Preprint accepted to Elsevier Journal of AdHoc Networks in May 2013

# DTLS based Security and Two-Way Authentication for the Internet of Things[☆]

Thomas Kothmayr,Corinna Schmitt,Wen Hu,Michael Brünig,Georg Carle

*Department of Computer Science, Chair for Network Architectures and Services, Technische Universität München, Germany, {kothmayr,carle}@in.tum.de*

*CSIRO ICT Centre, Brisbane, Australia, {wen.hu,michael.bruenig}@csiro.au*

*Communication Systems Group (CSG), Institute for Informatics, University of Zurich, Switzerland, {schmitt}@ifi.uzh.ch*

**Abstract**

In this paper, we introduce the first fully implemented two-way authentication security scheme for the Internet of Things (IoT) based on existing Internet standards, specifically the Datagram Transport Layer Security (DTLS) protocol. By relying on an established standard, existing implementations, engineering techniques and security infrastructure can be reused, which enables easy security uptake. Our proposed security scheme is therefore based on RSA, the most widley used public key cryptography algorithm. It is designed to work over standard communication stacks that offer UDP/IPv6 networking for Low power Wireless Personal Area Networks (6LoWPAN). Our implementation of DTLS is presented in the context of a system architecture and the scheme's feasibility (low overheads and high interoperability) is further demonstrated through extensive evaluation on a hardware platform suitable for the Internet of Things.

[☆]Part of this work was published at the 7th IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp) in conjunction with IEEE LCN 2012 [1]. The extensions of this work include the analysis of the handshake behaviour under link layer packet loss (Section 5.2), a comparison with software implementations of RSA and elliptic curve cryptography (ECC) algorithms (Section 5.4), and a detailed case study (Section 6). This work was mostly done when Corinna Schmitt was with Technische Universität München [2].



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

## 1. Introduction

Today, there is a multitude of envisioned and implemented use cases for the IoT and wireless sensor networks (WSNs). It is desirable, in most of these scenarios, to also make the data globally accessible to authorized users and data processing units through the Internet. Naturally, much of the data collected in these scenarios, such as locations and personal IDs, is of a sensitive nature. Even seemingly inconspicuous data, such as the energy consumption measured by a smart meter, can lead to potential infringements on the users' privacy, e.g. by allowing an eavesdropper to conclude whether or not a user is currently at home. From an industry perspective, there is also a pressing need for security solutions based on standards. The market research firm Gartner, Inc. states in their report - *2012 Hype Cycle for the Internet of Things* - [3]: "The Internet of Things concept will take more than 10 years to reach the Plateau of Productivity - mainly due to security challenges, privacy policies, data and wireless standards, and the realization that the Internet of Things requires the build-out of a topology of services, applications and a connecting infrastructure." Regarding the infrastructure, security risks are aggravated by the trend toward a separation of sensor network infrastructure and applications [4, 5]. Therefore, a true end-to-end security solution is required to achieve an adequate level of security for the IoT. Protecting the data once it leaves the scope of the local network is not enough.

A similar scenario in the traditional computing world would be a user browsing the Internet over an unsecured WLAN. Attackers in physical proximity of the user can capture the traffic between the user and a web server. Countermeasures against such attacks include the establishment of a secured connection to the web server via HTTPS, the use of a VPN tunnel to securely connect to a trusted VPN endpoint and using wireless network security such as WPA.

These solutions are comparable to security approaches in the IoT area. Using WPA is similar to the traditional use of link layer encryption. The VPN solution is equivalent to creating a secure connection between a sensor node and a security end-point, which may or may not be the final destination of the sensor data. Establishing a HTTPS connection with the server is comparable to our approach: We investigate the use of the DTLS protocol in an end-to-end security architecture for the IoT. DTLS is an adaption of the widespread TLS protocol, used to secure HTTPS, for unreliable datagram transport. By choosing DTLS we have made three high-level design decisions:



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

**Implementation of a standards based design:** Standardization has helped the widespread uptake of technologies. Radio chips can rely on IEEE 802.15.4 for the physical and the MAC layer. The IPv6 Routing Protocol for Low power and Lossy Networks (RPL) or 6LoWPAN provide routing functionality and CoAP [6] defines the application layer. So far, no such efforts have addressed security in a wider context for the IoT.

**Focus on application-layer end-to-end security:** An end-to-end protocol provides security even if the underlying network infrastructure is only partially under the user's control. As the infrastructure for Machine-to-Machine (M2M) communication is getting increasingly commoditized, this scenario becomes more likely: The European Telecommunications Standards Institute (ETSI) is currently developing a standard that focuses on providing a "horizontal M2M service platform" [5], meaning that it plans to standardize the transport of local device data to a remote data center. For stationary installations security functionality could be provided by the gateway to the higher level network. However, such gateways would present a high-value target for an attacker. If the devices are mobile, for example in an logistics application, there may be no gateway to a provider's network that is under the user's control, similar to how users of smart phones connect directly to their carrier's network. Another example that favors end-to-end security is a multi-tenancy office building that is equipped with a common infrastructure for metering and climate-control purposes. The tenants share the infrastructure but are still able to keep their devices' data private from other members of the network. Using a protocol like DTLS, which is placed between transport and application layer, does not require that the infrastructure provider supports the security mechanism. It is purely in the hands of the two communicating applications to establish security. If the security is provided by a network layer protocol, such as IPsec, the same is true to a lower degree because the network stacks of both devices must support the same security protocol.

**Support for unreliable transport protocols:** Reliable transport protocols like TCP incur an overhead over simpler, unreliable protocols such as UDP. Especially for energy starved, battery powered devices this overhead is often too costly and TCP has been shown to perform poorly in low-bandwidth scenarios [7]. This is reflected in the design of the emerging standard CoAP, which uses UDP transport and defines a binding to DTLS for security [6]. By using DTLS in conjunction with UDP our approach does not force the application developer to use reliable transport - as would be the case if TLS would be used. It is still possible to use DTLS over transport protocols like TCP, since DTLS only assumes unreliable transport.



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

This is a weaker property than the reliability provided by TCP. However, the adaptations of DTLS for unreliable transport introduce additional overhead when compared to TLS. There might be a benefit in using TCP during the handshake phase but, as we point out in Section 5.2, the DTLS reliability mechanism should be adapted to the special requirements of constrained networks. A study of TCP's infuence on the handshake is therefore out of scope of this article.

The rest of the paper is organized as follows: We outline related work, mainly from the field of security in Wireless Sensor Networks (WSNs), in Section 2. WSNs are a suitable reference point because they are constrained in terms of computational power, available memory, energy consumption and network bandwidth. Section 3 provides the reader with an introduction to the DTLS protocol before we present our system architecture in Section 4. In order to assess the feasibility of using DTLS in a constrained environment we implemented a prototype on a constrained device. We thoroughly evaluate this implementation in Section 5 to identify areas in which the standard protocol could be modified to better meet the challenges of a WSN environment. In Section 6, we show a practical proof of concept in a building scenario. Our conclusion is given in Section 7.

## 2. Related Work

Traditionally, security protocols in sensor networks focus on link layer security, protecting data on a hop-by-hop basis. The simplest approach to link layer security consists of using a network-wide encryption key, which often is the case in ZigBee networks [8]. ZigBee also provides support for cluster and individual link keys. MiniSec [9] is another well known security mechanism for WSNs that provides data confidentiality, authentication and replay protection. As with ZigBee, the packet overhead introduced by MiniSec is in the order of a few bytes. The widespread TinySec link layer security mechanism is no longer considered secure [9].

Most security protocols do not include a mechanism for how encryption keys are distributed to the nodes. Keys are either loaded onto the nodes before setup or a separate key establishment protocol is used. Public key cryptography (PKC) is used in traditional computing to facilitate secure key establishment. However, public key cryptography, in particular the widespread RSA algorithm, has been considered too resource consuming for constrained devices. Some security protocols, such as Sizzle [10], advocate the use of the more resource efficient Elliptic Curve Cryptography (ECC) public key cryptosystem. Other research efforts, such as the secFleck [11] mote, provide support for faster RSA operations through hardware.



