# SMACK: Short Message Authentication ChecK Against Battery Exhaustion in the Internet of Things

Christian Gehrmann
SICS Swedish ICT AB, Security Lab
Scheelevägen 17, Lund, Sweden
Email: chrisg@sics.se

Marco Tiloca
SICS Swedish ICT AB, Security Lab
Isafjordsgatan 22, Kista, Sweden
Email: marco@sics.se

Rikard Höglund
SICS Swedish ICT AB, Security Lab
Isafjordsgatan 22, Kista, Sweden
Email: rhoglund@sics.se

***Abstract*—Internet of Things (IoT) commonly identifies the upcoming network society where all connectable devices will be able to communicate with one another. In addition, IoT devices are supposed to be directly connected to the Internet, and many of them are likely to be battery powered. Hence, they are particularly vulnerable to Denial of Service (DoS) attacks specifically aimed at quickly draining battery and severely reducing device lifetime. In this paper, we propose SMACK, a security service which efficiently identifies invalid messages early after their reception, by checking a short and lightweight Message Authentication Code (MAC). So doing, further useless processing on invalid messages can be avoided, thus reducing the impact of DoS attacks and preserving battery life. In particular, we provide an adaptation of SMACK for the standard Constrained Application Protocol (CoAP). Finally, we experimentally evaluate SMACK performance through our prototype implementation for the resource constrained CC2538 platform. Our results show that SMACK is efficient and affordable in terms of memory requirements, computing time, and energy consumption.**

## I. Introduction

*Internet of Things* (*IoT*) is commonly used today to refer to the growing information technology trend towards a *networked society*, where all devices that can benefit from a connection will be connected with one another [1][2]. This means that, unlike the past paradigm where mainly computers and mobile phones were globally interconnected over the Internet, now all kinds of electronic equipment are about to be available online. This trend is expected to accelerate in the next years, thanks to the decreasing cost of hardware platforms and network devices, as well as the maturity of the Internet technology.

However, many IoT devices feature limited resources, and are likely to be battery powered. Also, many of them are used in sensitive or even critical tasks, such as industrial plant control and health monitoring application scenarios. Thus, if devices run out of battery power and stop functioning, they can possibly cause severe damage or injuries to people, and result in further costs to perform system recovery, as well as battery replacement or recharge. Hence, it is vital to preserve device battery life as much as possible.

A common *low-level* approach to reduce energy consumption consists in allowing devices to switch to *sleep mode* for a while, in a pre-defined periodic fashion or when no pending tasks have to be accomplished. While in sleep mode, network interfaces can stop their activities, or even be turned off, so considerably reducing energy consumption. On the other hand, traditional *high-level* communication protocols are considered inefficient or too complex to be adopted on simple, resource constrained, IoT devices. This has led to designing the *Constrained Application Protocol* (*CoAP*) [3], a *lightweight* web transfer protocol for use with constrained devices.

However, IoT constrained devices are particularly prone to a number of security threats, especially if *directly* connected to the Internet, i.e. with no protection provided by a firewall or gateway. In particular, possible security attacks include *Denial of Service* (*DoS*), which may have the specific intent to prevent devices from switching to sleep mode, thus quickly draining battery energy. Such attacks are often referred as *Denial of Sleep* attacks, and represent a real and severe threat against small battery powered devices [4][5][6]. Moreover, the occurrence of DoS cannot be avoided altogether. In fact, it is sufficient that an adversary keeps on transmitting bogus messages to force the recipient device to process them. Instead, a feasible and practical countermeasure would consist in reducing the attack impact as much as possible, with particular reference to energy consumption due to useless message processing.

Counteracting such attacks is definitely not easy, especially for IoT devices directly connected to the Internet and globally accessible for operation requests, as well as remote management and configuration. Although CoAP suggests to secure communication by means of the *Datagram Transport Layer Security* (*DTLS*) protocol [7], the latter relies on a complex and costly handshake to establish a secure connection. Moreover, DTLS handshake itself is vulnerable to DoS battery drain attacks. In fact, an adversary can flood the target device with fake *ClientHello* messages, thus repeatedly triggering costly operations on the device side. Most alternative solutions against battery exhaustion rely on intrusion detection techniques, or traditional link layer security. However, the formers normally require to collect and analyze a large amount of network traffic, so being unwieldy or even unfeasible to be adopted [4], while the latters are likely to display a non negligible impact on performance, especially in terms of energy consumption [8][9].

This paper presents *SMACK*, a security service based on *short Message Authentication Codes* (*MACs*), which *early* and *efficiently* detects invalid messages upon their reception, thus considerably reducing the impact of Denial of Sleep attacks. In particular, by checking short MAC correctness, it is possible

This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme. The research leading to these results has received funding from the *European Union* Seventh Framework Programme (*FP7/2007-2013*) under *grant agreement* n° 246016.

978-1-4673-7331-9/15/$31.00 © 2015 IEEE

to quickly verify whether received messages are valid or not, i.e. if they have been possibly sent by illegitimate sources. In such a case, devices can avoid performing useless additional parsing and processing, so considerably reducing the damage that an attacker can cause in terms of energy consumption, and thus preserving battery life. Also, we suggest a tunable and adaptive *reaction* mechanism that can be adopted upon detecting invalid messages, so specifically addressing Denial of Sleep attacks, and forestalling battery exhaustion.

We provide an adaptation of SMACK for the CoAP protocol that does not require to modify the original message format. Nevertheless, SMACK is based on a general approach, and can be integrated in other protocols at any communication layer. Besides, we experimentally evaluate SMACK performance through our prototype implementation for the resource constrained CC2538 platform. Our results show that SMACK is efficient and affordable in terms of memory requirements, computing time, and energy consumption. Also, when compared to DTLS, our approach detects invalid messages while displaying a smaller computing overhead, and requiring no additional communication with the constrained device.

The rest of the paper is organized as follows. In Section II, we discuss related works. Section III describes the considered application scenario, while Section IV overviews the CoAP protocol. Section V presents SMACK and how it can be adapted to CoAP, while in Section VI we propose a reactive strategy to address DoS attacks upon their occurrence. Section VII discusses a possible way to perform the short MAC computation, while in Section VIII we show how the administrative key material can be generated. Section IX describes a possible way to adapt the generation of CoAP Message IDs, while in Section X we provide a performance evaluation of SMACK. Finally, in Section XI, we draw our conclusive remarks.

## II. Related work

Battery exhaustion has been considered a real and severe threat for a long time now. In 1999, Stajano and Anderson suggested that a malicious node could perform a *sleep deprivation torture* attack, i.e. behave with the only purpose to consume the victim's battery energy [5]. Then, a number of relevant works have stressed the severity of Denial of Sleep attacks, both in Wireless Sensor Networks (WSNs) and in general mobile computer environments [4][6]. More recently, even energy attacks on server systems have been considered as a possible serious threat to deal with [10].

