S3K: Scalable Security with Symmetric Keys - DTLS Key Establishment for the Internet of Things

Shahid Raza, Ludwig Seitz, Denis Sitenkov, and Göran Selander

Abstract—DTLS is becoming the de facto standard for communication security in the Internet of Things. In order to run the DTLS protocol one needs to establish keys between the communicating devices. The default method of key establishment requires X.509 certificates and a Public Key Infrastructure, an approach which is often too resource consuming for small IoT devices. DTLS also supports the use of pre-shared keys and raw public keys. These modes are more lightweight, but they are not scalable to a large number of devices.

We present *Scalable Security with Symmetric Keys (S3K)*, a key management architecture for the resource constrained Internet of Things. S3K provides a flexible and scalable way of establishing keys between resource constrained IoT devices. S3K enables devices that have no previous, direct security relation to use DTLS with either pre-shared symmetric keys or raw public keys established and authorized during the DTLS handshake. We implement S3K in the Contiki OS and evaluate it on real IoT hardware. Our evaluation shows that S3K is feasible in constrained environment and at the same time scalable to a large number of devices.

Note to Practitioners: Key management is one of the hardest problems in cyber security. It is even more challenging in the Internet of IoT considering that most things are resource-constrained. Therefore, IoT devices either end-up using the symmetric cryptography with pre-shared key mode or asymmetric cryptography with raw public keys (RPK) mode. These modes either require a pre-provisioning of all expected trusted clients in individual nodes before deployment or requires out-of-band validation of RPKs. Also, if the number of clients that a node would communicate with varies dynamically, this would demand frequent re-provisioning of each trusted client to the individual nodes. The approach based on pre-provisioning and re-provisioning of trusted keys is certainly not scalable and requires a continuous management of security policies. We therefore propose a solution that is scalable and does not require pre-provisioning or re-provisioning the individual nodes with keys for all future trusted clients. The basic approach is to establish shared keys between resource servers and a trust anchor. When a client wants to establish a trust relationship with a resource server it requests a key from a trust anchor. The trust anchor asserts a secret key or a public key of the client that can be conveyed to the resource server.

Index Terms—DTLS, Scalability, Keys, Security, CoAP, 6LoWPAN, Internet of Things

Shahid Raza, Ludwig Seitz and Denis Sitenkov are with the SICS Swedish ICT, Isafjordsgatan 22, 16854 Stockholm, Sweden. e-mail: {shahid, ludwig}@sics.se, and denis.sytenkov@gmail.com

Göran Selander is a Security Research at Ericsson, Färögatan 6, SE-16480 Stockholm, Sweden. e-mail: goran.selander@ericsson.com

Manuscript received June 01, 2015; final accepted paper submitted on December 21, 2015

I. INTRODUCTION

THE most frequently used security protocol on the Internet is TLS (often in conjunction with HTTPS) [1][2]. In the Internet of Things (IoT), TLS can often not be used, since it requires TCP as underlying transport protocol, and many IoT applications use UDP instead. The IETF has developed Datagram TLS (DTLS) [3], a UDP based variant of TLS, which is therefore commonly applied in IoT settings.

Both TLS and DTLS define a handshake protocol in order to initiate a secure session between a client and a server. The protocol requires X.509 certificates and a corresponding verification infrastructure. Since such an infrastructure is often not viable for large deployments of small IoT devices, such as sensor networks, additional protocols have been developed, to support the use of either raw public keys [4] or pre-shared symmetric keys [5].

When using one of these new modes, an important issue is key establishment, especially when the client has no direct prior relation to the server. Since a client might only have an ephemeral relation to a specific server, disclosing the server's symmetric key to the client in the pre-shared key mode is not a good solution. Large scale distribution of raw public keys is also not viable.

Protocols like Kerberos [6] have been used to solve exactly this problem for a long time, by using a Key Distribution Center (KDC) - a Trusted Third Party (TTP) - that generates and distributes ephemeral keys to clients, without disclosing the secret shared key of the server. For many Internet applications there is also a TTP involved. For example, in order to handle some degree of flexibility in authorizing access to resources, an Authorization Server is assumed that provides trusted assertions to servers about requesting clients [7].

A similar setting is anticipated and even more needed for IoT applications, since constrained devices may typically need support with authorization management as well as session key establishment. Examples of IoT applications which benefit from trusted third party support include Smart Metering, Building Automation, Personal Health Monitoring and Industrial Control Systems (see ACE in Section III).

However in addition to message sizes, the number of messages exchanged has a negative impact on the performance of constrained devices, so in order to reduce the overhead this should be kept at a minimum. Introducing an additional key exchange mechanism such as Kerberos, works against this objective.

This article proposes *Salable Security with Symmetric Keys (S3K)* a framework that comprises two schemes to establish

a trust relation between communication partners that have no previous, direct relationship.

The first approach is similar to Kerberos, but it integrates seamlessly with both TLS and DTLS without requiring any changes to the original protocols, in particular no extra round trips are needed. In a dynamically changing setup of clients and servers, leveraging an existing TTP, this approach allows to establish a new pre-shared symmetric key between client and server within the DTLS/TLS handshake without disclosing the secret key of the server.

The second approach applies to raw public keys and also integrates seamlessly with both DTLS and TLS: The TTP issues a symmetric key based assertion of the client public key, which the client can use as certificate in the handshake. This also has the advantage of being much more lightweight than a X.509 certificate and not requiring any PKI associated processing.

Furthermore our framework can be used to expire keys on devices that cannot reliably measure time, revoke existing keys and prevent replay of previous key establishment messages.

The rest of the article is structured as follows: section II presents the technological background of our framework, including protocols and standards on which this framework is based. In section III we present related work in the field of key establishment for Internet of Things. Section IV presents the architecture that is the basis of our framework; while section V gives details on the implementation we did for testing purposes. We present the results of measurements and other evaluation performed on our implementation in section VI and finally draw conclusion on our work in section VII.

II. BACKGROUND

Our baseline assumptions are constrained devices with about 10 KB of RAM and about 100 KB of persistent memory (e.g. flash). This corresponds to class 1 devices according to the terminology for constrained-node networks [8]. We expect a significant number of these devices to be battery powered, therefore they need to minimize network traffic, as this is the largest power consumption factor compared to e.g. cryptography [9][10].

Furthermore we assume the following communication protocol stack, although our approach can be applied to other configurations as well:

- The application layer protocol is CoAP [11], using UDP at the transport layer.
- CoAP runs on top of DTLS [3].
- IPv6 [12] is assumed to be the Internet layer protocol.
- 6LoWPAN [13] is used as adaptation layer protocol between the Internet layer and the MAC and physical layer.
- On the MAC and physical layer we assume that IEEE 802.15.4 [14] is used.