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

Approaches without PKC often rely on the pre-distribution of connection keys. Random key pre-distribution schemes, such as the q-composite scheme by Chan et al. [12], establish connections with a node's neighbors with a certain probability $p < 1$. Intuitively, pre-distributed key schemes such as this require a large amount of keys to be loaded onto the nodes before deployment. Depending on the method used, this approach is scaling in $\mathcal{O}(n^2)$ or $\mathcal{O}(n)$ where $n$ is the number of nodes in the network. The Peer Intermediaries for Key Establishment protocol (PIKE) achieves sub linear scaling in $\mathcal{O}(\sqrt{n})$ by relying on the other nodes as trusted intermediaries. While PIKE provides higher memory efficiency than random schemes, it still leaks additional key information when motes are captured.

Recently, more research into end-to-end security protocols for the IoT and WSNs is being conducted. As outlined in the introduction, such a protocol protects the message payload from the data source until it reaches its target. Because end-to-end protocols are usually implemented in the network or application layer, forwarding nodes do not need to perform any additional cryptographic operations since the routing information is transmitted in the clear. On the flip side, this means end-to-end security protocols do not provide the same level of protection of a network's availability as a link layer protocol could. One example of an end-to-end security protocol is Sizzle by Gupta et al. [10]. Sizzle is a compact web server stack providing HTTP services secured by SSL. It uses 160-bit ECC keys for key establishment which provide a similar level of security as 1024-bit RSA keys. In contrast to our work, it requires a reliable transport layer which has been shown to incur large performance penalties in low bandwidth situations [7]. Sizzle also omits two-way authentication: Only the Sizzle enabled node is authenticated by a remote, more resource rich, client. This is insufficient for machine to machine communication in the IoT. SSNAIL [13] makes similar design choices as Sizzle and performs an ECC handshake over reliable TCP transport. Similar to our implementation, SSNAIL is able to perform a full, two-way authenticated handshake but it still requires a reliable transport protocol.

Raza et al. [14] discuss how the IPsec protocol can be integrated into 6LoWPAN, the compressed IPv6 implementation used in most IP-enabled sensor networks. Their work focuses on how data transfer with IPsec can be made efficient in the context of 6LoWPAN. Regarding the Internet Key Exchange protocol (IKE), which is used for key establishment in IPsec networks, Raza et al. [15] discuss methods for reducing the headers to make IKE more suitable for constrained devices, but do not present a performance analysis alongside their proposal.



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

As mentioned in the introduction, CoAP is an application layer standardization effort for the Internet of Things. The current draft specifies a binding of CoAP to DTLS to achieve security [6]. Another proposal by Raza et al. aims to reduce the communication overhead of the DTLS headers through compression [16]. As with the work on IPsec, we are currently not aware of any publication evaluating the performance of DTLS over 6LoWPAN. Our work can thus support these efforts by providing a set of real-world measurements from our DTLS implementation.

## 3. The Datagram Transport Layer Security protocol

All messages sent via DTLS are prepended with a 13 bytes long DTLS record header. This header specifies the content of the message (e.g. application data or handshake data), the version of the protocol employed, as well as a 64-bit sequence number and the record length. The top two bytes of the sequence number are used to specify the epoch of the message which changes once new encryption parameters have been negotiated between client and server. Figure 1 shows the DTLS record header in white. The record header is either followed by the plaintext if no security has been negotiated yet, or by the DTLS block cipher. If a block cipher is used, the plaintext is prepended by a random Initialization Vector (IV), which has the size of the cipher block length. This protects against attacks where attackers can adaptively choose plaintext. The plaintext is followed by a Hash-based Message Authentication Code (HMAC) which allows the receiver to detect if the DTLS record has been altered. Finally, the message is padded to a multiple of the cipher block length. The area of the message shown in grey in Figure 1 is encrypted with the block cipher, striped parts are not used to calculate the HMAC. Unlike TLS, DTLS does not allow for stream ciphers because they are sensitive to message loss and reordering. Instead, DTLS uses block ciphers in the Cipher-Block Chaining (CBC) mode of operation.

The key material and cipher suite, consisting of a block cipher and a hash algorithm, are negotiated between client and server during the handshake phase which commences before any application data can be transferred. There are three types of handshake: unauthenticated, server authenticated and fully authenticated handshakes. During an unauthenticated handshake neither party authenticates with the other, and during a server authenticated handshake only the server proves its identity to the client. In a fully authenticated handshake the client has to authenticate itself to the server as well. In the following we will not consider the unauthenticated handshake because it provides no authenticity at all.



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

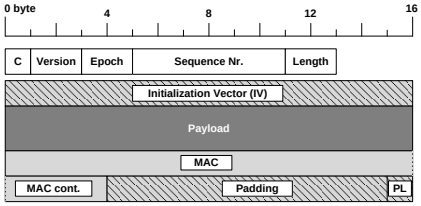


Figure 1: A DTLS record protected with CBC block cipher [1].

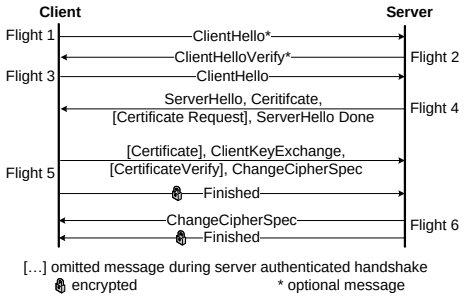


Figure 2: A fully authenticated DTLS Handshake [1, 2].

There are different algorithms that can be used for authentication in a DTLS handshake. Variants based on ECC have been shown in embedded networks [10]. Since we argue for a standard-based communication architecture for the IoT to promote *interoperability*, the rest of the paper will focus on authentication based on RSA. Because it is today's dominant PKC system [17] a suitable infrastructure for obtaining certificates from commercial Certificate Authorities (CA) is already in place.

Figure 2 shows a fully authenticated DTLS handshake. Individual messages are grouped into 'message flights' according to their direction and occurrence sequence. Flight 1 and 2 are an optional feature to protect the server against Denial-of-Service (DoS) attacks. The client has to prove that it can receive data as well as send data by resending its `ClientHello` message with the cookie sent in the `ClientHelloVerify` message by the server. The `ClientHello` message contains the protocol version supported by the client as well as the cipher suites that it supports. The server answers with its `ServerHello` message that contains the cipher suite chosen from the list offered by the client. The server also sends a X.509 certificate to authenticate itself followed by a `CertificateRequest` message if the server expects the client to authenticate. The `ServerHelloDone` message only indicates the end of flight 4. If requested and supported, the client sends its own certificate message at the beginning of flight 5. The `ClientKeyExchange` message contains half of the pre-master secret encrypted with the server's public RSA key from the server's certificate. The other half of the pre-master secret was transmitted unprotected in the `ServerHello` message. The keying material is subsequently derived from the pre-master secret. Since half of the pre-master secret is encrypted with the server's public key it can only complete the handshake if it is in possession of the private key matching the public key in the server certificate. Accordingly, in the `CertificateVerify` message the client authenticates itself by proving that it is in possession of the private key matching the client's public key.



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

It does this by signing a hashed digest of all previous handshake messages with its private key. The server can verify this through the public key of the client. The `ChangeCipherSpec` message indicates that all following messages by the client will be encrypted with the negotiated cipher suite and keying material. The `Finished` message contains an encrypted message digest of all previous handshake messages to ensure both parties are indeed operating based on the same, unaltered, handshake data. The server answers with its own `ChangeCiperSpec` and `Finished` message to complete the handshake.