In [6], the authors identify three different forms of sleep deprivation attack: *service request*; *benign*; and *malignant* power attacks. In service request attacks, the adversary repeatedly performs valid service requests to the victim device, to drain its residual energy. In a benign attack, she induces her victim to indefinitely perform a valid but energy-hungry process. In a malignant power attack, she compromises devices and alters their original behavior to increase energy consumption.

Several works about detection techniques against Denial of Sleep attacks have been presented, often based on traffic analysis or other anomaly detection mechanisms. In [11], the authors describe *B-SIPS*, a system based on an innovative Dynamic Threshold Calculation algorithm that alerts upon detecting power changes. Specifically, it considers running activities, and compares their energy consumption with the expected one, looking for possible anomalies. In [12], Bhattasali and Chaki propose a probabilistic model to detect Denial of Sleep attacks, based on Absorbing Markov Chain (AMC). In particular, they associate the Markov Chain absorption time with the network lifetime, and assume that a Denial of Sleep attack is being performed in case the system state converges to network death too fast with respect to the *expected* battery discharging trend. Unfortunately, as highlighted in [4], such solutions based on intrusion detection techniques normally require to capture and analyze a large amount of network traffic, which can be difficult or even unfeasible for many resource constrained IoT devices.

Other presented approaches rely on traditional security services, especially at the link layer. In [6], the authors propose a *power-secure* architecture, relying on multi-layer authentication and energy signature monitoring. It aims at processing only requests that deserve a high level of trust, and identifying intrusions that would result in the execution of energy-hungry services. However, apart from the general suggestions, no concrete schemes to achieve battery drain robustness have been proposed in [6]. Nonetheless, it is claimed there is a real need for an *early* and *lightweight* message authentication check, in order to withstand battery drain attacks. In [13], Raymond and Midkiff present *CARL*, a rate limiting approach based on current host-based intrusion detection techniques. Specifically, CARL performs attack detection at the link layer, considering messages to be legitimate only if both authenticated and not replayed. Then, the rate of malicious traffic is determined by comparing maintained information on recent messages, and actions based on *rate limiting* are taken to mitigate the attack effects, trading network lifetime with network throughput. In [4], Raymond *et al.* consider different attacks against a number of link layer protocols, and prove that Denial of Sleep attacks can be easily performed, with no need to break link layer encryption. In order to mitigate the impact of such attacks, they propose a framework based on link layer authentication, replay protection, and jamming identification.

The above mentioned detection approaches based on traditional security services are likely to result in a not negligible overhead, especially in terms of energy consumption [8]. Also, in case security services at the link layer are adopted, message validity must be checked by *every* network node in the path between the source and destination endpoint. Of course, this results in a further non negligible impact on network performance as a whole. Moreover, it requires to establish secure trust relationships among all involved nodes in the communication path (e.g. through pairwise cryptographic keys), thus likely failing in *efficiently* providing protection against Denial of Sleep attacks from an end-to-end standpoint.

Unlike the solutions mentioned above, SMACK is a *proactive* security service that *promptly* and *efficiently* recognizes invalid (and likely hostile) messages upon their reception, in order to detect Denial of Sleep attacks. It has been designed to address mainly service-requesting attacks, but represents a valid countermeasure also against benign attacks [6]. As described in Section V, SMACK relies on a short and lightweight *Message Authentication Code* (*MAC*) in order to check validity of messages early after their reception, and avoid any additional and unnecessary parsing efforts on non valid ones.

## III. Application scenario and threat model

We consider an application scenario where an adversary can easily perform a *Denial of Sleep* attack against a resource constrained device. With particular reference to a *service request* attack [6], the adversary can repeatedly send request messages to the device relying on unicast or multicast communication, so inducing it to continuously parse and process them upon their reception. Hereafter, we refer to such requests sent by the adversary as *invalid messages*. Furthermore, the considered attack would prevent the device from possibly switching to *sleep mode*, i.e. it would be unable to stop its network interface activities in order to reduce power consumption. Also, the device might be induced to execute operations requested by the adversary, with additional damage in terms of service availability and energy consumption.

As a consequence, as well as affecting network performance and application availability, the considered attack severely impacts on the device energy consumption, thus drastically reducing its battery lifetime. Thus, we believe it is vital to *distinguish* between valid and non valid request messages as soon as possible once they have been received by the device, as we further discuss in Section VI.

However, at the same time, we would like to avoid that only a single entity has access right to the constrained device, which must actually be available to all *authorized* clients having such a permission. A natural way to address this is considering a model where a client, *before* requesting a service to the device, needs to obtain an *authorization* to perform such a request. Similarly to the model described in [14], we assume that such decisions are taken by a dedicated centralized entity. By offloading the whole authorization procedure to a central entity, a lot of computational overhead can be avoided on the constrained device side. In particular, we assume that the implemented policies allow the central entity to effectively determine whether to issue an authorization to a requesting client. In the considered scenario, we rely on such a central entity as a pure key manager, i.e. a *Key Distribution Center* (*KDC*), but it can be utilized also as a more general authorization server as described in [14]. Note that, in practice, the *KDC* can be implemented either as a real centralized entity or according to a distributed architecture. With respect to an approach based on proxies, this model has the advantage that the actual client and server communication at the end user application does not require any particular adaptations.

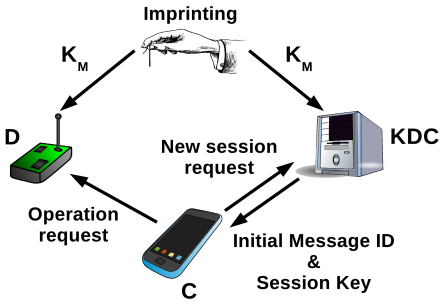


Fig. 1. Application scenario.

In the rest of this paper, we refer to the application scenario depicted in Figure 1, and consider the presence of three distinct entities, namely an IoT *device D*, a *client C*, and the *KDC*.

We assume that device *D* is associated to the *KDC*, and that they are in a mutual trust relationship. Also, *D* and the *KDC* share a symmetric cryptographic master key $K_M$, which is not directly used for secure communication, but only to generate other security material, as described in Section VIII. In the following, we assume that $K_M$ has been established before device deployment, during the initial configuration phase of device *D*, also referred as *imprinting* [5]. Note that, in real IoT scenarios, the *KDC* would reasonably provide also other services, such as the authorization procedure mentioned above. Hence, key management does not practically require to introduce a dedicated key infrastructure.

In order to communicate with one another, the involved entities $C$, $D$, and the $KDC$ rely on the *Constrained Application Protocol* (*CoAP*) [3]. Communication can be possibly secured, i.e. exchanged messages may be protected to assure that application security requirements are met. To this aim, the CoAP specifications recommend to adopt the DTLS protocol. Further details about specific cryptographic means adopted to secure communication are out of the scope of this paper.

Before starting to communicate with device *D*, client *C* first contacts the *KDC*, in order to obtain: i) a cryptographic *session key* used by SMACK to process messages addressed to *D*; and ii) a valid *Message ID* to be used for the first one of such messages. We provide more details about management of session keys and Message IDs in Sections VIII and IX.