Currently security for the most constrained devices in the Internet of Things is addressed mostly in terms of communication security (confidentiality, integrity, possibly non-repudiation) and key management. A variety of competing or complementary protocols are in use, each of which has its own security solution.

Many wireless sensor networks use the IEEE 802.15.4 protocol [14] at the physical layer. This protocol includes some security services at the MAC sublayer, namely data confidentiality, data authenticity and replay protection. It also provides some features for key establishment needed for these services, however it does not define how keys are to be managed (cf. chapter 7 of [14]).

The ZigBee protocol [15] is based on IEEE 802.15 and thus its security makes use of the IEEE 802.15 security services. The ZigBee specification defines services for key establishment, key transport, frame protection, and device management, using a Trusted Third Party (TTP), called the Trust Center, for key management (cf. chapter 4 of [15]).

The Bluetooth specification [16] includes a profile for Bluetooth Low Energy, targeted at novel applications in the healthcare, fitness, security, and home entertainment industries. The security features of Bluetooth Low Energy are basically a subset of the overall Bluetooth security features, including link layer encryption and authentication using the CCM algorithm. Furthermore the specification defines a Security Manager component of the Bluetooth architecture, in charge of device pairing and key management (cf. Volume 2 Part H, Volume 3 Part H and Volume 6 Part E of [16]).

The Internet Engineering Task Force (IETF) maintains a number of standards to enable constrained devices to use internet protocols. The 6LoWPAN [13] protocol is an adaptation layer on top of IEEE 802.15.4 that allows the use of IPv6 in low power lossy networks.

The CoAP protocol [11] is a specialized web transfer protocol at the application layer, for use with constrained nodes and constrained (e.g., low-power, lossy) networks. CoAP security is mainly defined through a binding to DTLS [17]. CoAP enabled devices are expected to maintain some kind of Access Control List (ACL) that specifies which other devices are authorized to initiate a DTLS connection with them. How these ACLs look like and how they are expected to be managed is not specified in CoAP. CoAP defines four security modes, for use with DTLS:

- NoSec mode has no security and DTLS is disabled.
- PreSharedKey (PSK) mode uses pre-shared secret keys.
- RawPublicKey (RPK) mode uses asymmetric keys without certificates.
- Certificate mode uses regular X.509 certificates.

CoAP does not specify how these keys are established except for the RPK mode. For RPK mode, the provisioning of the necessary list of other public key identifiers that are allowed to initiate DTLS sessions with a device is static, i.e. it is assumed that this is done at commissioning time.

III. RELATED WORK

Different efforts are underway to address the security and privacy challenges in billions of connected IoT devices. Previously, solutions have been developed to protect communication between resource-constrained IoT by using lightweight IPsec [18], DTLS [17] [19], and link-layer security [18]. In addition to the communication security, network security in the IoT is also provided using an intrusion detection system [20],

and an efficient solution to protect stored data inside a resource constrained node is also proposed [21]. Different standardization efforts are also underway to provide security in the IoT.

The *DTLS In Constrained Environments (DICE)* working group¹ at IETF has been created with the goal of defining profiles that adapt DTLS to constrained environments. Furthermore the *Authentication and Authorization for Constrained Environments (ACE)* working group² also at IETF has been chartered in June 2014 with the goal of defining more concrete, standardized ways of authenticating and authorizing access involving constrained nodes and constrained networks.

The Open Mobile Alliance is working on a standard for machine to machine (M2M) device management called OMA Lightweight M2M [22]. This draft standard is based on CoAP and DTLS, but it defines an additional communication security binding for CoAP over SMS. Furthermore it defines an Access Control List (ACL) structure, the management of these ACLs and processing rules for reaching access control decisions from these ACLs. The draft standard defines encodings for establishing DTLS keys for the CoAP security modes, and different methods for bootstrapping; it also specifies that bootstrapping must use a secure session for sensitive security-related data. The draft is silent on how this secure channel can be set up in the absence of any pre-established credentials.

Kerberos [6] is an authentication protocol developed in 80-ies. It allows two parties previously unknown to each other to establish a shared secret key through the use of a trusted third party, with whom both parties share a secret key already. Our approach fundamentally tries to solve the same problem, using a very similar architecture. However our approach is integrated with DTLS, while using Kerberos would require the constrained device to implement and perform the Kerberos protocol in addition to session security such as DTLS.

Hernandez-Ramos et al. have designed an authentication and authorization framework for smart objects [23]. Their authentication approach is based on EAP over LAN (EAPOL) [24] for security bootstrapping in order to establish keys for DTLS. This burdens the constrained device with the necessity to implement and execute the EAPOL protocol in addition to DTLS, an approach which seems to be ill-adapted to constrained devices.

Pereira et al. introduce a framework for authentication and access control for CoAP-based Internet of Things [25]. Their framework leverages Kerberos [6] and RADIUS [26] to provide authentication and access control. It proposes to use either IPSec or DTLS for session security with the credentials established with Kerberos and RADIUS. As with the previous approaches this requires constrained devices to implement and execute additional protocols in conjunction with DTLS, thus we consider this to be too heavyweight for constrained devices.

IV. S3K FOR DATAGRAM TLS

Key management is one of the hardest problems in cyber security. It is even more challenging in the internet-

connected IoT considering that most *things* are resource-constrained (limited storage, processing, and bandwidth). On the Internet today client devices authenticate servers (mostly through a web browser) using a digital certificate, whereas servers authenticate clients using a username or a password. The lack of traditional user interfaces (such as keyboard and display screen) on constrained things hinders the use of a username and password for client authentication; this approach also has inherent weaknesses that limit the use of dynamic passwords which may render the use of constant passwords, which introduces weaknesses.

Asymmetric cryptography with Public Key Infrastructure (PKI) is not always suitable for constrained environments either, until we have smaller standardized certificates for resource-constrained devices and economical methods of developing PKI with globally trusted certificates. Therefore, constrained devices either end-up using the symmetric cryptography with Pre-shared key (PSK) mode or asymmetric cryptography with raw public keys (RPK) mode. These modes either require a pre-shared key in a constrained node for each remote client that wants to establish a secure connection with the node, or requires out-of-band validation of RPKs. If a node is expected to communicate with a large number of clients this would require a pre-provisioning of all expected trusted clients in individual nodes before deployment. Also, if the number of clients that a node would communicate with varies dynamically, for example pay-per-use setting, this would demand frequent re-provisioning of each trusted client to the individual nodes.