## 4. A Standard Based End-to-End Security Architecture

Our system architecture is following the IoT model. We assume that the Internet is connected by IPv6 in the near future, and parts of it run 6LoWPAN. The Transport layer in 6LoWPAN is UDP which can be considered unreliable, the routing layer is RPL [18] or Hydro [7]. Our implementation uses Hqydro for routing, because at the time of writing our implementation code there was no avaliable RPL implementation for TinyOS. RPL has since been standardized in RFC 6550 and is distributed with newer versions of TinyOS. However, both routing protocols are similar enough so that a change should have negligible impact on the presented results. IEEE 802.15.4 is used for the physical and Media Access Control layer. Based on this protocol stack we chose DTLS as our security protocol which places it in the application layer on top of the UDP transport layer. Figure 3 summarizes the protocols used in our architecture.

Similar to security needs in traditional networks such as the Internet, we consider three security goals:

- **Authenticity:** Recipients of a message can identify their communication partners and can detect if the sender information has been forged.
- **Integrity:** Communication partners can detect changes to a message during transmission.
- **Confidentiality:** Attackers cannot gain knowledge about the contents of a secured message.

By choosing DTLS as the security protocol we can achieve these goals. DTLS is a modification of TLS for the unreliable UDP and inherits its security properties [19]. Using an application layer security protocol like DTLS as opposed to link or network layer security protocols such as MiniSec [9] has a number of advantages but also some drawbacks:



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

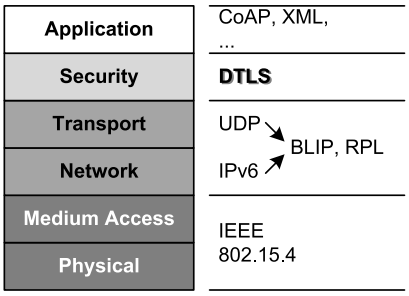


Figure 3: Protocol stack used in our security architecture [1, 2]

Lower layer security protocols do not provide end-to-end communication security. On each hop in a multi-hop network, data is decrypted on receipt and re-encrypted for forwarding. An attacker can thus gain access to all clear text data that passes through a compromised node. Scalability is often also an issue for these protocols because they need to establish a secured connection with each of their neighbors to form a mesh network, and cryptographic overhead occurs on each hop. On the other hand, in an end-to-end security protocol, cryptographic overhead occurs on the sender and receiver only. Compromised nodes provide an attacker with access to the measurement data from local nodes only. Routing algorithms are also agnostic of the payload protection, thus even nodes that have not established a secure connection can be used to forward packets to a subscriber/destination. One such scenario could be in an office building shared by multiple occupants (parties): each party subscribes to a part of the sensor readings only and wishes to keep the data they subscribed to private from other parties, yet they still may share a common communication network to reduce cost.

However, an application layer security protocol does not protect routing information. Adversaries can therefore analyze the traffic patterns of a network in clear text. They may even launch a DoS, worm hole, or resource consumption attack that lowers the availability of the network [20]. In this paper, we focus on end-to-end communication security, and rely on other schemes for securing lower communication layers [20].

Scenarios like the one above raise the need for proper authentication of data publishing devices and access control throughout the network. We therefore introduce an Access Control server (AC) into our architecture. The AC is a trusted entity and a more resource-rich server, on which the access rights for the publishers (=motes) of the secured network are stored. The identity of a default subscriber is usually preconfigured on a publisher before it is deployed. If any additional subscribers want to initialize a con-



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

nection with the publisher, they first have to obtain an access ticket from the AC. The AC verifies that the subscriber has the right to access the information avaliable from the publisher. The publisher then only has to evaluate the identity of the subscriber and verify the ticket it has received from the AC. Details of this scenario are subsequently omitted because they are out of scope of this paper. More details can be found in reference [2]. This requires a unique identity for a publisher in the network. In the Internet, identities are usually established via PKC and the identifiers provided through X.509 certificates. A X.509 certificate contains, among other information, the public key of an entity and its common name (e.g. my-bank.com). The certificate is signed by a trusted third party, called the Certificate Authority (CA), which serves two purposes: Firstly, the signature allows the receiver to detect modifications to the certificate. Secondly, it also states that the CA has verified the identity of the entity that requested the certificate.

Hu et al. showed that RSA, the most commonly used public key algorithm in the Internet, can be used in sensor networks with the assistance of a Trusted Platform Module (TPM), which costs less than 5% of a common sensor node [11]. A TPM is an embedded chip that provides tamper proof generation and storage of RSA keys as well as hardware support for the RSA algorithm. The certificate of a TPM equipped publisher and the certificate of a trusted CA must be stored on the publisher prior to deployment. For publishers that are not equipped with TPM chips we propose authentication via the DTLS pre-shared key cipher-suite, which requires a small number of random bytes, from which the actual key is derived, to be preloaded to the publishers before deployment. This secret must also be made available to the AC server which will disclose the key to devices with sufficient authorization. Figure 4 provides an overview of the proposed architecture which is described in detail in references [1] and [2].

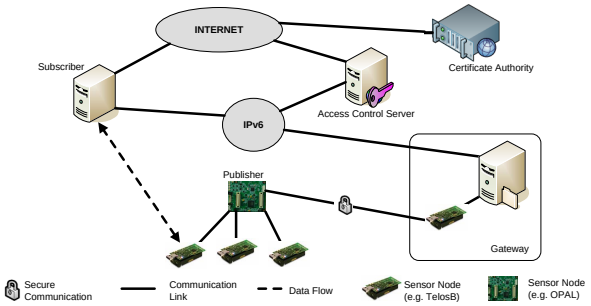


Figure 4: The overview of our proposed system architecture [2].



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

## 5. Evaluation

Previous work has already demonstrated techniques to reduce the protocol header overhead during data transmission [14] and has proven the feasibility of performing software encryption and hashing on the sensor node [9], also called mote. Indeed, even for DTLS, first proposals for a compressed header format have been made by Raza et al. recently [16]. Gupta et al. showed the feasibility of a server authenticated SSL handshake [10]. Therefore, the component of our security architecture that is currently least understood in the context of the IoT is the fully authenticated DTLS handshake, which includes both client and server authentication.

We have implemented a DTLS client that performs the DTLS handshake with an OpenSSL 1.0.0d server. The client is targeted at the OPAL sensor node [21] which features an Atmel SAM3U micro-controller and the Atmel AT97SC3203S TPM. It has 48 kB RAM and the micro-controller is clocked at 48 MHz in our implementation. In the following sections we will evaluate our implementation with regards to its performance during the handshake and data transmission, as well as its energy and memory consumption. Unless otherwise stated, the DTLS cipher suite performed was TLS-RSA-with-AES-128-CBC-SHA. AES-128 has been shown to be one of the fastest block ciphers on motes [22] and offers sufficient security. Furthermore, the cipher suite we chose is the required block cipher suite for DTLS from version 1.2 onwards. Other common cipher suites are either based on RC4, which is a stream cipher and thus not permitted by DTLS, or 3DES which is very slow and thus causes a large cryptographic overhead.

### *5.1. Data transfer latency*

In this section we will consider latency as a measure of the system's cryptographic performance. Figure 5 shows the round-trip time (RTT) for different sizes of plaintext data through a single hop network and a multi hop network with four hops. We measured the timing for the DTLS packets on the mote. Readings for pure plaintext data without any additional headers were obtained by issuing the `ping6` command on the subscriber.

A packet sent with both a SHA-1 HMAC and AES-128 encryption is denoted as "AES-128". The denotation "SHA-1" is used if a packet only contained a SHA-1 HMAC. The reading for 8 byte plaintext data is missing because the ICMP-Header and the timestamp sent by `ping6` are together at least 16 byte long.



