows to use both our 32-bit and the 64-bit bitsliced implementations.

The designed bitslice implementation presented in the previous section is suitable for the encryption process, however, what is typically done on the gateway is the decryption phase. Therefore, we had to invert the encryption algorithm to get the decryption scheme. The inverted operations are applied in a reverse order to the original operations, i.e., InvMixColumnsSerial, InvShiftRows, InvSubCells, and InvAddConstants which are the reverse operations applied in order. The inverted operations are designed as follows. In InvMixColumnsSerial, the state is multiplied by the inverse of the constant matrix that is used in MixColumnsSerial. In InvShiftRows, the rows of the states are shifted to the right with the same criteria of shifting as in ShiftRows. In InvSubCells, the inverted S-Box is used for substitution. Finally, in InvAddConstants, the same round constants as AddConstants are used but provided in the reverse order of rounds. The bitslice is then applied in the same way as the encryption process.

First, the bitslice implementation is evaluated to measure the reduction in the number of processor instructions when applying bitslicing. We used the reference LED implementation provided by the work in [19]. Fig. 6 shows the average number of instructions required to decrypt one block of plaintext data when different number of plaintext blocks are available. We assume

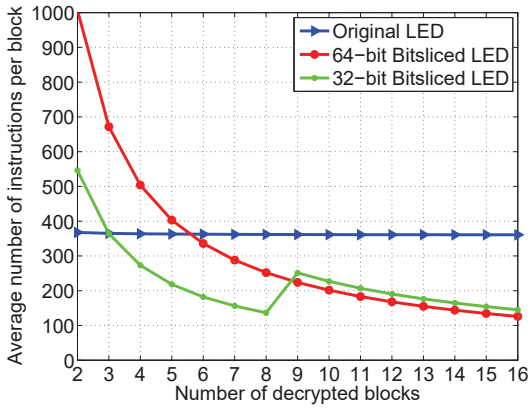

Fig. 6. Average number of instructions required to decrypt one block in the original LED implementation, 32-bit bitsliced applied twice, and the 64-bit bitsliced implementation. The number of instructions is normalized by 1000 for an easier representation.

that every block is received from a different device. We compare the original LED implementation, our 32-bit bitsliced version, and our main 64-bit bitsliced implementation. Note that the 32-bit version processes only 8 blocks at a time and, thus, we apply it twice for more than 8 blocks. From Fig. 6, we can see that the original implementation has an approximately constant average. In fact, there is only a small difference when the number of packets is small due to the processor initialization instructions having a higher effect on the total number of instructions. However, this difference is not significant. The 32-bit bitsliced version has the lowest average when there are 3 to 8 packets to be decrypted. Our bitsliced implementation requires half of the processor instructions required by the original implementation when decrypting 8 packets. The increase after decrypting 8 packets happens because the processor will start over to apply the bitslicing again and, thus, needs to execute more instructions. Therefore, applying the 32-bit version twice consumes a little bit more instruction than the 64-bit implementation. At 16 decrypted packets, our 64-bit implementation consumes half of the processor instructions compared to the original implementation.

Finally, the results in Fig. 6 allow the gateway to determine the algorithm that will be used to decrypt the number of available packets. If there are less than 3 packets, the original algorithm is preferred. The 32-bit implementation should be used when there are 3 to 8 packets. The 64-bit implementation is superior for more than 8 packets of data.

Next, we discuss the cost of applying MTD when bitslicing is used at the gateway. Note that bitslicing itself is known to increase the code size on the device, i.e., the gateway. However, as the gateway is assumed to be a computationally capable device, the increased code size will not be problematic so it will not be considered as a cost here. The focus will be on the number of wasted (or additional) processor instructions. Clearly, if all the devices will send their next packets with the updated key, no cost will be incurred. However, when some packets are received encrypted with the old key, the gateway will decrypt them using the new keys which will result in wrong packet formats. The gateway will conclude that the key is not updated yet in these devices and will re-decrypt these packets using the old keys.