The approach based on pre-provisioning and re-provisioning of trusted keys is certainly not scalable and requires a continuous management of security policies. We therefore propose a solution that is scalable and does not require pre-provisioning or re-provisioning the individual nodes with keys for all future trusted clients that intend to establish secure connections with these nodes. Our solution requires only a single trusted relationship between a constrained device and trust anchor.

Before presenting our solutions we introduce the terminologies we will be using in this paper: *resource server* is a resource-constrained device that offers one or more resources such as temperature readings; *client* is a device that requests a resource from a resource server; and *trust anchor* is a trusted third-party that both resource server and client use to establish trust relationship between them.

The basic approach is to establish shared keys between resource servers and a trust anchor. When a client wants to establish a trust relationship with a resource server it requests a key from a trust anchor. The trust anchor asserts a secret key or a public key of the client that can be conveyed to the resource server. The assertion is performed such that the public key is integrity protected and the secret key is confidentiality and integrity protected. In the following sections we will further elaborate and apply this technique to establish a secure connection using the DTLS protocol with PSK and RPK based on the asserted secret key or public key, respectively. Recall that the DTLS is the standard security protocol for CoAP [11], a new web standard for the IoT.

The normal mode of operations for DTLS is to use X.509

¹<https://datatracker.ietf.org/wg/dice/>

²<https://datatracker.ietf.org/wg/ace/>

certificates, however other more lightweight solutions have been developed, relying on pre-shared secret keys (PSK) [5] or on raw public keys (RPK) [4] that are validated out of band in an unspecified manner.

When using *certificates*, one can establish DTLS connections in a flexible scalable way provided that a PKI is present and certificates are deployed to all participating entities. This is clearly not a reasonable expectation in large scale deployment of constrained IoT devices.

In *PSK mode* the communicating parties have to establish a shared secret key before the DTLS session is initiated. How that is done is not specified in the standard describing PSK [5]. Basic PSK mode does not provide perfect forward secrecy (PFS), meaning that if the pre-shared key is somehow compromised, an attacker can decrypt the messages of previous sessions. There are PSK modes that provide PFS, using Diffie-Hellman exchanges, but these are not the mandatory to implement modes of CoAP. Furthermore anyone holding a pre-shared key can impersonate the other communication partner, especially if the same key is shared within a group.

In *RPK mode* raw public keys are used instead of X.509 certificates, thus transmission overhead is reduced and a PKI is no longer necessary. However these keys still need to be bound to a specific entity in order for the DTLS protocol to be secure. Therefore some out-of-band method is assumed by which the entities participating in a communication learn of each other's public keys.

The CoAP standard describes some measures that can be used for provisioning RPKs or lists of RPKs, as e.g. a method to derive a short identifier from the public key.

A. Datagram TLS with DerivedKey Mode

When performing the DTLS handshake in PSK mode, the pre-shared keys of the communicating entities need to be provisioned. One possible solution is to hard-code these keys and identities for all possible future clients in a resource server before deployment. This does not scale well in dynamic settings.

A common approach, that is adopted by our solution, is to use a trust anchor to establish keys and identities between different resource servers and previously unknown clients. This approach is more scalable and easier to manage.

In this approach a resource server and a trust anchor share a secret key, which is stored in the resource server before deployment. We denote this key by K_t . A trust anchor shares a unique K_t with each resource server. Also, a resource server can contain multiple unique K_t shared with different trust anchors; for example, a resource server can offer a resource A via an access through trust anchor T_A and for a resource B via an access through trust anchor T_B . For simplicity, we consider the case where a resource server shares a key with only one trust server; this is also the typical case.

A client can request a PSK, denoted by K_c in this paper, from the trust anchor, that in turn generates a K_c and a $psk_identity$ and sends them to the client via a secure channel; section IV-A4 discusses the creation of a secure channel between a trust anchor and a client. The client uses the K_c

Fig. 1: Sequence of required steps when a client C wants to establish a secure DTLS PSK-based connection with a resource server using our DerivedKey mode.

and $psk_identity$ during the DTLS handshake to establish a secure connection with a resource server. Figure 1 illustrates this process.

Upon receiving a DTLS connection request from a client with $psk_identity$ as identify hint in the *ClientKeyExchange* message, resource server parses the $psk_identity$ field and proceeds with the derivation of K_c using the K_t and $psk_identity$. In the following section, we explain the derivation of $psk_identity$ and K_c .

1) *psk_identity Derivation*: In order to provide additional key management services, discussed in Section IV-A3, a nonce is used as the $psk_identity$ field of the DTLS *ClientKeyExchange* message. A cryptographic nonce is a number that is used only once in a cryptographic protocol. A trust anchor generates the nonce as follows.

```
psk_identity = nonce = BASE64enc(
  Kc_prefix + TA_id + C_id + RS_id + Kc_size + seq_number
)
```

where K_{c_prefix} is a 3-bytes constant value set to $\{0x0C, 0x44, 0x4A\}$, which specifies the use of PSK with our DerivedKey mode; TA_id is a 1-byte value that represents the identity of the trust anchor, which a resource server can use to select the right K_t ; C_id is a 12-byte identity of a client, which ensures that K_c is generated for a specific client; RS_id is a 12-byte identity of a resource server, which ensures that client can only use K_c to access a specific resource server; K_{c_size} is 1-byte value to represent the size of K_c in bytes, which allows the flexibility to use different key sizes; and seq_number is a 8-byte sequence number maintained by the trust anchor and resource server, which is discussed in section IV-A3.

The sequence number ensures that each nonce is unique.

In order to be able to use these byte values in the $psk_identity$ that requires ASCII characters, we Base64 end-code the resulting byte-string.

2) *PSK Derivation*: The pre-shared key (i.e. K_c) derivation approach is based on the data expansion method proposed in the Transport Layer Security (TLS) Protocol Version 1.2 [1]. The data expansion function, denoted by $P_hash(secret, data)$, uses a single hash function to expand a secret and seed into an arbitrary quantity of output. This function is defined as

```
P_hash (secret, seed) =
    HMAC_hash(secret, A(1) + seed) +
    HMAC_hash(secret, A(2) + seed) +
    HMAC_hash(secret, A(3) + seed) + ...
```

where $+$ indicates concatenation and $A(i)$ is defined as:

```
A(0) = seed
A(i) = HMAC_hash(secret, A(i-1))
```

We can iterate P_hash as many times as necessary to produce the required quantity of data. For example, if P_SHA256 is being used a single iteration will produce 32 bytes or 256 bits. Using the SHA256 hash function if we want to generate a 256-bit K_c , the data expansion method will be the following. In our case, seed is $psk_identity$ and secret is K_t .

```
K_c = P_SHA256(K_t, psk_identity) =
    HMAC_SHA256(K_t, A(1) + psk_identity)
```

where

```
A(1) = HMAC_SHA256(K_t + A(0)) =
    HMAC_SHA256(K_t + psk_identity)
```

In case we want to use 128-bit K_c , we can keep the first 128 bits and discard the rest.