## IV. CoAP overview

*Constrained Application Protocol* (*CoAP*) [3] is a standard web transfer protocol, for use with constrained devices and networks (e.g. low-power and lossy). It is intended especially for *machine-to-machine* (*M2M*) applications, such as smart energy and building automation, and runs over UDP, thus not assuming reliable message transport. Also, it is not session-based and can handle loss or delayed delivery of messages.

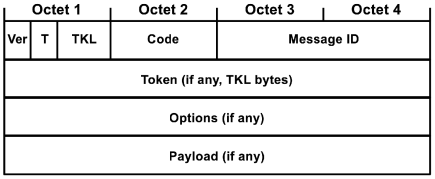


Fig. 2. CoAP message format.

Figure 2 depicts the structure of a CoAP message. The first byte contains the protocol version *Ver*, a type field *T* with basic message type information, and the size in bytes of the *Token* field, namely *TKL*. The *Code* field contains more precise message type information, while the *Message ID* field is a unique ID used to track messages and detect possible duplications. The optional *Token* field can be used to match request and response messages, and its values should be generated at random as well as uniquely for each request. The field ranges between 0 and 8 bytes in size, aims at making CoAP more robust against IP-spoofing attacks, and its usage is strongly recommended in case security is not provided at the transport layer. Furthermore, several different CoAP *options* have been defined, and it is possible to specify a list of them according to a *Type-Length-Content* scheme. Finally, the CoAP message content is included in the *Payload* field.

So far, no specific security related procedures have been defined for the CoAP protocol. Instead, it is recommended to adopt DTLS [7] if authentication, integrity, or confidentiality are required. However, using DTLS to provide protection against battery drain attacks might not be the best solution, since it requires the execution of a complex handshake procedure, which is itself a considerable target for DoS attacks.

## V. Message validity check

If we consider resource constrained, typically battery powered, devices and their need to limit energy consumption, then it would be better if they could *early* and *easily* check if received messages come from legitimate sources or not. Such a procedure should be *compatible* with the adopted communication protocol stack, and should allow for detecting invalid messages as early as possible upon their reception. Also, it should be as *lightweight* as possible, i.e. *efficient* in terms of computing overhead and resulting energy consumption.

Early detection of message validity for battery drain prevention is applicable to many communication scenarios. How to actually use such an approach strictly depends on the considered protocols and communication technologies. However, we believe it is interesting to address battery drain prevention with particular reference to constrained embedded devices. Hence, we chose to design our approach so that it is primarily tailored to CoAP. That is, in the rest of this paper, we assume that SMACK is adopted at the application level, and relies on the CoAP protocol. Nevertheless, SMACK is general, and can be potentially adopted at any layer in the communication stack, by introducing a short MAC value in the regular message header.

More in detail, SMACK relies on early checking a short MAC, thus *promptly* asserting if a given message can be considered valid or not upon its reception. So doing, in case of invalid (possibly forged) received messages, it is possible to avoid further parsing and verification procedures, thus saving a consistent amount of energy, and extending battery life. Also, if several invalid messages are received in a very short time interval, the recipient device may assume that a DoS attack is being performed, and can adopt reactive countermeasures in order to limit its impact (see Section VI).

The goal of our approach is to provide a good protection against battery drain attacks, while displaying very low complexity as well as an affordable computing and energy overhead. To this end, we rely on short *Message Authentication Codes* (*MACs*), computed by means of security material derived from the shared master session key $K_M$, and refer to unconditionally secure MACs [15], which, even if small in size, provide a very high level of protection. We recall that, unlike traditional message integrity and authenticity checks, the main purpose of SMACK is early checking the *validity* of CoAP messages, in order to detect (likely) ongoing Denial of Sleep attacks, and reduce their impact on energy wasting.

Given its format and position within the CoAP message, we believe it is perfect to rely on the *Token* field to include the short MAC, as a validity check information. Specifically, upon finalizing an outgoing CoAP request message, the sender client produces a 16 bit random *Request ID* $R$. Such a value is used to actually correlate the request message with its associated response, which is the main purpose of the CoAP *Token* field

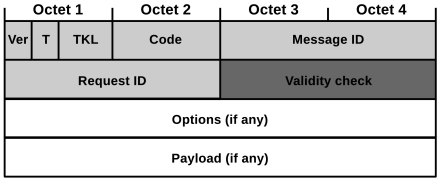


Fig. 3. CoAP message including short MAC.

[3]. Further details about how $R$ is generated are out of the scope of this paper. Then, SMACK computes a 16 bit *short* MAC $SM$, according to the procedure described in Section VII, which takes the concatenation of the following pieces of information as a 6 byte input: i) the 4 byte CoAP header, i.e. the *Ver*, *T*, *TKL*, *Code* and *Message ID* fields; and ii) the 2 byte Request ID $R$. Finally, both $R$ and $SM$ are included within the *Token* field of the CoAP message. Specifically, as depicted in Figure 3, the Request ID $R$ and the short MAC $SM$ are carried in the *Request ID* and the *Validity check* subfield of the *Token* field, respectively. Note that, since the short MAC computation can be considered a pseudo random function, SMACK assures that the *Token* field as a whole always carries a random value.

## VI. Reactive strategies

Together with the message validity check based on short MAC, device $D$ can rely on an additional strategy, aimed at *reacting* in case too many invalid messages have being received. In fact, while it is reasonable to accept occasional invalid messages, due to the lack of reliability in adopted communication protocols, repeated errors and frequently failed validity checks should be treated as a deliberate DoS attack.

Let us assume that, for each active session $S$, device $D$ maintains two counters, namely $CounterError_S$ ($CE_S$) and $CounterAccepted_S$ ($CA_S$), initialized as $CE_S = CA_S = 1$ upon the establishment of session $S$. Also, $D$ maintains two *global* predefined threshold values, i.e. $TR_E$ and $TR_A$. Then, as a possible reactive strategy, device $D$ performs the following actions upon receiving a message $M$ during a session $S$.

**Valid messages**. If message $M$ conveys a *valid* short MAC, then $CA_S = CA_S + 1$. After that, if $CA_S = TR_A$, then $CA_S$ is reinitialized to 1, and $CE_S = max(1, CE_S - 1)$.

**Invalid messages**. If message $M$ conveys an *invalid* short MAC and $CE_S$ is lower than threshold $TR_E$, i.e. $CE_S < TR_E$, then $CE_S = CE_S + 1$. Instead, if $CE_S = TR_E$, device $D$ assumes that a DoS attack is occurring, and stops any network activity related to session $S$, for a predefined time interval $t_S$. For instance, it can terminate session $S$, or even turn off the network interface. Once the time interval $t_S$ has elapsed, counters $CE_S$ and $CA_S$ are reinitialized to 1, and new messages are accepted within session $S$.