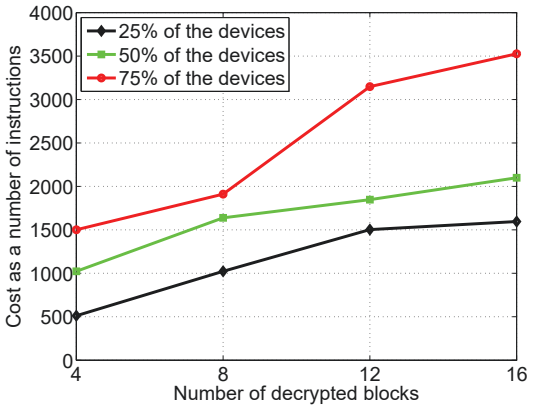

Fig. 7. The cost of applying MTD in a system with late response devices. The figure shows three cases for the percentage of the devices that will have a delay. The number of instructions is normalized by 1000 for an easier representation.

Thus, we can formulate the cost as follows:

$$C = \left\{ \begin{array}{ll} (b_{\max} - b) \cdot (\frac{L_{32}}{8} + R), & \text{for } 3 < b_{\max} \leq 8, \\ (b_{\max} - b) \cdot (\frac{L_{64}}{16} + R), & \text{for } 8 < b_{\max} \leq 16, \end{array} \right\}$$

where $b_{\max}$ is the maximum number of packets expected by the gateway, b is the number of successfully decrypted packets, and L_{32} and L_{64} are the total process's instructions for the 32-bit and 64-bit bitsliced versions, respectively. The decryption cost R is determined by the number of re-decrypted packets $b_{\max} - b$. If the number is 3 or less, the gateway will use the original LED to decrypt each packet individually, if the number exceeds 3 either version of bitslicing will be used and R will equal L_{32} or L_{64}.

Fig. 7 shows the cost in terms of gateway processor instructions if some devices send a single packet with the old encryption key. Three cases are considered when one quarter, half, and three quarters of the devices will send one packet with the old key. In case only a quarter of the devices wrongly encrypt one packet, we observe that the increase rate in the cost is less after twelve packets. This is due to the fact that, after twelve packets, the quarter will exceed three packets and, hence, the 32-bit bitslice version will be used to re-decrypt the missed packets thus reducing R as well as the total cost. A similar behavior can be seen for the case of half of the devices, where the cost increases at a slower rate after eight packets when the 32-bit bitslice version is used. However, in the case of three quarters of the devices, the increase in cost is higher after eight packets as the 32-bit version was used until eight packets, i.e., $R = L_{32}$ and the 64-bit version will be used after that consuming more processor's instructions as $R = L_{64}$.

It is interesting to note that the maximum cost according to this implementation is when all the sixteen packets need to be re-decrypted, i.e., $C_{max} = 2 \cdot L_{64}$. As L_{64} equals half of the instructions required by the original LED implementation as shown in Fig. 6, then the maximum cost equals the same number of processor instructions of the original LED implementation, if no packet is missed. Missed packets due to using MTD, with the original LED implementation, are re-decrypted individually causing more cost. Thus, the worst-case cost of applying MTD with bitslicing is bounded by the best-case, no cost, of applying MTD with the original LED implementation.

V. Conclusion

In this paper, we have proposed a novel security mechanism for m-Health IoT systems. The mechanism depends on using secret keys between the devices and the gateway and then applying MTD by frequently changing the encryption keys used in the network. The new key is calculated by encrypting the old key using another pre-shared secret key known as the MTD key, hence only one key needs to be shared between the gateway and each device. We have applied this mechanism to a system which involves a performance improvement technique for the encryption algorithm using bitslicing. We have formulated a 32-bit and 64-bit bitslicing implementations for LED, a light weight encryption technique. We have also defined performance metrics for the system including the cost for applying MTD. We have used a virtual processor to evaluate both bitslicing implementations and the cost of applying MTD. Implementation results have shown that the bitslicing implementation significantly outperforms the original implementation of the encryption algorithm. We have also discussed the optimal packet number for using both bitslicing versions.