3) Additional Key Management Services:

a) *Unreliable Clocks*: Constrained devices often have no reliable means of measuring time, especially if they sleep for extended periods, in order to conserve battery power. Therefore date and time based mechanisms to ensure freshness and expiration of keys work badly in such environments. Note that the CC2538 platform we used for our implementation includes a real time clock and is therefore able to reliably measure time. However this cannot be expected of the smallest devices which we target with this framework.

b) *Key Freshness*: The freshness security service defends against replay attacks using the sequence number to mark already used keys K_c . The sequence number counter is set to zero when an association between a trust anchor and a resource server is made, i.e., when a new K_t is provisioned to a resource sever and a trust anchor. When a sequence number range gets exhausted a new K_t must be provisioned before setting the sequence number counter back to zero.

A trust anchor increments a sequence number by 1 each time it generates a new K_t . The resource server increases the sequence number each time it gets a valid DTLS connection request and the connection is successful. For each trust anchor (as a resource server can have multiple trust anchors), a resource server maintains a window of most recently verified sequence numbers using the sliding window procedure described in RFC 2402 [27]. A request for a new DTLS connection by a client is considered as fresh if the sequence number is new and falls within the sliding window, or to the right of window. For each received request for a DTLS connection, the resource server must first check the

nonce's sequence number part before moving on to the further verification of nonce and generation of K_c . The sliding window ensures that keys can be used in different order than they were issued. This is important, since the delays in especially wireless IoT networks can cause requests to arrive in different order than the one they were sent in.

c) *Key Expiration*: Security keys often have a lifetime and hence an expiry date. The sequence number part of the nonce can be used to expire the K_c - nonce pair. The size of the sliding window described in the previous paragraph provides a weaker form of expiration in the absence of reliable time measurements. When the sequence number associated to a key K_c drops out of the sliding window, it is considered to be expired.

d) *Key Revocation*: It is sometime desirable to revoke keys before they expire. For example if a client is compromised or a service/resource agreement between a resource server and a client is void. In such cases a trust anchor can send a key revocation message to a resource server with the sequence number of the key (K_c) it wants to revoke. The resource server marks the key as used in the sliding window thereby making it unusable.

4) *Secure Connection between a Trust Anchor and a Client*: The key exchange between a trust anchor and client must be carried out on a secure channel. We assume that both trust anchors and clients are not so constrained devices and can establish a secure connection using the conventional communication security protocols such as TLS/DTLS and IKE/IPsec. How this is done is outside the scope of this paper and is well known. Upon request for a resource, a trust anchor can verify if a client is allowed to access a resource from a resource server. A trust anchor or a third party can for example run a commercial service to control protected access to resource servers. How this is done is also outside the scope of this paper.

B. Datagram TLS with AuthorizedPublicKey Mode

The pre-shared key K_t can also be used to perform the DTLS handshake in the Raw Public Key (RPK) mode. Our AuthorizedPublicKey (APK) mode is analogous to the RPK but without the need for out-of-band validation of public keys. Figure 2 shows the APK exchange between a client and a resource server plus the full handshake using this mode. For a DTLS handshake with a resource server, a client requests an APK from a trust anchor that in turn creates an authorization certificate, protects it with a Message Authentication Code (MAC) using the key K_t , and sends it to the client along with the public key of the resource server to which this APK applies. During the DTLS handshake in the RPK mode, a client replaces the RPK with the APK (i.e. the authorization certificate). The resource server verifies the APK and proceeds with the DTLS handshake in the RPK mode. For this new mode we require similar changes as were required by the RPK mode [4], i.e., we need a new *certificate_type* and a new structure; we specify these extensions in Section IV-B1. Note that the resource server sends an empty *certificate_list* in the *Certificate* message, since the public key of the resource

Fig. 2: A full DTLS handshake with our AuthorizedPublicKey mode.

server is already provided to the client during the initial exchange; step 2 in Figure 2. The use of X.509 is just an indication that allows us to send an empty *certificate_list* in the *Certificate* message. In response to the resource server's *CertificateRequest* message, the client sends *MacCert* (defined in Section IV-B1) in the *Certificate* message.

The use of smaller sized APK makes the client's *Certificate* message very lightweight compared to X.509 certificates. The use of a MAC further reduces the overhead compared to using asymmetric signatures. Our APK is in fact the RPK of the RFC 7250 [4] with an additional MAC.

1) *Structure of the APK Extension*: We call the new *certificate_type* *AuthorizedPublicKey* and the new structure is named *MacCert*.

```
struct {
  select(certificate_type) {
    //Our proposed certificate type
    case AuthorizedPublicKey:
      MacCert certificate;

    // RawPublicKey certificate defined in [RFC7250]
    case RawPublicKey:
      opaque ASN.1_subjectPublicKeyInfo<1..2^24-1>;

    // X.509 certificate defined in [RFC 5246]
    case X.509:
      ASN.1Cert certificate_list<0..2^24-1>;

    // Additional certificate type based on TLS
    // Certificate Type Registry
  };
} Certificate;
```

The *MacCert* structure is defined as follows:

```
struct {
  opaque ASN.1_subjectPublicKeyInfo<1..2^24-1>
```

Fig. 3: Our IoT setup that is used to evaluate DTLS with scalable symmetric keys.

```
opaque trust_anchor_id;
uint32 sequence_number;
MACAlgorithm mac_algorithm;
uint8 mac_length;
opaque MAC[mac_length];
} MacCert;
```

The *ASN.1_subjectPublicKeyInfo* is defined in the RFC 7250[4]. *MACAlgorithm* is specified in RFC 5246 [1] and the *mac_algorithm* parameter is a function defined as follows:

$MAC = M(key, message)$

where *key* is K_t and *message* is the certificate with an empty MAC. We also need the extended *client_hello* and *server_hello* messages defined in the RFC 7250 with *AuthorizedPublicKey* as *certificate_type*.