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

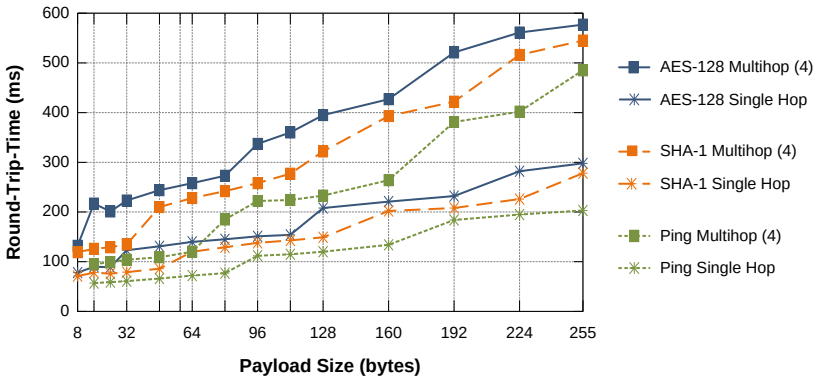


Figure 5: Average (n=100) packet round-trip time for different plaintext sizes [1]

The chart shows a linear increase of round-trip time with jumps occurring approximately every 100 bytes. These spikes can be attributed to the 128 byte maximum link layer frame size defined by IEEE 802.15.4 which includes header and trailer. These jumps occur earlier when sending DTLS protected packets due to the additional DTLS packet headers, the HMAC size and the explicit Initialization Vector in each packet. See Section 3 for more details on the packet structure.

Both the increased packet size and processing overhead lead to an increased end-to-end transmission latency for DTLS packets compared to plaintext packets. In the single hop scenario, transmission latency was increased by up to 95 ms for AES-128 and up to 75 ms for SHA-1 encryption which were an average increase of 62% and 35% respectively over the plaintext case. In the multi hop scenario, round trip times increased by a maximum of 163 ms and were 74% longer on average for AES-128 encrypted packets. Packets with a SHA-1 HMAC took up to 129 ms longer for the round-trip with an average of 40% more time being spent. The decreased performance for transmission latency is mostly due to the large packet overhead of up to 64 bytes which consists of 13 byte DTLS record header, 16 byte Initialization Vector, 20 byte HMAC, and up to 15 byte padding. Calculating a SHA-1 hash of a 255 byte plaintext message only takes 9 ms, encryption with AES-128 takes another 12 ms. Both operations do not contribute significantly to the overall transmission latency. This is consistent with the measurements for 16-byte plaintext (RTT of 58 ms) which increases to 90 ms with AES-128. Including the overhead of the DTLS record format, 16 plaintext bytes are expanded to a 77 byte message. Sending 80 bytes via `ping` requires 78 ms which indicates a computational overhead of around 12 ms in this case. A more detailed analysis of the transmission overhead from an energy perspective is provided in Section 5.4.



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

*5.2. Handshake latency*

Another performance indicator to consider is the latency introduced by performing a DTLS handshake. We measured the time from the beginning of the handshake establishment until a `Finished` message has been received on the client. In addition to using a 2048-bit key, we included the results for a 1024-bit key for comparison. Figure 6 shows the average latency for a fully authenticated and a server authenticated handshake. We conducted 15 measurements for each type of handshake. The bars show the average over these measurements, and the error bars show the standard deviation.

The large standard deviation is caused by our implementation behavior when message loss occurs. DTLS states that an implementation should wait for an answer for a set amount of time after sending a flight of messages. If it does not receive an answer during this period it retransmits the whole flight. We set this timeout value to 5 seconds to avoid unnecessary retransmissions in networks with a high end-to-end delay, which is common in a low power lossy network, and/or with energy limited thin clients that are slow to respond. DTLS implementations for the Internet often choose a retransmission timeout of one second or less. In general, we see that the time to execute a handshake is shorter for smaller RSA-keys and reduced by almost two seconds when client authentication is omitted in the handshake. We observed packet loss mainly in a multi-hop environment and when larger DTLS messages were being sent. This increases the total handshake time significantly because of the large DTLS retransmission timeout. However, total energy consumption of the client does not increase significantly because all TPM operations, which are the largest contributor to overall handshake energy costs (cf. Section 5.4), are only executed after successful receipt of all relevant server messages. Losing a packet with information obtained from the TPM does not lead to a repeated execution of the TPM operations because the resulting messages are buffered and can be retransmitted. During our experiments we did not see any failed handshake attempts. In earlier stages of development a lost `Finished` message from the server would cause the handshake to fail.

The client did not receive the expected `Finished` message and kept retransmitting its last message flight. The server, however, already considered the handshake to be complete and was waiting for bulk data transfer from the client, disregarding its repeated retransmissions of the handshake messages. DTLS 1.2 addresses this issue by always issuing a retransmission of the server's last message flight when it receives a `Finished` message from the client. We ported this behavior to our version of OpenSSL to address this problem.



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

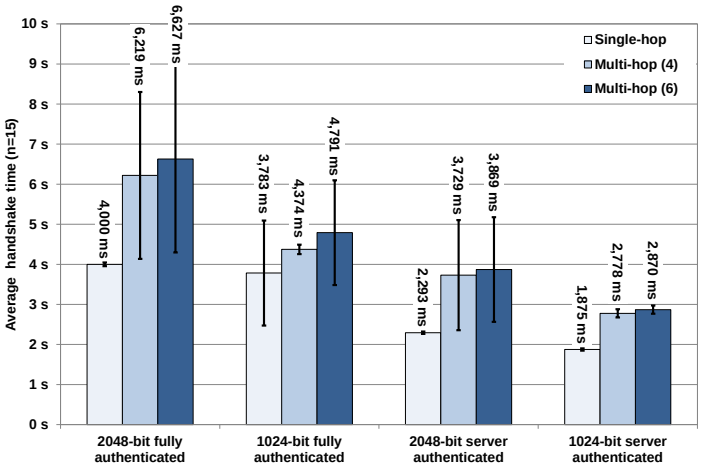


Figure 6: Time to complete different types of DTLS handshakes [1].

DTLS requires successful transmission of all handshake packets over an unreliable transport layer. Since it provides its own reliability mechanism during the handshake, network topology, congestion and link quality have a large impact on the time needed to complete a DTLS handshake. One parameter the programmer can influence to achieve better performance in lossy networks is the maximum transmission unit (MTU) for DTLS handshake packets which determines the size of individual handshake packet fragments. To study the influence of the MTU on overall handshake establishment time we introduced a random, artificial packet drop rate on the link layer and measured handshake completion times for various MTUs.

Figure 7 shows that even a small amount of packet loss has a large impact on overall handshake completion time. We consider each link layer packet to have an independent chance of being dropped, resulting in the total loss of all packets that follow. If we take a typical, fully authenticated DTLS handshake which causes 2,438 bytes of traffic as an example, there is a 72.26%[1] chance of packet loss while transmitting the 2,438 bytes of handshake payload at 5% link layer packet loss. If the link layer packet loss rate is 10%, there is a 92.82%[2] chance of packet loss occurring. In that case, the DTLS reliability mechanism is waiting for a timeout before resending the whole message flight [19]. As before, the retransmission timer was set to 5 seconds during our experiments.

[1] $P(Packetloss) = 1 - 0.95^{\lceil \frac{2{,}438 bytes}{100 bytes} \rceil} = 0.7226$

[2] $P(Packetloss) = 1 - 0.90^{\lceil \frac{2{,}438 bytes}{100 bytes} \rceil} = 0.9282$



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

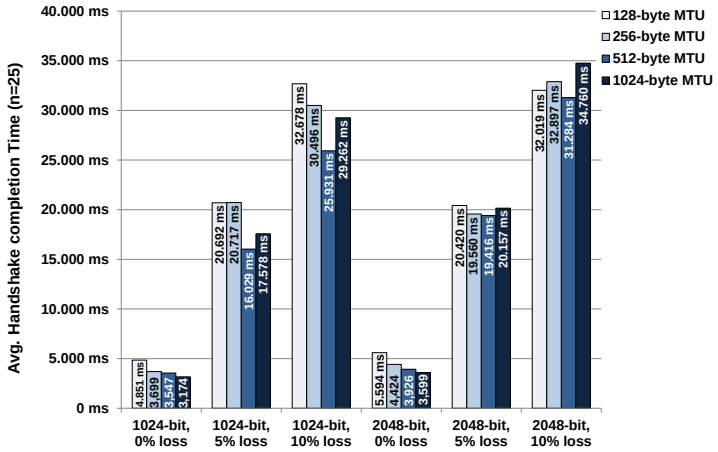


Figure 7: Handshake completion times with various amounts of artificial link layer packet loss and different MTUs.