Note that the mechanism described above can be further enhanced, by making the device more sensitive and *suspicious* about DoS attack detection. For instance, in case $CE_S$ goes over threshold $TR_E$ after a predefined short time or short amount of received messages, then device $D$ can decrease $TR_E$ by a fixed value. As a result, the adopted reaction policy against DoS would be triggered after a minor number of invalid messages has been received. Further details about adaptive

detection and reaction policies are out of the scope of this paper, and are left for further investigation.

## VII. Short MAC computation

In this section, we describe how SMACK computes short MACs introduced in Section V. Note that the procedure we suggest here is just one among several possible ways to perform the short MAC computation. However, since the resulting output is an unconditionally secure MAC [16], and algorithms used to produce it have been extensively studied in terms of correctness and complexity, we strongly believe that the suggested method is close to be optimal in terms of simplicity and computational efficiency.

Having defined the short MAC to be 16 bits in size, we assume that elements involved in the short MAC computation are 16 bits in size as well. This is only for the sake of simplicity in the following description, while it is clearly possible to rely on elements of different sizes. Since CoAP does not include a session concept [3], in the following we refer to a *session* as a sequence of messages exchanged between the same client $C$ and device $D$ over a short amount of time.

We denote $GF(2^{16})$ as a Galois field with a size of 16 bits [17], and define: i) $a, b \in GF(2^{16})$ as 16 bit key values, unchanged for an entire session; and ii) $c_i \in GF(2^{16})$ as a 16 bit key value, updated for every new message $M_i$ exchanged in the same session.

More specifically, let $M_i$ be the $i$-th CoAP message sent from client $C$ to device $D$ in a given session[1]. Then, let $m_i$ be the first 48 bits in message $M_i$, i.e. the 32 bit CoAP header together with the 16 bit *Request ID* subfield of the *Token* field. We represent $m_i$ as the concatenation of three 16 bit elements, i.e. $m_i = \{m_{i_0}||m_{i_1}||m_{i_2}\}$, where $m_{i_0}, m_{i_1}, m_{i_2} \in GF(2^{16})$, and compute the short MAC $v_i$ associated to message $M_i$ as

$$v_i = (m_{i_0} + a \cdot m_{i_1} + a^2 \cdot m_{i_2}) \cdot b + c_i \tag{1}$$

Since $a^2$ can be precalculated and it does not change throughout the whole session, computing a short MAC $v_i$ requires only three additions and three multiplications[2] in $GF(2^{16})$, thus resulting to be extremely efficient.

Before sending message $M_i$ to device $D$, client $C$ computes the short MAC $v_i$ according to Equation 1, and writes it into the *Validity check* subfield of the *Token* field. Then, upon the reception of $M_i$, device $D$ considers the first 6 bytes of the message, and provides them as input to Equation 1, in order to compute the short MAC $v_i^*$. Finally, $D$ retrieves the short MAC $v_i$ carried within the received message, and compares it with $v_i^*$. In case of a positive match, $D$ considers the message to be *valid*, i.e. it assumes that the message has been sent by a legitimate source. Further, more in-depth, checks to verify the authenticity of message $M_i$ can then possibly take place, according to other security mechanisms provided by the application or, in general, by the adopted communication stack.

As a final note, even though the process in Equation 1 displays a very small computational effort, short MACs produced as output still display a low probability to be forged by an external adversary, as we prove in the following theorem.

*Proposition 1:* Given that the key values $a, b$, and $c_i$ in Equation 1 are truly random, a successful forgery attack against a short MAC produced by SMACK occurs with probability equal to $2^{-15}$.

*Proof:* It follows from Construction 2 and Theorem 2.3 in [16] that, given a block code $E$ with parameters $(n, M, d)$ and a random selected key $k = \{a, b, c\}$, $a, b, c \in GF(q)$, then the MAC tag $v$ for message $\overline{M}$, $v \in GF(q)$, is calculated as

$$v = be_a(\overline{M}) + c \tag{2}$$

where $e_a(\overline{M})$ denotes the $a$-th code symbol in $GF(q)$ for message $\overline{M}$, and has a probability of successful impersonation or substitution attack which is independent of the computing power of the attacker, and is equal to

$$\max(1/q, 1 - d/n) \tag{3}$$

The short MAC computation we show in Equation 1 is based on the block code construction in Equation 2, with reference to a Reed-Solomon code [18] over $GF(2^{16})$, and parameters $(n, M, d) = (2^{16}, 2^{48}, 2^{16} - 2)$. Therefore, a successful forgery attack against a short MAC produced by SMACK occurs with probability equal to

$$\max(2^{-16}, 1 - (2^{16} - 2)/2^{16}) = 2^{-15} \tag{4}$$

■

## VIII. Key derivation procedure

In this section, we describe a possible approach to derive the security keys $a$, $b$, and $c_i$ used for the short MAC computation (see Section VII). We recall that all the three keys are 16 bits in size.

With reference to the application scenario depicted in Figure 1, we denote by $K_M$ the *master key* shared between the device $D$ and the Key Distribution Center $KDC$. We recall that the establishment of $K_M$ is part of the device *imprinting* process, that we assume to take place at deployment time. Also, $K_M$ is not directly used to secure communication, but only to produce further key material. Finally, $D$ and the $KDC$ store a pre-shared randomly generated number of sufficient length (e.g. 256 bits), namely *seed*, which is used together with $K_M$ to generate additional security material.

The proposed key derivation procedure relies on a suitable *pseudo random function* $PRF(\cdot)$ [19] commonly agreed by both the $KDC$ and $D$, which we assume to produce a 256 bit output. There are many available and well known pseudo random functions, including *HMAC* functions [20] based on the MD5, SHA1, and SHA2 hash functions. Although some of them display a relatively high computational complexity, our key derivation procedure does not require to invoke $PRF(\cdot)$ as often as SMACK performs the short MAC computation, thus limiting the resulting processing overhead.

[1] Since we are interested in protecting the *device* from battery drain attacks, we focus only on messages from the client to the device, and do *not* consider the opposite communication direction.

[2] In case special hardware for field calculations is not available, it is possible to adopt an even more efficient scheme, where a prime field $GF(p)$, $p > 2^{16}$, is used instead of a $GF(2^{16})$ field. So doing, only simple and extremely efficient $mod\ p$ operations are necessary. However, this means that *slightly* larger keys, i.e. 17 instead of 16 bits in size, are needed for the inner operations, and the key consumption is consequently slightly larger as well.

After device deployment, both $D$ and the $KDC$ uses $K_M$ and $seed$ to derive a *master session key* $K_{MS}$ as follows.

$$K_{MS} = PRF(K_M, seed) \quad (5)$$

Hereafter, $K_{MS}$ is used on the client (device) side to produce the actual key used to compute (verify) short MACs conveyed by CoAP messages. Also, $K_{MS}$ needs to be regularly renewed through an update procedure, and securely provided to device $D$. Possible schemes according to which such a key redistribution actually takes place are out of the scope of this paper.