References

[1] M. Mozaffari, W. Saad, M. Bennis, and M. Debbah, "Unmanned aerial vehicle with underlaid device-to-device communications: Performance and tradeoffs," *IEEE Transactions on Wireless Communications*, vol. 15, no. 6, pp. 3949–3963, June 2016.

[2] R. S. Istepanian, A. Sungoor, A. Faisal, and N. Philip, "Internet of m-health things 'm-iot'," in *IET Seminar on Assisted Living 2011*. IET, 2011, pp. 1–3.

[3] R. Istepanian, S. Laxminarayan, and C. S. Pattichis, *M-health*. Springer, 2006.

[4] C. Doukas, I. Maglogiannis, V. Koufi, F. Malamateniou, and G. Vassilacopoulos, "Enabling data protection through pki encryption in iot m-health devices," in *IEEE 12th International Conference on Bioinformatics & Bioengineering (BIBE)*, 2012, pp. 25–29.

[5] T. Park, N. Abuzainab, and W. Saad, "Learning how to communicate in the internet of things: Finite resources and heterogeneity," *IEEE Access*, vol. 4, pp. 7063–7073, Nov. 2016.

[6] H. Suo, J. Wan, C. Zou, and J. Liu, "Security in the internet of things: a review," in *International Conference on Computer Science and Electronics Engineering (ICCSEE)*, vol. 3, 2012, pp. 648–651.

[7] W. Saad, X. Zhou, B. Maham, T. Başar, and H. V. Poor, "Tree formation with physical layer security considerations in wireless multi-hop networks," *IEEE Transactions on Wireless Communications*, vol. 11, no. 11, pp. 3980–3991, Nov. 2012.

[8] J.-Y. Lee, W.-C. Lin, and Y.-H. Huang, "A lightweight authentication protocol for internet of things," in *International Symposium on Next-Generation Electronics (ISNE)*. IEEE, 2014, pp. 1–2.

[9] X. Wang, J. Zhang, E. M. Schooler, and M. Ion, "Performance evaluation of attribute-based encryption: Toward data privacy in the iot," in *IEEE International Conference on Communications (ICC)*, 2014, pp. 725–730.

[10] X. Yao, Z. Chen, and Y. Tian, "A lightweight attribute-based encryption scheme for the internet of things," *Future Generation Computer Systems*, vol. 49, pp. 104–112, 2015.

[11] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[12] Y. Sharaf-Dabbagh and W. Saad, "On the authentication of devices in the internet of things," in *Proceedings of 17th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, Coimbra, Portugal, june 2016, pp. 1–3.

[13] S. L. Albuquerque and P. R. Gondim, "Security in cloud-computing-based mobile health," *IT Professional*, vol. 18, no. 3, pp. 37–44, 2016.

[14] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein, "Finding focus in the blur of moving-target techniques," *IEEE Security & Privacy,*, vol. 12, no. 2, pp. 16–26, 2014.

[15] M. Sherburne, R. Marchany, and J. Tront, "Implementing moving target ipv6 defense to secure 6lowpan in the internet of things and smart grid," in *Proceedings of the 9th Annual Cyber and Information Security Research Conference*. ACM, 2014, pp. 37–40.

[16] A. Eldosouky, W. Saad, and D. Niyato, "Single controller stochastic games for optimized moving target defense," in *IEEE International Conference on Communications (ICC)*, 2016, pp. 1–6.

[17] J. Guo, T. Peyrin, A. Poschmann, and M. Robshaw, "The led block cipher," in *Cryptographic Hardware and Embedded Systems–CHES 2011*. Springer, 2011, pp. 326–341.

[18] ARM-Development-Tools. (2016) Arm ds-5 development studio. [Online]. Available: http://ds.arm.com/ds-5/

[19] J. Guo, T. Peyrin, A. Poschmann, and M. Robshaw. (2014) Led reference implementation. [Online]. Available: http://led.crypto.sg/downloads