We are considering uncorrelated packet loss in this evaluation, even tough packet loss is correlated in reality. The reasoning behind these figures is that we cannot know at which time during the handshake the interference that causes packet loss will start. We therefore use a constant probability of packet loss, which will cause all following fragments of the current message flight to be dropped. Additional, correlated packet loss before the next retransmission intervall has no adverse impact because the damage is already done.

The MTU influences the granularity at which handshake messages can be reassembled by the receiver. A small MTU splits large handshake messages into many different packets, allowing the receiver a fine grained reassembly if packets are lost. Since every new packet has to bear the DTLS header, the overall amount of traffic increases, which in turn increases the probability of packet loss. A larger MTU splits messages into fewer packets which reduces the probability of packet loss because there is less network traffic. However, if packet loss does occur, reassembly cannot be done as fine grained as with a smaller MTU. Figure 7 shows that a MTU of 512 bytes seems to strike the best balance between reassembly and network traffic in our experiments.

### *5.3. Memory*

In order to determine the static memory allocation to individual components of our implementation we analyzed the entries in the symbols table of the OPAL binary after compilation. Memory has been measured for a fully authenticated handshake with 2048-bit RSA keys. This type of handshake has the largest memory requirements since it needs more code and buffer



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

space for the client's `Certificate` and `CertificateVerify` messages. We divide the memory consumption into six, respectively seven categories as illustrated in Table 1. Additionally we measured the maximum stack size by filling the stack with a dummy variable directly after boot and analyzing how much of that continuous memory block had been overwritten after a successful DTLS handshake. The first subtotal of Table 1 only considers static memory allocation. Because it currently contributes a significant portion of overall stack use, we have implemented two prototypical methods of initializing the client certificate. The method represented by "Stack Minimum" directly sets each individual Byte of the outgoing message buffer to the matching value from the Certificate. The drawback is a increased ROM use because the code basically contains hundereds of statements in the form `buffer[x] = 0xff`. The "Stack Maximum" method initializes the outgoing message buffer from a temporary array which is filled from a hardcoded, anonymous array, e.g. `uint8_t[CERT_LEN] = {0xff, 0xff, 0xff, ...}`. In production the certificate would usually be read from the mote's flash memory which should fall somewhere in between the figures from these two approaches.

| | RAM (bytes) | ROM (bytes) |
|---|---|---|
| Cryptography | 541 | 10,838 |
| DTLS Messages | 1,174 | 2,568 |
| DTLS Network | 4,294 | 5,672 |
| TPM | 4,321 | 4,928 |
| BLIP | 6,352 | 9,298 |
| Application | 166 | - |
| System | 991 | 30,075 |
| **Total Data + BSS** | **17,839** | **63,379** |
| Stack Minimum | 1,098 | 0 |
| Stack Maximum | 2,300 | 3,936 |
| **Total** | **18,937 - 20,139** | **63,379 - 67,315** |

Table 1: RAM and ROM usage by component [1, 2].

In total approximately 20 kB of RAM and 67 kB of ROM is required for the implementation. The BLIP implementation requires most of the resources, followed by TPM drivers and DTLS networking code. Overall, the implementation is still below the 48 kB of RAM and 256 kB of program memory provided by OPAL [1, 2].

### *5.4. Energy consumption*

We measured the energy consumption during the handshake phase across a 10Ω resistor with an oscilloscope. This yielded a value for the electric potential which can be converted into a value for the current draw by dividing it through the value of the resistance (10Ω).



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

The energy costs can then be calculated as $\frac{U_{probe}}{R} \times t \times U_{battery}$. $U_{probe}$ is the measured voltage, $R = 10\Omega$ is the value of the resistor, $t$ is the transaction time, and $U_{battery} = 3.998V$ is the battery voltage. Table 2 shows the energy consumption during a typical execution of different handshake types. We use a 2048-bit RSA key because 1024-bit keys are not recommended for future deployments [23]. Values for current draw in Table 2 specify the amount that each component contributes to the total current draw. Figure 8 shows a capture from the oscilloscope for a 2048-bit RSA fully authenticated handshake. [1, 2]

| | Current | Fully authenticated handshake | Server authenticated handshake |
|---|---|---|---|
| Computation | 30 mA | 35 ms, 4.18 mJ | 33 ms, 3.95 mJ |
| Radio TX | 18 mA | 242 ms, 17.4 mJ | 70 ms, 5.03 mJ |
| TPM Start | 52.2 mA | 836 ms, 174.46 mJ | 836 ms, 174.5 mJ |
| TPM TWI | 43.6 mA | 688 ms, 120.0 mJ | 476 ms, 83.0 mJ |
| TPM Verify | 51.8 mA | 59 ms, 12.2 mJ | 56 ms, 11.6 mJ |
| TPM Encrypt | 51.8 mA | 39 ms, 8.07 mJ | 40 ms, 8.28 mJ |
| TPM Sign | 52.2 mA | 726 ms, 151.5 mJ | - |
| Total minimum | | 487.8 mJ | 286.4 mJ |
| CPU idle | 11.4 mA | 3965 ms, 180.7 mJ | 2265 ms, 103.2 mJ |
| Radio idle | 18 mA | 3758 ms, 270.4 mJ | 2228 ms, 160.3 mJ |
| Total | | 939.0 mJ | 549.9 mJ |

Table 2: Transaction time / energy consumption of DTLS handshake (2048-bit key) [1, 2]

We chose to neglect the contribution of the radio and micro-controller in further discussion, which have been marked as 'CPU idle' and 'Radio idle' in Table 2. Both can be considerably reduced by using power saving techniques, e.g. by using the TinyOS Low Power Listening (LPL) Media Access Control layer for the radio (less than 1% radio duty cycles have been reported by the literature repeatedly), and setting the micro-controller into a lower power state where it consumes less than 15 $\mu$A for SAM3U[3]. However, the transmission costs of messages increases significantly if LPL is activated. This tradeoff is subject to the design and configuration of each deployed network. For better comparison we view the idle energy use as outside of our field of control and focus on the energy costs which will ocurr in any case. Sending messages ('Radio TX') and performing cryptographic operations ('Computation') contribute very little to the overall energy costs that are directly dependent on our DTLS implementation. The total cost is then largely bound by the energy usage of the TPM.

[3]ATMEL, Datasheet SAM3U Series: `http://www.atmel.com/Images/doc6430.pdf`



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

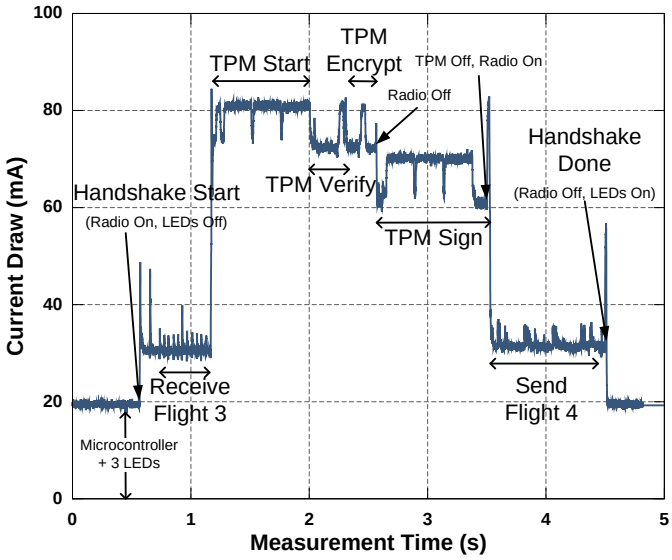


Figure 8: Current draw for a fully authenticated DTLS handshake [1, 2]