When contacted by $C$, the $KDC$ replies providing it with: i) the *initial Message ID* $ID^*$ to be used in the *first* CoAP message addressed to $D$ in this session; and ii) a *session key* $K_S$, which is valid for the current session only, and is computed as

$$K_S = PRF(K_{MS}, ID^*) \quad (6)$$

If we define $K_{S_j}$ as

$$K_{S_j} = \begin{cases} K_S & \text{if } j = 0 \\ PRF(K_S, K_{S_{j-1}}) & \text{if } j \geq 1 \end{cases} \quad (7)$$

then the short MAC $v_i$ for the $i$-th message in the current session, can be computed according to Equation 1, providing the following information as input[3]: i) $a$ coincides with the $K_S$ bits ranging from position 0 to bit in position 15; ii) $b$ coincides with the $K_S$ bits ranging from position 16 to bit in position 31; and iii) $c_i$ coincides with the 16 bits in $K_{S_j}$, where $j = \lfloor (i+2)/16 \rfloor$, ranging from position $((i+2)\ mod\ 16) * 16$ to position $((i+2)\ mod\ 16) * 16 + 15$.

We discuss a possible method to generate initial Message IDs in Section IX. Note that, since CoAP Message IDs are only 16 bits in size, it may not take a long time before the $KDC$ generates an already used initial Message ID, regardless of whether the very same client $C$, or a different one, is involved. Thus, before providing a client with an initial Message ID already used together with the current $K_{MS}$, the latter must be renewed and securely provided to $D$.

## IX. Message ID management

In this section, we describe how new initial Message IDs can be generated and managed, and how device $D$ is supposed to process them upon receiving CoAP messages.

### A. Message ID generation

As described in Section VIII, SMACK relies on a session key $K_S$ to process CoAP messages within the same session. However, in order to avoid replay attacks, it must be assured that a different *fresh* session key is used at every session. Thus, the adopted Message ID generation strategy must be such that the same initial Message ID is *not* reused together with the same master session key $K_{MS}$. In the following, we propose a scheme where the $KDC$ generates *fresh* initial Message IDs.

First, client $C$ contacts the $KDC$, and requests to establish a new session $S_i$ with device $D$. Then, the $KDC$ computes the new initial Message ID $ID_i^*$ as $ID_i^* = (ID_{i-1}^* + p)\ mod\ 2^{16}$, where $ID_{i-1}^*$ is the initial Message ID associated to the previous session $S_{i-1}$, and $p > 2$ is a suitable *prime* number, of the order of 100, pre-shared by the $KDC$ and $D$. Once the new initial Message ID $ID_i^*$ has been determined, it is used to generate a new session key $K_S$ according to Equation 6.

After that, the $KDC$ provides client $C$ with $K_S$ and $ID_i^*$. Then, $C$ starts to communicate with device $D$, using $ID_i^*$ as the Message ID of the first sent CoAP message. Given the key generation scheme described in Section VIII, after having sent the very first message of a new session, the client $C$ *must* receive a confirmation of correct reception on the device side. That is, client $C$ can transmit further messages in the same session only after such a confirmation has been received.

Since the Message ID value is incremented by 1 every time a different message is exchanged between $C$ and $D$ [3], then a new session *must* be established, i.e. a new initial Message ID must be generated, when: i) the same client $C$ has sent $p$ messages to $D$ in the same session; or ii) a different client $C'$ contacts the $KDC$ and requests to communicate with $D$.

The proposed scheme makes it possible to use the whole Message ID space without duplication, so that different sessions are associated with different session keys $K_S$. However, although it might take a practically long amount of time, the Message ID generation process described above eventually gets "exhausted", once the $KDC$ has used all the $2^{16}$ available initial Message ID values. When this happens, a new master session key $K_{MS}^+$ must be established between $D$ and the $KDC$. This can be done by *securely* exchanging a new *seed* value between $D$ and the $KDC$, every time the latter generates a new initial Message ID already used together with the current master session key $K_{MS}$. Once received the new seed value, $D$ can locally and securely compute $K_{MS}^+$, and use it to generate future session keys. Further details about methods to distribute the new seed value are out of the scope of this paper.

### B. Message ID processing

Upon receiving a CoAP message, device $D$ must check if the conveyed Message ID is fresh, in order to prevent replay attacks. However, since CoAP is a connectionless protocol, it is possible that messages are received in an arbitrary order. Hence, we propose a *sliding window* approach to handle messages that can possibly arrive out of order.

| Name | Content |
|---|---|
| $A$ | Predefined acceptance windows size, i.e. maximum number of active sessions |
| $T$ | Predefined window step, i.e. number of sessions skipped on acceptance window update ($0 < T < A$) |
| $ID_C$ | Acceptance window indicator, i.e. the initial Message ID of the oldest active session |
| $\widehat{ID}_C$ | Set of initial Message IDs related to sessions active in the current acceptance window |
| $ID_F$ | First initial Message ID in the current session epoch |
| $ID_L$ | Last initial Message ID in the current session epoch |

TABLE I. Device state information.

Device $D$ maintains the state information reported in Table I. We assume that, during the imprinting process, $D$ has been provided with a valid value for $ID_F$, i.e. the initial Message

[3]Once the initial Message ID $ID^*$ has been received, the device can precalculate all the session key material, hence speeding up short MAC computations upon the reception of future messages.

ID to be used by the $KDC$ for the first session involving device $D$. Also, for every currently active session $S$ with initial Message ID $ID^*$, $D$ maintains a set $\widehat{R}_{ID^*}$ containing Message IDs of all valid messages belonging to $S$ received so far. This makes it possible to promptly detect retransmissions of old *authentic* messages, hence preventing possible replay attacks.

The following procedure describes how device $D$ evaluates Message IDs upon message reception.

**Epoch startup**. Device $D$ initializes $\widehat{ID_C}$ as $\widehat{ID_C} = \emptyset$, and accepts *any* Message $M$ conveying a Message ID $ID_M$ s.t. $ID_M = (ID_F + m \cdot p) \ mod \ 2^{16}$, $m \in \mathbb{N}, 0 < m < A$. Then, $D$ derives the session key $K_S$ according to Equation 6, i.e. $K_S = PRF(K_{MS}, ID_M)$, and verifies $M$ validity by checking its short MAC, as described in Section VII. In case $M$ is found to be a valid message, $D$ initializes $ID_C$ as $ID_C = ID_M$, and updates $\widehat{ID_C}$ as $\widehat{ID_C} \leftarrow \widehat{ID_C} \cup \{ID_M\}$. Also, it determines the last valid initial Message ID $ID_L$ in the current session *epoch*[4], s.t. $(ID_L + A \cdot p) \ mod \ 2^{16} = ID_C$. Finally, $ID_M$ is stored in the $\widehat{R}_{ID_M}$ associated to the just started session $S$.

Then, for every new message $M$ with Message ID $ID_M$ received by device $D$, the following actions are performed.