A resource server can verify the public key of client that is sent in the *ClientCertificate* message by (i) searching the *trust_anchor_id* in its list of trust anchors, (ii) checking the validity of *sequence_number* and *ASN.1_subjectPublicKeyInfo*, and (iii) verifying the MAC using the key K_t . If any of them fails the resource server must abort the handshake.

V. IMPLEMENTATION

In order to verify and evaluate S3K DerivedKey mode we implemented the three agents (client, TA, and RS) in an IoT setup shown in Figure 3.

- (i) The client is implemented in Java as a desktop application.
- (ii) The TA is implemented in Java as a web service application.
- (iii) The RS is implemented in C on a constrained device.

The TA web service is continuously running to accept the DTLS key request as HTTP requests over a secure TLS channel. The client is also connected to a Border Router (BR) over the serial port utilizing Serial Line Internet Protocol (SLIP). The RS is a part of a 6LoWPAN network and connected to the BR over the IEEE 802.15.4 low power radio.

A. Resource Server Implementation

Major part of the development effort was spent on the RS implementation. The RS is implemented on a constrained device, but it should still provide enough memory and computational power to run the operating system, IP stack, DTLS, CoAP protocol as well as our proposed S3K scheme, and a test application. Another important requirement for the device is presence of a 802.15.4 radio unit.

We therefore choose the CC2538 platform, which is based on a 32 bit ARM-Cortex M3 core, with a built-in 802.15.4 radio unit. Furthermore the CC2538 has some cryptographic accelerators that support SHA2 and AES-128/256 and a hardware

pseudo random number generator that can be initialized with random numbers generated by radio module.

We use the CC2538DK [28] development kit. The development board has CC2538F512RKU processor, 512kB of internal flash or ROM, 32kB of RAM, clock frequency of 32MHz, 2.1V voltage, processor core current of 13mA, and oscillator frequency of 32.768kHz. Moreover, the CC2538DK development board has 4 built in LEDs, accelerometer and a light sensor [29] that helps to build a test application for protocol evaluation.

1) *Operating System*: In this paper we use Contiki, an open source operating system designed for the IoT [30]. One of the important benefits of this operating system is support for the IP protocol stack. This includes IPv6 (with 6LoWPAN), TCP, and UDP protocols. In addition to this, the Contiki OS comes with built in CoAP implementation Erbium [31]. It supports both server and client modes and has a small code overhead. We use Erbium for the evaluation.

2) *DTLS library*: We use the tinyDTLS [32] library as a basic implementation of DTLS. TinyDTLS is an open source library implemented in C and optimized for constrained devices running the Contiki OS. It supports the PSK mode with the TLS_PSK_AES_128_CCM_8 cipher suite that is mandatory to implement in CoAP. Other DTLS alternatives include GnuTLS [33], openssl [34] and MatrixSSL [35]. We extend the tinyDTLS library to support S3K and implement the following additional features.

- The software base pseudo random number generation (PRNG) is replaced with hardware accelerator for PRNG generation.
- Functions that utilize the SHA2 hardware accelerator for HMAC calculation were implemented to replace the software based SHA256 calculation.
- DTLS session handler was extended to keep a pointer for custom information and search of existing session by custom information.
- The new event DTLS_EVENT_CLOSED was declared. This event is send when the DTLS session is about to be closed but session information was not removed yet. It can be handled by the event function together with DTLS_EVENT_CONNECTED.

These modifications are not application specific and treated as the part of the DTLS library.

3) *DTLS and CoAP integration*: We integrate the tinyDTLS library and the Erbium CoAP implementation to support secure CoAP (CoAPs). This integration supports enabling and disabling of security based on the IS_SECURE flag that we add. We also implement callback methods to process secure CoAP messages.

The *coap_receiver* process receives the new packet and checks by destination port number if it has to be processed as secure or insecure. If the data was sent to insecure CoAP connection the request is processed as any other CoAP request. Figure 4 shows the flow of secure and insecure messages in the integrated CoAP-DTLS.

4) *S3K Key Derivation*: The key derivation logic in the DerivedKey mode is implemented as a function that accepts the PSK identity and length of the PSK identity in bytes and

Fig. 4: Message processing flowchart of a secure and insecure CoAP-based communication.

Fig. 5: S3K key derivation process.

returns the status of the operation (0 if key is generated and non-0 if an error occurs), the key, the length of the key in bytes, the TA identifier and the sequence number from the nonce. The key derivation flowchart is shown in Figure 5.

In order to prevent redundant base64 decoding, the function verifies that the length of the PSK identity field is at least 37 (the size of a nonce) but does not exceed 60 bytes (the maximum length of nonce Base64 encoding). If the length of PSK identity is within the acceptable limit the nonce is decoded from Base64. If the result of the decoding is exactly 37 bytes long, the data of the nonce is verified in the following order: first 3 bytes of the nonce identify the derived key mode and are equal to 0x0C, 0x44, 0x4A; the RS identifier specified in the nonce is equal to the RS identifier of the device; the TA identity is known to the device; the sequence number of the nonce satisfies requirements of the sliding window of the specified TA. If all checks of the nonce are successful then the derived key is generated and truncated if it is required by the key size parameter of the nonce. As the implemented key management protocol is designed to be used together with the DTLS protocol the key generation function utilizes the HMAC

calculation module of the tinyDTLS library to reduce the code footprint.

5) *Key revocation*: The key revocation is implemented as a CoAP resource that is not protected with transport security. The resource utilizes the DELETE method of the CoAP protocol and the request is sent as a raw sequence of bytes in the payload. The request is not encrypted but protected with a MAC. As HMAC calculation may cost significant computational resources, following two conditions are verified when the new key revocation request is received: (i) the RS identity specified in the request has to be equal the resource identity of the device, and (ii) either the DTLS session for the sequence number specified in the key revocation request exists or key is marked as unused in sliding window. If verification above is successful the MAC is calculated with key K_t and compared with the received MAC. In case the message sender identity is not confirmed the request is ignored. Otherwise the sliding window is updated and the DTLS session is terminated if it exists.

6) *Test Application in RS*: The test application is implemented as a CoAP resource that is protected with the DTLS protocol and performs a simple function of switching on and off LEDs on the evaluation board. In addition to this it defines callback functions for tinyDTLS context and handles tinyDTLS events. Among other things it updates the sliding window information when the DTLS handshake is finished and the session is marked as connected. The modified tinyDTLS library supports both pure PSK mode and derived key mode. The test application also contains service code to configure the DTLS context and start CoAP server. In the RS we also implement the sliding window mechanism for sequence number freshness.