As can be seen in Figure 8, 'TPM Start' and 'TPM Sign' are the longest consecutive operations. The TPM is performing an operation with its RSA private key in 'TPM Sign' which is more complex than that with a RSA public-key. During the 'TPM Start' phase the TPM performs a series of internal self tests to detect tampering and unauthorized commands. The second large block is 'TPM TWI' which describes the amount of time that is spent passing data to the TPM and receiving data from it via the TWI bus clocked at 100 kHz. It shows as a lower current draw in Figure 8. It can be seen directly after the end of the 'TPM Start' sequence and before the short spike in 'TPM Verify'. The spike is the actual verification operation performed by the TPM. Similarly, the actual 'TPM Encrypt' operation is the spike that follows another section of data transfer on the TWI bus. During 'TPM Verify' the TPM uses the stored key of a CA to verify the server certificate presented during the handshake. The 'TPM Encrypt' operation is used to encrypt a nonce with the server's public key. If the mote is expected to authenticate itself during the handshake, it performs a 'TPM Sign' operation to sign a hash over all previous handshake messages with it's RSA private key. Since a server authenticated handshake does not require the expensive 'TPM Sign' operation it uses significantly less energy but also provides weaker overall authentication since an attacker could impersonate a mote toward the server. Communication time is also shorter since the sensor node does not send its certificate. [1, 2]



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

If the mote is powered by two AA 2,800-mAh batteries, they have an energy of approximately 30,240 Joule. If 5% of the energy is used for DTLS handshakes for (re)keying purposes, which happen once per day, it could last for more than 8.5 years for a fully authenticated handshake at 487.8 mJ each, or more than 14.5 years for a server authenticated handshake at 286.4 mJ each. As stated earlier, the calculation of a SHA-1 hash for 255 bytes takes 9 ms and encryption with AES-128 another 12 ms. Given the current draw for computation of 30 $mA$ at 48 MHz clock speed from Table 2, this results in the order of 9.9 $\mu J$ per Byte. [1, 2]

The energy consumption after the completion of the handshake is closely related to the latency values from Figure 6 which portrait the influence of the network and processing overhead introduced by DTLS. The increase in latency naturally also leads to an increase in energy consumption, since the radio has to be held in the transmitting state for longer, preventing it from entering a sleep state. Figure 9 shows the overhead in percent that occurs when a plaintext of a given size is encrypted and sent in a secure DTLS record. The baseline for this comparison is the time it would take to send the plaintext without any additional headers or other meta data.

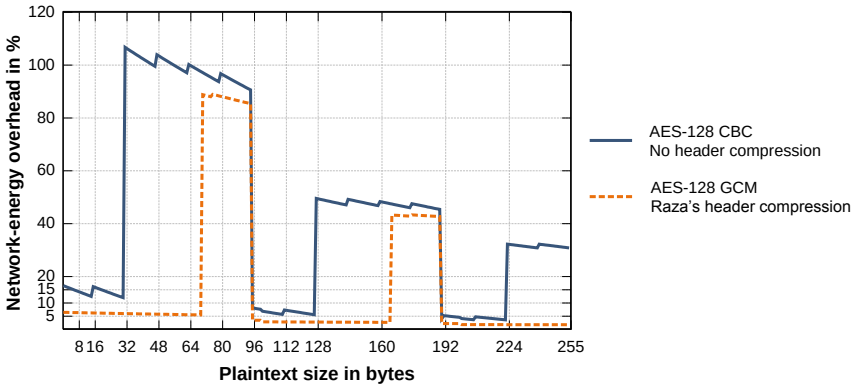


Figure 9: Network energy overhead caused by the DTLS record format.

We assume that the energy cost to send a message with length $x$ via BLIP follows a discontinuous piecewise linear function: $c(x, a, b) = \lceil \frac{x}{100} \rceil * a + x * b$. Here, $a$ represents the amount of energy needed to access the medium for one IEEE 802.15.4 message and sending the preamble and all other fixed energy costs for one message. The energy required for transmitting one byte of payload without the fixed costs is represented by $b$. The constant 100 is the maximum link layer message length defined by BLIP. Since we are only interested in the relative overhead, we ignore the current draw and only



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

analyze the relation between message length and time. For this purpose we used the round-trip times measured in Figure 6 for a simple ping and divided them by two. We then used Matlab to find the minimum of our error function $err(a,b) = \sum_{x \in M} \| \frac{c(x,a,b) - t(x)}{x} \|$ where $M$ is the set of plaintext lengths for which we have obtained measurement times and $t(x)$ returns the measured time for a plaintext length $x$. This optimization returned $a = 27.368$ and $b = 0.072$. With these results we could then calculate the approximate time required to send plaintext and larger DTLS records for the same amount of plaintext.

Figure 9 shows that the overhead introduced by the DTLS record format is under 17% for small plaintext lengths. It raises to over 100% when the DTLS record won't fit into a single link-layer packet anymore. BLIP then has to fragment the packet and bear the expensive medium access a second time. One way to reduce the network overhead is reducing the size of DTLS records. Our proposal is to employ the header compression detailed by Raza et al. [16]. This reduces the size of a DTLS record header from 13 to 5 bytes. Further savings are possible if the block cipher mode of operation is changed from CBC to Galouis/Counter mode of operation (GCM). The plaintext encrypted by GCM will always lead to a ciphertext of the same length [24]. Since GCM belongs to the class of block cipher modes called Authenticated Encryption with Associated Data (AEAD) the SHA-1 HMAC is no longer necessary. Instead, GCM can be used directly to authenticate the data and associated headers. The 20 byte SHA-1 HMAC is thus replaced by the maximum length GCM auth tag which requires 16 bytes. Additionally, the explicit IV of Figure 1 is no longer necessary because GCM is not susceptible to the vulnerability that makes the IV necessary. The maximum DTLS record overhead can thus be reduced from 64 bytes down to 21 bytes: Five bytes for the compressed record header plus the 16 byte GCM auth tag. Figure 9 shows that this more than doubles the area in which a DTLS record only incurs little overhead over sending the plaintext directly.

| | Current | Computation time | Energy consumption |
|---|---|---|---|
| RSA - Public Key @ 48 MHz | 30 mA | 440 ms | 52.8 mJ |
| RSA - Private Key (high memory) @ 48 MHz | 30 mA | 4,725 ms | 566.7 mJ |
| RSA - Private Key (low memory) @ 48 MHz | 30 mA | 14,895 ms | 1,786 mJ |
| **Handshake RSA total @ 48 MHz** | **30 mA** | **5,165 ms** | **619.5 mJ** |
| RSA - Public Key @ 96 MHz | 48 mA | 221 ms | 42.4 mJ |
| RSA - Private Key (high memory) @ 96 MHz | 48 mA | 2,362 ms | 453.3 mJ |
| RSA - Private Key (low memory) @ 96 MHz | 48 mA | 7,447 ms | 1,429 mJ |
| **Handshake RSA total @ 96 MHz** | **48 mA** | **2,583 ms** | **495.7 mJ** |

Table 3: Software RSA (2048-bit key) on OPAL. One Private Key and two Public Key operations are required for a handshake.



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

In order to put the TPM energy consumption and processing time in context, we also performed measurements of RSA and ECC in software. The RSA and ECC TinyOS modules available to us did not support 2048-bit RSA keys or their respective ECC equivalent. We therefore ported the RSA and ECC implementation of the open source project CyaSSL[4] to TinyOS. This port includes many of the optimization techniques adopted in TinyECC [25], such as Barrett Reduction, Sliding Window multiplication, Shamir's Trick and others. It does not, however, include inline assembly instructions to speed up natural number operations. Our implementation is made available to the TinyOS community under the GPLv2 license[5]. Table 3 shows the results for individual RSA operations with a 2048-bit RSA key performed in software. The figures for the handshake only pertain to the DTLS client, as was the case in our previous evaluations.