**Case 1: New session message**. If Message ID $ID_M = (ID_C + m \cdot p) \ mod \ 2^{16}$, $m \in \mathbb{N}, 0 < m < A$, and $ID_M \notin \widehat{ID_C}$ (i.e. $M$ is not a replay), then $D$ computes the session key $K_S$ as $K_S = PRF(K_{MS}, ID_M)$, and checks the short MAC conveyed in $M$. If $M$ results to be valid, it is considered as the first message of a new session $S$, and $D$ updates $\widehat{ID_C}$ as $\widehat{ID_C} \leftarrow \widehat{ID_C} \cup \{ID_M\}$. Then, $ID_M$ is stored in the set $\widehat{R}_{ID_M}$ associated to the new session $S$. Then, if $|\widehat{ID_C}| = A$, device $D$: i) removes the $T$ elements related to the $T$ "oldest" active sessions from $\widehat{ID_C}$, i.e. it moves the acceptance window forward by $T$ positions; ii) removes the $T$ sets $\widehat{R}$ associated to such sessions; and iii) assigns $ID_C$ to the element in $\widehat{ID_C}$ related to the oldest active session.
**Case 2: Active session message**. If $\exists \ \overline{ID} \in \widehat{ID_C}$, s.t. $\overline{ID} < ID_M < (\overline{ID} + p) \ mod \ 2^{16}$, $ID_M \notin \widehat{R}_{\overline{ID}}$ (i.e. $M$ is not a replay), and the short MAC check is passed, then $D$ stores $ID_M$ in the set $\widehat{R}_{\overline{ID}}$, and accepts $M$ as a message of an active session within the current acceptance window.
**Case 3: Invalid message**. If none of the above conditions holds, then $M$ is discarded and no further actions are taken.

Finally, once $2^{16} - (A - 1)$ sessions have been authorized by the $KDC$, the latter renews the master session key $K_{MS}$ in order to prevent its reusage. Then, the $KDC$ provides device $D$ with the new master session key $K^+_{MS}$ and the first initial Message ID of the next session *epoch*, i.e. $\overline{ID}_F$. On the other hand, when a message $M$ whose Message ID $ID_M = ID_F$ is received, and it is accepted as described in *Case 1* above, then $D$ starts processing only messages associated to already active sessions, and stops from establishing any further new sessions. Then, once it has been provided with the new master session key $K^+_{MS}$ and $\overline{ID}_F$, $D$ initializes $ID_F$ as $ID_F = \overline{ID}_F$, and moves back to the *Epoch startup* phase described above, hence initiating the new session *epoch*.

[4] $ID_L$ can be computed as $ID_L = ID_C - A \cdot p$ if $ID_C \geq A \cdot p$, or $ID_L = 2^{16} - (A \cdot p - ID_C)$ if $ID_C < A \cdot p$.

## X. Performance evaluation

To evaluate SMACK, we developed a prototype implementation for resource constrained platforms. In this section, we show that, even with no particular optimizations, our implementation displays an affordable overhead in terms of memory requirements, computing time and energy consumption. Also, when compared to DTLS, our approach detects invalid messages while displaying a smaller computing overhead and requiring no additional communication with device $D$.

For our tests, we considered a commodity PC as CoAP client $C$, featuring 16 GB of RAM and an Intel i5-3470 CPU, and running the Java SE runtime environment. The client ran our extended version of the Java library Californium, which provides both the CoAP and DTLS protocols [21]. As CoAP server $D$, we considered a resource constrained device based on the CC2538 platform [22], featuring a 32 MHz CPU, and provided with 512 KB and 32 KB of flash memory and RAM, respectively. The server ran our extended version of CoAP provided by Contiki and based on the *Erbium* library [23]. Also, we referred to the *tinydtls* library implementing the DTLS protocol for resource constrained platforms [24]. For simplicity, we considered the initial message ID and session key $K_S$ as pre-loaded on $C$. To enable communication between $C$ and $D$, we relied on an additional router node based on the same CC2538 platform and running the Contiki operating system. Finally, we used the *Energest* framework provided by Contiki to collect time measurements on device $D$. Energest has been proven to accurately estimate energy consumption and increase the computing time only of the $0.7\%$ [25].

Compared to the original CoAP protocol provided by Contiki, our implementation of SMACK, even without any particular optimizations, results in additional 1.65 KB ($+1.01\%$) of memory on device $D$, including also an H-MAC function (i.e. the same one adopted by DTLS [7]). In contrast, providing support for DTLS through the *tinydtls* library results in additional 37.39 KB ($+22.71\%$) of memory on device $D$.

We ran a set of experimental tests on a local network, considering a system not running any security services and a system relying on DTLS. More in detail, we first ran our test environment in the presence of CoAP only. Then, we relied on the very same set-up, with the additional presence of DTLS to secure communication. Finally, we ran the very same experiments considering CoAP together with SMACK only, without the presence of DTLS. This set of experiments allows us to compare overheads in terms of increase in the CoAP message exchange length, and additional computing time and energy consumption, with respect to a system including either only CoAP or both CoAP and DLTS. All results have been averaged over 20 independent replications, and confidence intervals have been derived, with 95% confidence level.

To keep this comparison as fair as possible, we considered the DTLS cryptosuite TLS_PSK_WITH_AES_128_CCM_8 [26], which is accounted as suitable to compact implementations and constrained devices. In fact, it is based on a pre-shared key handshake, hence avoiding more resource consuming operations based on public keys. Also, although it provides both confidentiality and data origin authentication, it is based on a single *authenticated encryption* operation.

A single test consists of 10 CoAP message exchanges,

namely *transactions*, between $C$ and $D$. More in detail, firstly $C$ transmits a CoAP request message $M1$ to $D$, whose total size is 15 bytes. Once received and processed it, $D$ replies back to $C$, sending a CoAP response message $M2$, whose total size is 24 bytes. We separately present results referred to either the first transaction or the last transaction, so distinguishing overheads occurring either at communication startup, or when communication has reached a steady state. The rest of this section refers to the following three test cases.

**1. CoAP_ONLY** considers the original CoAP protocol only, with no additional security services.
**2. CoAP_SMACK** considers the CoAP protocol together with SMACK, without the presence of DTLS. Since only CoAP request messages carry a short MAC, every transaction results in 1 short MAC computation on the client side, and 1 short MAC check on the server side. The key generation procedure described in Section VIII relies on a $PRF(\cdot)$ function based on the SHA-256 hash function. Note that no additional communication occurs with device $D$.
**3. CoAP_DTLS** considers the CoAP protocol together with DTLS. Every transaction results in: i) 4 security operations, i.e. 1 encryption and 1 decryption, both on the client and server side; and ii) the transmission of 42 additional bytes, due to the presence of the 13 byte DTLS header and the 8 byte DTLS Message Authentication Code (MAC). Note that the first transaction comprises the execution of the DTLS handshake.