B. Border router

The task of the border router is to connect our client, which is using wired Internet, to the RS which uses the wireless IEEE 802.15.4 radio. The border router is a CC2538DK board that is programmed with a RPL border router application. That application can be found in Contiki OS examples. The router is connected to PC with the Serial Line Internet protocol (SLIP). In order to run the SLIP protocol the tunslip6 tool from Contiki 2.7 distribution was used. This tool creates the additional network interface with the specified prefix and allows IP packet exchange between the PC and the RPL border router.

C. Trust anchor implementation

The Trust Anchor is implemented in Java as a web application that provides a REST service with 2 endpoints.

The key generation endpoint (token/generate) creates the new nonce and derived key pairs. As input it accepts the JSON object that contains the alias of the client that sends the request and the alias of the RA that needs to be accessed. The TA finds existing RS and client identifiers by aliases in the database, generates the nonce and the key and increments the sequence number counter. The response is a string representation of the

	code	constant data	variables	total
S3K	1636 B	48 B	24 B	1708 B
key management (part of S3K)				392 B
tinyDTLS	19368 B	140 B	2084 B	21592 B

TABLE I: Code size and memory overhead of S3K Derived-Key mode

JSON object that contains nonce encoded as Base64 and the session key K_c .

The key revocation endpoint (token/revoke) is designed to send key revocation requests. As input it accepts the JSON object that contains the alias of the RA where the key has to be revoked and the sequence number of that key. The TA finds the RS identifier by the alias and sends key revocation request to the selected RS.

The REST service is secured with the TLS protocol based on self-signed X.509 certificates. Since our main goal is to evaluate the part of the protocol that have to run on the constrained devices we did not implement any authentication.

D. Client implementation

The client application is implemented in Java. It is designed to execute three main functions: obtain the nonce and the derived key K_c from the TA, perform the DTLS handshake with the RS using granted nonce, and send the CoAP request over the DTLS record protocol to RS.

The nonce and the derived key are obtained in a JSON object from the key generation endpoint. In order to implement the DTLS handshake and the secure CoAP communication we used the Scandium open source DTLS implementation and the Californium CoAP implementation. These frameworks were selected because they provide all necessary functionality and are easy to integrate.

VI. EVALUATION

In order to show the feasibility of S3K in constrained environments, we evaluate it in an IoT setup shown in Figure 3.

A. Memory Overhead

In order to evaluate code size and memory usage the *arm-none-eabi-size* utility was used. For more detailed information about RAM and ROM usage we used the tool *arm-none-eabi-objdump*. Both tools are included in the GNU toolchain for ARM processor utility. To perform a measurement for a certain functional part of the code that is in charge of specific functionality, it is removed from compilation with the `#define C` directive and the value is calculated as delta of the application size with and without the functional part.

The total size of S3K is 1708 bytes including 1636 bytes of code, 48 bytes of static information and 24 bytes allocated for global variables. This number also includes the 16 bytes long RS identifier and 78 bytes long key K_t . The key management functionality in S3K accounts for a total of 392 bytes.

The total size of the tinyDTLS library (including the necessary modifications) is 21592 bytes. This code footprint

Fig. 6: Energy spent on the S3K key derivation and key revocations.

includes 19368 bytes of program, 140 bytes of constant data and 2084 bytes allocated for global variables.

Table I summarizes these findings.

B. Time and Energy Overhead

The computational time is measured with the Contiki *Energist* [36] module. This module is based on the real time clock and measures usage time of different platform units separately (CPU time, CPU time in low power mode, radio listening time and radio transmission time). The real time clock on CC2538EM platform works on 32.786kHz frequency that allows to perform measurements with resolution 0.03 ms. The Energist module accumulates the number of real time clock ticks in 64 bit unsigned values since its activation. That capacity is more than enough to measure the computational time of the most of S3K operations.

Energy consumption is calculated from time that is measured with the Energist module according to the following equation.

$$E = U * I * t$$

In this equation U is a power supply voltage taken from platform documentation, I is an average current for the respective module from processor specification and t is the time value measured with the Energist. The energy value is calculated for each model and then summed up to take into account not only energy used by processor for computation, but also potentially lost radio duty cycles.

The energy consumed by S3K is measured separately for the key derivation function and for the key revocation function. Figure 6 shows the energy consumes on key handling part of S3K. The key derivation using the software SHA256 computation takes 75.47 ticks or 2.30 milliseconds on average while the same computation with SHA2 hardware accelerator takes 14.6 ticks or 0.45 milliseconds. The key revocation with and without SHA2 accelerator takes 73.7 ticks (2.25 ms) and 11.4 ticks (0.35 ms) respectively. The energy in nJ is calculated from the time using the above formula.

We also measure the energy overhead of individual DTLS handshake messages. For that the processing time is measured starting from receiving the message and until the reply is sent. Moreover the total handshake time starting from the first *ClientHello* message and ending with processing of the

Fig. 7: Energy spent on different DTLS handshake messages in S3K DerivedKey mode.

last *Finished* message was measured to get an idea about the maximum number of handshakes that can be processed by the device within a unit of time. Figure 7 shows the energy spend on individual handshake message plus the total energy spend on the full handshake process.

The minimum time for the complete handshake for SHA256 software computation is 533.05 ms and average time is 775.05 ms. In case the SHA2 hardware accelerator is used the minimum and average time is 511.65 ms and 711.11 ms respectively.

The average RTT value was 362.95 ms. Hence, the average OWD value is 181.48 ms and we can assume that attacker can send 8.44 requests per second.

In order to evaluate how many requests can be send by an attacker per second we measured the Round Trip Time (RTT) between client and server with a ping request. As a ping request processing time is negligible and the network consists of two nodes so the route is deterministic we assume the One Way Delay (OWD) is a half of RTT. Hence, the number of requests that can be sent by an attacker during the period of time is the length of the period divided by OWD.

The measurements were performed with ContikiMAC radio duty cycle protocol [37]. The Java client running on the PC was sending either a DTLS handshake requests sequence or a key revocation request the RS running on the embedded platform, depending on the evaluated function. Data was collected 30 times for each function.

The collected data was analyzed to make a conclusion about performance of the S3K framework and its resistance for Denial of Service (DoS) and battery-drain attacks that are specific for constrained devices.

C. Evaluation Discussion

Analysis of the key management framework code footprint shows that key derivation and revocation functions take 7.9% of the total DTLS implementation. The key derivation function takes 6.0% of total computational time per handshake. SHA2 hardware acceleration has a significant impact on the key derivation and revocation processing time for DTLS handshake messages. The accelerator speeds up the key derivation computation 5.11 times and the key revocation computation 6.43 times. Also the accelerator speeds up overall computation time of the handshake 4.37 times.