With a clock speed of 48 MHz, the software implementation requires more than twice as much time as the TPM and almost 1.5 times the amount of energy. The respective values for the TPM where 2,348 ms and 466.2 mJ. This advantage is dimished when the TPM is compared to software RSA being performed at 96 MHz, where both require roughly the same amount of time and energy. The RSA implementation still has room for improvement through embedded Assembler code and could thus be made more time and energy efficient than the TPM on our platform. However, the TPM still provides secure storage of the RSA-key, which cannot be achieved by software means, and the implementation complexity and RAM requirements of the TPM drivers are far less than those of a software RSA implementation. Additionally, newer versions of our TPM chip have more than halved the computation time for 2048-bit RSA keys.

| | Current | Computation time | Energy consumption |
|---|---|---|---|
| EC-DH @ 48 MHz | 30 mA | 387 ms | 46.4 mJ |
| ECDSA sign @ 48 MHz | 30 mA | 432 ms | 51.8 mJ |
| ECDSA verify @ 48 MHz | 30 mA | 795 ms | 95.4 mJ |
| **Handshake ECC total @ 48 MHz** | **30 mA** | **1,614 ms** | **193.6 mJ** |
| EC-DH @ 96 MHz | 48 mA | 187 ms | 35.8 mJ |
| ECDSA sign @ 96 MHz | 48 mA | 205 ms | 39.3 mJ |
| ECDSA verify @ 96 MHz | 48 mA | 380 ms | 72.9 mJ |
| **Handshake ECC total @ 96 MHz** | **48 mA** | **772 ms** | **92.6 mJ** |

Table 4: Software ECC over 224-bit prime curve (secp224r1) on OPAL. One of each operation is required for a handshake.

[4]Embedded SSL Library: `http://www.yassl.com/yaSSL/Products-cyassl.html`
[5]Source: `http://www-db.in.tum.de/~kothmayr/tinypkc`



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

If secure storage of a mote's private key is not a design goal, we recommend a software implementation of ECC instead. As Table 4 shows, it requires far less time and energy than either solution for RSA. The figures given were computed over the NIST named curve secp224r1, also known as NIST P-224. It provides equivalent security to a 2048-bit RSA key.
The operations performed during the DTLS handshake are Elliptic Curve Diffie-Hellman (EC-DH) for key-agreement followed by a two-way authentication via the Elliptic Curve Digital Signature Algorithm (ECDSA) to avoid Man-in-the-Middle attacks.

## 6. Case Study

In the department of Computer Science at the Technische Universität München the TinyIPFIX protocol was developed in order to support an efficient data transmission in a wireless sensor network with constrained hardware. One of the application scenarios is building automation where different environmental data, such as temperature, sound, light, and humidity, is monitored [2].

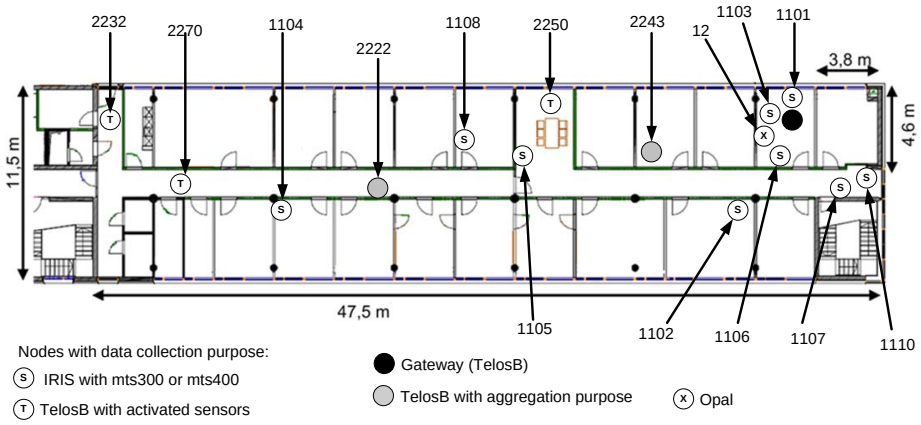


Figure 10: Deployed wireless sensor network at the Computer Science Department [2].

The TinyIPFIX protocol is based on the IETF Standard IPFIX which was developed for monitoring in large Peer-to-Peer networks. It is interesting for sensor networks because it is easy to parse and has a high transmission efficiency and little overhead due to its push-protocol characteristic and its template-based design [26]. In sensor networks the data is measured periodically in pre-defined intervals and often processed and aggregated in the network in order to save network traffic on the way to the data sink.



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

TinyIPFIX supports these properties and is described in detail in references [26] and [2]. In order to provide more security, the established wireless sensor network was extended by sensor hardware performing DTLS security. As before, we chose the OPAL node [21].

The TinyIPFIX protocol introduced earlier is included on the application layer in the performed solution of the department, which allows an independent functionality to the underlying layers. Thus, it is straightforward to integrate a DTLS solution into the network while still using TinyIPFIX as the application protocol of choice. However, our current implementation requires more resources than smaller motes, such as the TelosB mote, have to offer. Thus, the network is subdivided into clusters where the OPAL node works as a cluster head. It can also perform the in-network message aggregation to reduce network overhead.

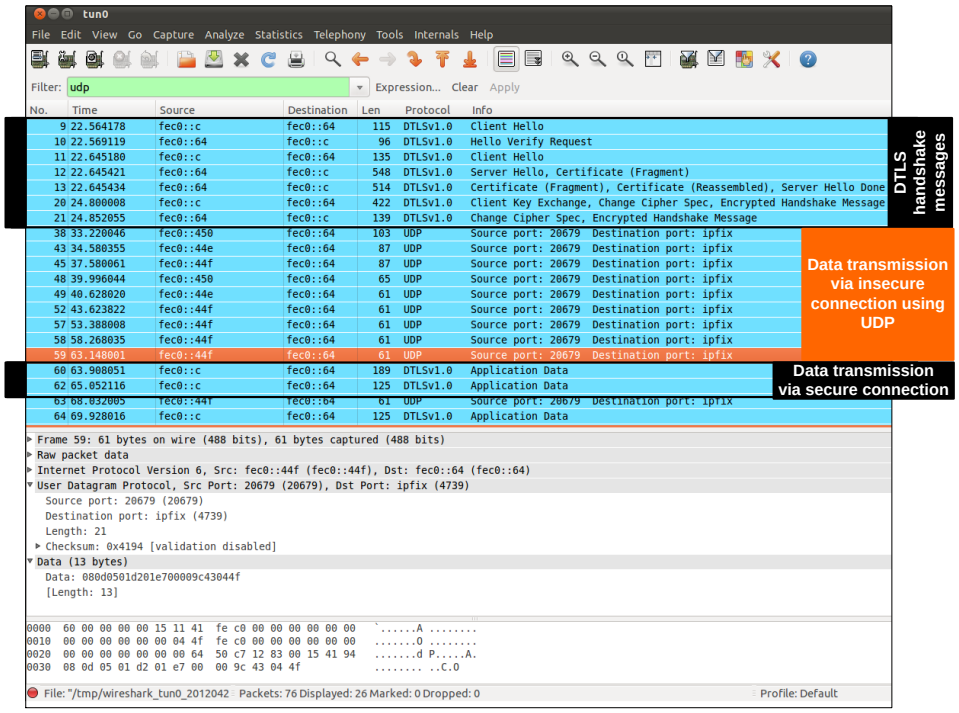


Figure 11: Snap shot of the communication in the WSN performing DTLS [2].