First, we evaluate the *transaction length TL* from the client standpoint. Depending on the specific test case, the TL is defined as $(t_1^e - t_1^b)$ for *CoAP_ONLY*, $(t_2^e - t_2^b)$ for *CoAP_SMACK*, or $(t_3^e - t_3^b)$ for *CoAP_DTLS*. Times $t_1^b$ and $t_1^e$ refer to when $C$ sends $M1$ and receives $M2$, respectively. Times $t_2^b$ and $t_2^e$ refer to when $C$ starts computing the short MAC on $M1$ and receives $M2$, respectively. Time $t_3^b$ refers to when $C$ starts a DTLS handshake, if needed, or begins securing $M1$, while time $t_3^e$ refers to when $C$ finishes to unsecure $M2$.

| | CoAP_ONLY | CoAP_SMACK | CoAP_DTLS |
|---|---|---|---|
| **First transaction** | 35.930 ms ±0.831 ms | 40.941 ms ±1.070 ms | 385.688 ms ±25.223 ms |
| **Last transaction** | 32.790 ms ±1.365 ms | 34.840 ms ±2.428 ms | 39.062 ms ±0.761 ms |

TABLE II. TRANSACTION LENGTH.

Table II shows the TL values collected on the client side. With respect to the *CoAP_ONLY* test case, we can observe that the TL slightly increases in the presence of SMACK, and further increases in the presence of DTLS. As expected, such increments become smaller when communication has reached a steady state. On the other hand, if we consider the first transaction, DTLS results in a TL which is roughly 10 times greater than in the *CoAP_ONLY* and *CoAP_SMACK* test cases. This is mainly due to the execution of the complex DTLS handshake, upon establishing a new DTLS session. Such results are consistent with the fact that, with respect to SMACK, the DTLS protocol performs more complex security operations, and requires the transmission of larger messages.

Now, we evaluate the *processing time* on the server side. In particular, we refer to the following two metrics.

$\mathbf{T_T^{CPU}}$: total time spent to perform SMACK or DTLS operations on a received message $M1$, in the test cases *CoAP_SMACK* and *CoAP_DTLS*, respectively. This takes into account also initialization and management of data structures, as well as the DTLS handshake for the test case *CoAP_DTLS*.
$\mathbf{T_S^{CPU}}$: time specifically spent to check the validity of a received message $M1$. In the test case *CoAP_SMACK*, it considers the short MAC check performed by SMACK. Instead, in the test case *CoAP_DTLS*, it encompasses the decryption and authenticity check performed by DTLS.

| | First transaction | | Last transaction | |
|---|---|---|---|---|
| | **CoAP_SMACK** | **CoAP_DTLS** | **CoAP_SMACK** | **CoAP_DTLS** |
| $T_T^{CPU}$ | 3.992 ms ±0.010 ms | 220.703 ms ±5.153 ms | 0.517 ms ±0.012 ms | 0.705 ms ±0.008 ms |
| $T_S^{CPU}$ | 0.455 ms ±0.010 ms | 0.676 ms ±0.010 ms | 0.447 ms ±0.012 ms | 0.680 ms ±0.008 ms |

TABLE III. DEVICE COMPUTING OVERHEAD.

Table III shows the computing overhead on the server side. As expected, the computing times $T_T^{CPU}$ in the first transaction are always sensibly higher than the ones in the last transaction. Also, the total computing overhead $T_T^{CPU}$ due to DTLS in the first transaction is roughly 55 times higher than the same overhead due to SMACK. Once again, such a result is mainly due to performing the DTLS handshake. Furthermore, we can observe that, in steady state conditions, the overall computing overhead $T_T^{CPU}$ due to SMACK is the 26.66% less than the same one due to DTLS. Besides, the overhead $T_S^{CPU}$ due to the actual short MAC check is the 34.26% less than the same overhead due to DTLS security operations.

Now, we evaluate the *energy consumption* on the server side. We consider the total energy consumption as the sum of three contributions, i.e. $E = E^{CPU} + E^{TX} + E^{RX}$. That is, $E^{CPU}$ is the energy consumed by the CPU, while $E^{TX}$ and $E^{RX}$ are the energies consumed by the radio interface in transmission and reception mode, respectively. Each single contribution is computed as the product between the related time collected by the Energest framework, and the related power consumption reported in [22]. Finally, we refer to the following four metrics.

$\mathbf{E_T}$: total energy spent for a received message $M1$. It takes into account also initialization and management of data structures, as well as the DTLS handshake for the test case *CoAP_DTLS*.
$\mathbf{E_S}$: energy specifically spent to check the authenticity of a received message $M1$. In the test case *CoAP_SMACK*, it considers the short MAC check performed by SMACK. Instead, in the test case *CoAP_DTLS*, it encompasses the decryption and authenticity check performed by DTLS.
$\mathbf{E_T^{CPU}}$: CPU-related fraction of the $\mathbf{E_T}$ energy.
$\mathbf{E_S^{CPU}}$: CPU-related fraction of the $\mathbf{E_S}$ energy.

| | First transaction | | Last transaction | |
|---|---|---|---|---|
| | **CoAP_SMACK** | **CoAP_DTLS** | **CoAP_SMACK** | **CoAP_DTLS** |
| $E_T$ | 323.236 $\mu$J ±0.781 $\mu$J | 18,034.863 $\mu$J ±374.101 $\mu$J | 41.808 $\mu$J ±0.855 $\mu$J | 57.376 $\mu$J ±0.649 $\mu$J |
| $E_S$ | 36.557 $\mu$J ±0.774 $\mu$J | 55.028 $\mu$J ±0.535 $\mu$J | 36.305 $\mu$J ±0.950 $\mu$J | 54.849 $\mu$J ±0.430 $\mu$J |
| $E_T^{CPU}$ | 83.826 $\mu$J ±0.209 $\mu$J | 4,634.766 $\mu$J ±108.214 $\mu$J | 10.863 $\mu$J ±0.248 $\mu$J | 14.804 $\mu$J ±0.166 $\mu$J |
| $E_S^{CPU}$ | 9.549 $\mu$J ±0.215 $\mu$J | 14.195 $\mu$J ±0.201 $\mu$J | 9.389 $\mu$J ±0.244 $\mu$J | 14.291 $\mu$J ±0.171 $\mu$J |

TABLE IV. DEVICE ENERGY OVERHEAD.

Table IV reports the energy spent on the server side. As expected, the energy consumptions $E_T$ and $E_T^{CPU}$ in the first transaction are always sensibly higher than the ones in the last transaction. More important, both $E_T$ and $E_T^{CPU}$ due to DTLS in the first transaction are roughly 55 times higher than the same overheads due to SMACK. This suggests that SMACK is

particularly efficient against DoS attacks inducing the establishment of new communication sessions. Furthermore, we can observe that, in steady state conditions, the overall energy consumptions $E_T$ and $E_T^{CPU}$ due to SMACK are the $27.13\%$ and $26.62\%$ less than the same ones due to DTLS, respectively. Besides, the energy consumptions $E_S$ and $E_S^{CPU}$ due to the actual short MAC check are the $33.81\%$ and $34.3\%$ less than the same overheads due to DTLS security operations. It follows that the message validity check based on the short MAC proves to be considerably more efficient than standard authentication techniques, and thus is a good choice to efficiently detect DoS attacks and reduce their impact on constrained devices.