We found out, by comparing overall handshake time with and without SHA2 hardware acceleration, that optimization of the computational time has almost no impact on total handshake time. Hence, the key derivation has no significant impact on total DTLS handshake time.

1) *Attack Analysis*: We analyze S3K for the battery-drain attack and DoS attacks. The battery-drain attack can be performed either by sending the sequence of key revocation requests with well-structured valid data but wrong MAC or by series of attempts to perform a DTLS handshake with well-structured and acceptable nonce. In case an attacker tries to send key revocation requests we assume that 8.44 requests per second can be transmitted and a constrained device uses the CR2032 coin battery with charge capacity 225 mAh, processor spends $0.07\mu Ah(2.25ms * 16 * 13mA)$ of battery charge per second. In other words, it requires 38 days to drain the battery with this method, which means that this type of attack is almost unfeasible.

In order to analyze the feasibility of the battery-drain attack we assume the worst case that an attacker can perform a handshake in minimal handshake time (once in 0.5ms). S3K requires that the full DTLS handshake should be performed to authenticate the client. We measured that it takes 38.64 ms to reject the client with valid nonce and invalid key. In this case the processor spends $0.28\mu Ah(38.64ms * 2 * 13mA)$ of battery charge per second. In other words, it requires 9.3 days of continuous sending in order to drain the battery. Again this type of attack can be considered unfeasible.

A denial of service attack can be considered feasible and successful if an attacker can send requests to the resource server that make it unresponsive. An attacker can perform this either by sending requests that require significant time to be processed or requests that block shared resources. As with battery-drain attack we analyze if DoS attack is feasible either by sending key revocation requests or by performing a handshake. Processing of each key revocation request without SHA2 accelerator takes 2.3 ms. Hence, the processor can service 434.7 requests per second that is much more than 8.44 requests per second that can be serviced by the network.

In DTLS handshake *ChangeCipherSpec* message requires the longest processing time of 17.89 ms. Hence, the processor can service 55.90 requests per second. That is more than amount of requests that can be transmitted over the network. So the DoS attack cannot be performed on the DTLS handshake.

At the same time the problem comes from the DTLS protocol specification. The DTLS state machine is initialized on the Resource Server right after the *ClientHello* message with valid cookie is received. Since a constrained device can handle a very limited number of slots for DTLS sessions (in some cases the device can handle only 1 session) the potential attacker may perform a Denial of Service (DoS) attack by sending *ClientHello* messages and keeping all session slots busy.

VII. CONCLUSION

Mandated by CoAP and recommended by the OMA Lightweight M2M and the IPSO alliance, DTLS is becoming

the de facto security standard for the Internet of Things. However, only the certificate-based DTLS provides scalability but requires heavyweight PKI with certificate validation. Considering the constrained nature of IoT devices, the certificate-based DTLS mode is not feasible for the IoT. The other DTLS modes that are based on symmetric keys and raw public keys are comparatively lightweight but are not scalable to a large number of IoT devices.

We have proposed, implemented, and evaluated a scalable security solution with symmetric keys (S3K) for the IoT. We have shown that S3K is lightweight and feasible to use in resource-constrained devices, and at the same time it is scalable to a large number of IoT devices. Our S3K architecture neither requires pre-provisioning of multiple pre-shared keys nor the need for out-of-band validation of public keys.

As a proof of concept, in this paper we have implemented S3K for the DTLS pre-shared key mode. In future, we plan to implement S3K for the DTLS with raw public keys and also for the other cyber security protocols such as IPsec/IKE.

ACKNOWLEDGMENT

This work is funded by VINNOVA. Part of the work of Ludwig Seitz and Göran Selander has been carried out in the context of the CelticPlus project CyberWI with funding from VINNOVA.

REFERENCES

- [1] Tim Dierks and Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. Request For Comments (RFC) 5246, Internet Engineering Task Force (IETF), August 2008. <http://www.ietf.org/rfc/rfc5246.txt>.
- [2] Eric Rescorla. HTTP Over TLS. Request For Comments (RFC) 2818, Internet Engineering Task Force (IETF), May 2000. <http://www.ietf.org/rfc/rfc2818.txt>.
- [3] Eric Rescorla and Nagendra Modadugu. Datagram Transport Layer Security Version 1.2. Request For Comments (RFC) 6347, Internet Engineering Task Force (IETF), January 2012. <http://www.ietf.org/rfc/rfc6347.txt>.
- [4] Paul Wouters, Hannes Tschofenig, John Gilmore, Samuel Weiler, and Tero Kivinen. Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). Request For Comments (RFC) 7250, Internet Engineering Task Force (IETF), June 2014. <http://www.ietf.org/rfc/rfc7240.txt>.
- [5] Pasi Eronen and Hannes Tschofenig. Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). Request For Comments (RFC) 4279, Internet Engineering Task Force (IETF), December 2005. <http://www.ietf.org/rfc/rfc4279.txt>.
- [6] Clifford Neuman, Tom Yu, Sam Hartman, and Kenneth Raeburn. The Kerberos Network Authentication Service (V5). Request For Comments (RFC) 4120, Internet Engineering Task Force (IETF), July 2005. <http://www.ietf.org/rfc/rfc4120>.
- [7] Leslie Daigle and Olaf Kolkman. The OAuth 2.0 Authorization Framework. Request For Comments (RFC) 6749, Internet Engineering Task Force (IETF), October 2012. <http://www.ietf.org/rfc/rfc6749.txt>.
- [8] Carsten Bormann, Mehmet Ersue, and Ari Kernanen. Terminology for Constrained-Node Networks. RFC 7228, May 2014.
- [9] Cíntia Margi, Bruno de Oliveira, Gustavo de Sousa, Marcos Simplicio, Paulo Barreto, Tereza Carvalho, Mats Nälund, and Richard Gold. Impact of Operating Systems on Wireless Sensor Networks (Security) Applications and Testbeds. In *Proceedings of 19th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–6, August 2010.
- [10] Giacomo de Meulenaer, François Gosset, François-Xavier Standaert, and Olivier Pereira. On the energy cost of communication and cryptography in wireless sensor networks. In *Proceedings of the IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WIMOB)*, pages 580–585, October 2008.