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

Figure 11 shows a Wireshark snap shot of the afore mentioned network. At the beginning, the OPAL node performs a DTLS handshake (marked in black) with the data sink in order to establish a secure channel. After the successful handshake messages transmitted via UDP are recorded. In this phase the clusterhead (IP = fec0::c) has not yet bound the data collectors (IP = {fec0::44f,fec::44e,fec::450}) to itself. As Figure 10 shows, the network consists of 15 nodes in total with three free data collectors. In the presented deployment, the OPAL node performs message aggregation with degree two, meaning it aggregates two incoming data messages into one outgoing message. As the Wireshark snap shot shows, the nodes with IP fec0::450 and fec0::44e win the competition and from recorded message no.60, respectively no.62, onwards they are connected to the clusterhead and can transmit their data via the DTLS secured channel. The node with IP fec0::44f still uses an unsecured UDP connection to the global data sink (marked in orange) [2].

## 7. Conclusion

We have introduced a standard based security architecture with two-way authentication for the IoT. The authentication is performed during a fully authenticated DTLS handshake and based on an exchange of X.509 certificates containing RSA keys. The extensive evaluation, based on real IoT systems, shows that our proposed architecture provides message integrity, confidentiality and authenticity with affordable energy, end-to-end latency and memory overhead. This shows that DTLS is a feasible security solution for the emerging IoT. We consider a fully authenticated handshake with strong security through 2048-bit RSA keys feasible for sensor nodes equipped with a TPM chip, since a fully authenticated, RSA based handshake consumes as little as 488 mJ. The memory requirement of under 20 kB RAM are well below the 48 kB of memory offered by our sensor node. Sensor nodes without a TPM chip forego protection against physical tampering, but can still perform a DTLS handshake based on ECC which could be performed on our platform with little more than 100 mJ of energy usage. Previous work has demonstrated techniques to minimize packet headers for similar protocols [14]. We plan to apply these techniques to DTLS in future work together with an Authenticated Encryption with Associated Data (AEAD) mode of operation to achieve the reduction in network overhead we have outlined in Section 5.4. Another focus will be the inclusion of more constrained nodes without a TPM in our architecture, for which we plan to use a variant of the DTLS pre-shared key cipher suites.



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

## 8. Acknowledgement

This presented work was supported by two projects partly funded by the German Federal Ministry of Education and Research: the SODA project under grant agreement no. 01IS09040A and the AutHoNe project under grant agreement no. 01BN070[2-5].

[1] T. Kothmayr, C. Schmitt, W. Hu, M. Brünig, G. Carle, A DTLS Based End-To-End Security Architecture for the Internet of Things with Two-Way Authentication, in: Proceedings of the 37th IEEE Conference on Local Computer Networks, LCN, 2012.

[2] C. Schmitt, Secure Data Transmission in Wireless Sensor Networks, Ph.D. thesis, Technische Universität München, Department of Computer Science (February 2013).

[3] H. LeHong, Hype Cycle for the Internet of Things, 2012, Tech. rep., Gartner Inc. (2012).

[4] I. Leontiadis, C. Efstratiou, C. Mascolo, J. Crowcroft, SenShare: Transforming Sensor Networks into Multi-application Sensing Infrastructures, in: Wireless Sensor Networks, Vol. 7158 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2012, pp. 65–81.

[5] ETSI TR 102681, Machine-to-Machine Communications (M2M); Smart Metering Use Cases, http://www.etsi.org (May 2010).

[6] Z. Shelby, K. Hartke, C. Bormann, B. Frank, Constrained Application Protocol (CoAP), IETF draft, RFC Editor (March 2013).
URL `http://tools.ietf.org/html/draft-ietf-core-coap-14`

[7] S. Dawson-Haggerty, A. Tavakoli, D. Culler, Hydro: A Hybrid Routing Protocol for Low-Power and Lossy Networks, in: Proceedings of the 1st IEEE International Conference on Smart Grid Communications, SmartGridComm, 2010, pp. 268–273.

[8] D. Raymond, S. Midkiff, Denial-of-Service in Wireless Sensor Networks: Attacks and Defenses, Pervasive Computing 7 (1) (2008) 74 – 81.

[9] M. Luk, G. Mezzour, A. Perrig, V. Gligor, MiniSec: A Secure Sensor Network Communication Architecture, in: Proceedings of the 6th International Conference on Information Processing in Sensor Networks, IPSN, 2007, pp. 479–488.



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

[10] V. Gupta, M. Wurm, Y. Zhu, M. Millard, S. Fung, N. Gura, H. Eberle, S. C. Shantz, Sizzle: A Standards-based End-to-End Security Architecture for the Embedded Internet, Pervasive Mob. Comput. 1 (2005) 425–445.

[11] W. Hu, H. Tan, P. Corke, W. C. Shih, S. Jha, Toward Trusted Wireless Sensor Networks, ACM Transactions on Sensor Networks 7 (2010) 5:1–5:25.

[12] H. Chan, A. Perrig, D. Song, Random Key Predistribution Schemes for Sensor Networks, in: Proceedings of Symposium on Security and Privacy, 2003, pp. 197–213.

[13] W. Jung, S. Hong, M. Ha, Y.-J. Kim, D. Kim, SSL-Based Lightweight Security of IP-Based Wireless Sensor Networks, International Conference on Advanced Information Networking and Applications Workshops (2009) 1112–1117.

[14] S. Raza, T. Voigt, U. Roedig, 6LoWPAN Extension for IPsec, in: Proceedings of the Interconnecting Smart Objects with the Internet Workshop, 2011.

[15] S. Raza, T. Voigt, V. Jutvik, Lightweight IKEv2: A Key Management Solution for both the Compressed IPsec and the IEEE 802.15.4 Security, in: Proceedings of the IETF Workshop on Smart Object Security, 2012.

[16] S. Raza, D. Trabalza, T. Voigt, 6LoWPAN Compressed DTLS for CoAP, in: Proceedings of the 8th IEEE International Conference on Distributed Computing in Sensor Systems, 2012.

[17] R. Watro, D. Kong, S. Cuti, C. Gardiner, C. Lynn, P. Kruus, TinyPK: Securing Sensor Networks with Public Key Technology, in: Proceedings of the 2nd ACM Workshop on Security of Ad Hoc and Sensor Networks, SASN, 2004, pp. 59–64.

[18] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, R. Alexander, RPL: IPv6 Routing Protocol for Low power and Lossy Networks, RFC 6550, RFC Editor (March 2012). URL `http://www.rfc-editor.org/rfc/rfc6550.txt`

[19] N. Modadugu, E. Rescorla, The Design and Implementation of Datagram TLS, in: Proceedings of the Network and Distributed System Security Symposium, NDSS, 2004.



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003

[20] P. Ning, A. Liu, W. Du, Mitigating DoS Attacks against Broadcast Authentication in Wireless Sensor Networks, ACM Transactions on Sensor Networks 4 (2008) 1:1–1:35.

[21] R. Jurdak, K. Klues, B. Kusy, C. Richter, K. Langendoen, M. Brünig, OPAL: A Multiradio Platform for High Throughput Wireless Sensor Networks, IEEE Embedded Systems Letters 3 (4) (2011) 121–124.

[22] J. Großschädl, S. Tillich, C. Rechberger, M. Hofmann, M. Medwed, Energy Evaluation of Software Implementations of Block Ciphers under Memory Constraints, in: Proceedings of the Conference on Design, Automation and Test in Europe, 2007, pp. 1110–1115.

[23] E. Barker, W. Barker, W. Burr, W. Polk, M. Smid, NIST SP800-57: Recommendation for Key Management - Part 1: General(Revised), Tech. rep., NIST (March 2007).

[24] D. A. McGrew, J. Viega, The Galois/Counter Mode of Operation (GCM), NIST Modes Operation Symmetric Key Block Ciphers.

[25] A. Liu, P. Ning, TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks, in: Proceedings of the 5th International Conference on Information Processing in Sensor Networks, EWSN, 2008, pp. 245 –256.

[26] T. Kothmayr, C. Schmitt, L. Braun, G. Carle, Gathering Sensor Data in Home Networks with IPFIX, in: Proceedings of the 7th European conference on Wireless Sensor Networks, EWSN, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 131–146.



Journal version: http://dx.doi.org/10.1016/j.adhoc.2013.05.003