Note that, during out tests, we did not benefit of any *power saving* mode available on the CC2538 platform. As a result, once a CoAP message has been entirely received, the radio interface remains active in reception mode, as long as the message processing is completed. This explains why, both for the first and last transaction, we always have $E_T \gg E_T^{CPU}$ and $E_S \gg E_S^{CPU}$, even though SMACK does not require any additional communication on the server side. Of course, energy consumption can be further reduced by properly exploiting power saving modes available on the considered platform.

## XI. Conclusion

We have presented SMACK, a security service aimed at reducing the impact of Denial of Sleep attacks against resource constrained IoT devices. SMACK relies on a short Message Authentication Code (MAC) to early and efficiently detect messages sent by illegitimate sources. This avoids additional parsing and processing on invalid messages, reducing the impact of Denial of Sleep attacks. We have provided an adaptation of SMACK for the CoAP protocol, and experimentally evaluated its performance on our prototype implementation for the constrained CC2538 platform. Results show that SMACK is sustainable and efficient in terms of memory footprint, computing time, and energy consumption. Also, when compared to DTLS, it detects invalid messages while displaying a smaller computing overhead, without requiring any additional communication with IoT devices. Future works will evaluate SMACK together with additional reaction mechanisms in the presence of different DoS attacks, and consider secure schemes to renew long-term key material.

## Acknowledgments

This work was supported by the EU FP7 Collaborative Project SEGRID (Grant Agreement no. FP7-607109). We also want to thank Göran Selander for his valuable contribution to this work, and Ericsson for its financial contribution.

## References

[1] G. Kortuem, F. Kawsar, D. Fitton and V. Sundramoorthy, "Smart objects as building blocks for the Internet of things," *IEEE Internet Computing*, vol. 14, no. 1, pp. 44–51, January-February 2010.

[2] L. Atzori, A. Iera and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, October 2010.

[3] Z. Shelby, K. Hartke and C. Bormann, *RFC 7252, Constrained Application Protocol (CoAP)*, Internet Engineering Task Force, June 2014.

[4] D. R. Raymond, R. C. Marchany, M. I. Brownfield and S.F. Midkiff, "Effects of Denial-of-Sleep Attacks on Wireless Sensor Network MAC Protocols," *IEEE Transactions on Vehicular Technology*, vol. 58, no. 1, pp. 367–380, January 2009.

[5] F. Stajano and R. J. Anderson, "The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks," in *Proceedings of the 7th International Workshop on Security Protocols*. London, UK: Springer-Verlag, April 1999, pp. 172–194.

[6] T. Martin, M. Hsiao, D. Ha and J. Krishnaswami, "Denial-of-Service Attacks on Battery-powered Mobile Computers," in *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, ser. PERCOM '04. Washington, DC, USA: IEEE Computer Society, March 2004, pp. 309–318.

[7] E. Rescorla and N. Modadugu, *RFC 6347, Datagram Transport Layer Security Version 1.2*, Internet Engineering Task Force, January 2012.

[8] R. Daidone, G. Dini and M. Tiloca, "On experimentally evaluating the impact of security on IEEE 802.15.4 networks," in *3rd International Workshop on Performance Control in Wireless Sensor Networks*, ser. PWSN 2011, June 2011, pp. 20–25.

[9] Y. Xiao, H. H. Chen, B. Sun, R. Wang and S. Sethi, "MAC security and security overhead analysis in the IEEE 802.15.4 wireless sensor networks," *Journal on Wireless Communications and Networking*, vol. 2006, pp. 1–12, April 2006.

[10] Z. Wu, M. Xie and H. Wang, "Energy attack on server systems," in *Proceedings of the 5th USENIX conference on Offensive technologies*. Berkeley, CA, USA: USENIX Association, August 2011, pp. 62–70.

[11] T. K. Buennemeyer, M. Gora, R. C. Marchany and J.G. Tront, "Battery Exhaustion Attack Detection with Small Handheld Mobile Computers," in *Proceedings of the 2007 IEEE International Conference on Portable Information Devices*, ser. PORTABLE07. Washington, DC, USA: IEEE Computer Society, May 2007, pp. 1–5.

[12] T. Bhattasali and R. Chaki, "AMC Model for Denial of Sleep Attack Detection," *Journal of Recent Research Trends (JRRT)*, February 2012.

[13] D. R. Raymond and S. F. Midkiff, "Clustered Adaptive Rate Limiting: Defeating Denial-of-Sleep Attacks in Wireless Sensor Networks," in *IEEE 2007 Military Communications Conference*, ser. MILCOM 2007. Washington, DC, USA: IEEE Computer Society, October 2007, pp. 1–7.

[14] L. Seitz, G. Selander and C. Gehrmann, "Authorization framework for the Internet-of-Things," in *D-SPAN workshop of the IEEE 14th International Symposium and Workshops on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2013)*. Washington, DC, USA: IEEE Computer Society, June 2013, pp. 1–6.

[15] D. Stinson, *Cryptography: Theory and Practice, Second Edition*. CRC/C&H, 2002.

[16] C. Gehrmann, "Topics in Authentication Theory," Ph.D. dissertation, Lund University, 1997.

[17] R. J. McElice, *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Publishers, 1987.

[18] I. S. Reed and G. Solomon, "Polynomial Codes over certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, pp. 300–304, June 1960.

[19] A. J. Menezes, S. A. Vanstone and P. C. Van Oorschot, *Handbook of Applied Cryptography*, 1st ed. CRC Press, Inc., 1996.

[20] H. Krawczyk, M. Bellare and R. Canetti, *RFC 2104, HMAC: Keyed-Hashing for Message Authentication*, Internet Engineering Task Force, February 1997.

[21] "Californium (Cf) CoAP framework - Java CoAP Implementation." [Online]. Available: http://people.inf.ethz.ch/mkovatsc/californium.php

[22] Texas Instruments, *CC2538 Powerful System-On-Chip for 2.4-GHz IEEE 802.15.4, 6LoWPAN and ZigBee Applications*, Texas Instruments Inc., September 2014. [Online]. Available: http://www.ti.com/lit/gpn/cc2538

[23] "Erbium (Er) REST Engine - C CoAP Implementation." [Online]. Available: http://people.inf.ethz.ch/mkovatsc/erbium.php

[24] "tinydtls." [Online]. Available: http://tinydtls.sourceforge.net/

[25] A. Dunkels, F. Österlind, N. Tsiftes and Z. He, "Software-based Online Energy Estimation for Sensor Nodes," in *Proceedings of the 4th Workshop on Embedded Networked Sensors*, ser. EmNets '07. New York, NY, USA: ACM, 2007, pp. 28–32.

[26] D. McGrew and D. Bailey, *RFC 6655, AES-CCM Cipher Suites for Transport Layer Security (TLS)*, Internet Engineering Task Force, July 2012.