- [11] Zach Shelby, Klaus Hartke, and Carsten Bormann. Constrained Application Protocol (CoAP). RFC 7252, June 2014.
- [12] Robert Hinden and Stephen Deering. IP Version 6 Addressing Architecture. Request For Comments (RFC) 4291, Internet Engineering Task Force (IETF), February 2006. <http://www.ietf.org/rfc/rfc4291.txt>.
- [13] Gabriel Montenegro, Nandakishore Kushalnagar, Jonathan Hui, and David Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944, September 2007.
- [14] IEEE Computer Society. IEEE Standard for Local and metropolitan area networks – Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs). IEEE Standard 802.15.4-2011, September 2011.
- [15] ZigBee Alliance. ZigBee Specification. ZigBee Document 053474r17, ZigBee Alliance, January 2008.
- [16] Bluetooth Special Interest Group. Specification of the Bluetooth System. Specification 4.1, December 2013.
- [17] Shahid Raza, Hossein Shafagh, Kasun Hewage, René Hummen, and Thiemo Voigt. Lite: Lightweight secure CoAP for the internet of things. *Sensors Journal, IEEE*, 13(10):3711–3720, 2013.
- [18] Shahid Raza, Simon Duquenooy, Joel Höglund, Utz Roedig, and Thiemo Voigt. Secure Communication for the Internet of Things - A Comparison of Link-Layer Security and IPsec for 6LoWPAN. *Security and Communication Networks*, Wiley, 7(12):2654–2668, December 2014.
- [19] René Hummen, Hossein Shafagh, Shahid Raza, Thiemo Voigt, and Klaus Wehrle. Delegation-based Authentication and Authorization for the IP-based Internet of Things. In *Sensing, Communication, and Networking (SECON), 2014 Eleventh Annual IEEE International Conference on*, pages 284–292. IEEE, 2014.
- [20] Shahid Raza, Linus Wallgren, and Thiemo Voigt. SVELTE: Real-time Intrusion Detection in the Internet of Things. *Ad Hoc Networks*, 11(8), nov 2013.
- [21] Ibrahim Ethem Bagci, Shahid Raza, Utz Roedig, and Thiemo Voigt. Fusion: coalesced confidential storage and communication framework for the iot. *Security and Communication Networks*, 2015.
- [22] Open Mobile Alliance. Lightweight Machine to Machine Technical Specification. Technical Specification OMA-TS-LightweightM2M-V1_020131105-D, Open Mobile Alliance, November 2013.
- [23] Jose Hernandez-Ramos, Marcin Pawlowski, Antonio Jara, Antonio Skarmeta, and Latif Ladid. Towards a Lightweight Authentication and Authorization Framework for Smart Objects. *IEEE Journal on Selected Areas in Communications*, 33(4):690–702, January 2005.
- [24] IEEE Computer Society. IEEE Standard for Local and Metropolitan Area Networks – Port-Based Network Access Control. IEEE Standard 802-1X-2010, IEEE Computer Society, February 2010.
- [25] Pablo Pereira, Jens Eliasson, and Jerker Delsing. An Authentication and Access Control Framework for COAP-based Internet of Things. In *Proceedings of the 40th Annual Conference of the IEEE Industrial Electronics Society (IECON)*, pages 5293–5299, Dallas, TX, USA, October 2014. IEEE Computer Society.
- [26] Carl Rigney, Allan Rubens, William Simpson, and Steve Willens. Remote Authentication Dial In User Service (RADIUS). RFC 2865, June 2000.
- [27] Stephen Kent and Randall Atkinson. IP Authentication Header. Request For Comments (RFC) 2402, November 1998. <http://www.ietf.org/rfc/rfc2402.txt>.
- [28] Texas instruments. cc2538em schematic rev 1.2.0. Technical report, Texas Instruments, December 2012.
- [29] Marius Ubostad. Smartrf06 evaluation board user’s guide. Technical report, Texas Instruments, May 2013.
- [30] Contiki: The open source os for the internet of things. <http://www.contiki-os.org/>. Accessed: 2015-05-27.
- [31] Matthias Kovatsch, Simon Duquenooy, and Adam Dunkels. A low-power coap for contiki. In *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*, pages 855–860. IEEE, 2011.
- [32] Tinydtls. <http://tinydtls.sourceforge.net/>. Accessed: 2015-05-27.
- [33] The gnutls transport layer security library. <http://gnutls.org/>. Accessed: 2015-05-27.
- [34] Openssl. <https://www.openssl.org/>. Accessed: 2015-05-27.
- [35] Matrixssl. <http://matrixssl.org/>. Accessed: 2015-05-27.
- [36] Adam Dunkels, Fredrik Österlind, Nicolas Tsiftes, and Zhitao He. Software-based on-line energy estimation for sensor nodes. In *Proceedings of the 4th Workshop on Embedded Networked Sensors, EmNets*, pages 28–32, January 2007.
- [37] Adam Dunkels. The contikimac radio duty cycling protocol. SICS Technical Report T2011:13, Swedish Institute of Computer Science, December 2011.

Shahid Raza Shahid Raza is a senior researcher at the SICS Swedish ICT Stockholm where he has been working since 2008. Shahid’s research interests include but are not limited to security and privacy in IPv6-connected IoT, interconnection of computing clouds and IoT, WirelessHART, the smart grid, and storage security. Shahid has completed his industrial PhD from the SICS Swedish ICT Stockholm and the Mälardalen University Västerås in 2013. He also holds a Technology of Licentiate degree from the Mälardalen University Västerås and a Master of Science degree from KTH The Royal Institute of Technology, Stockholm. See more at www.ShahidRaza.info

from INSA Lyon, France.

Ludwig Seitz Ludwig Seitz is a senior researcher at the Security Lab at SICS Swedish ICT in Lund since 2011. Ludwig Seitz has extensive experience in identity management and access control, as well as related standardization. He has contributed to the standardization process in the XACML technical committee between 2006-2010, and is currently contributing to the IETF standardization on Authentication and Authorization in Constrained Environments (ACE). Ludwig holds a M. Sc. in Computer Science from Karlsruhe University in Germany, and a Ph.D.

Denis Sitenkov Denis Sitenkov was master student and researcher at SICS Swedish ICT. He has many years of experience as a software engineer on different projects including Embedded Systems and M2M communication. His main area of interest is security and communication protocols for the Internet of Things. Denis received his Master Degree in Embedded Systems at 2014 from the Royal Institute of Technology, Stockholm.

Göran Selander Göran Selander is a Master Researcher at Ericsson Research with 15 years experience within security research. Göran has been working with a wide range of security topics and has a special interest in security protocols and authorization. His current main area is security research and standardization related the Internet of Things and the next generation mobile networks. He received his PhD in mathematics 1999 from the Royal Institute of Technology, Stockholm.