Key Update for the IoT Security Standard OSCORE

Rikard Höglund, Marco Tiloca, Simon Bouget, Shahid Raza
Cybersecurity Unit - RISE Research Institutes of Sweden
{rikard.hoglund, marco.tiloca, simon.bouget, shahid.raza}@ri.se

***Abstract*—The standard Constrained Application Protocol (CoAP) is a lightweight, web-transfer protocol based on the REST paradigm and specifically suitable for constrained devices and the Internet-of-Things. Object Security for Constrained RESTful Environment (OSCORE) is a standard, lightweight security protocol that provides end-to-end protection of CoAP messages. A number of methods exist for managing keying material for OSCORE, as to its establishment and update. This paper provides a detailed comparison of such methods, in terms of their features, limitations and security properties. Also, it especially considers the new key update protocol KUDOS, for which it provides a more extended discussion about its features and mechanics, as well as a formal verification of its security properties.**

***Index Terms*—IoT, key update, key establishment, lightweight, OSCORE, CoAP, secure communication, end-to-end, IETF**

I. Introduction

In the Internet-of-Things (IoT) vision, a multitude of devices (including sensors, actuators, and everyday-life objects) have communication capabilities and are Internet-enabled. These are continuously increasing in number, are predicted to approach 60 billion by 2025 [1], and many of them are different from conventional, general-purpose computers.

It is crucial that such devices implement and correctly use appropriate and high-level security solutions, thus practically avoiding being a target and vector of cybersecurity attacks. This is particularly challenging in the IoT, as most of such connected devices are expected to be constrained in terms of memory availability, computing power, access to the communication medium, and energy budget (e.g., if battery-powered).

The standard Constrained Application Protocol (CoAP) [2] is a lightweight, web-transfer protocol based on the REST paradigm, that a client and a server peers can use to communicate at the application layer. CoAP was specifically designed to be suitable for constrained devices and the IoT, as conducive to lightweight message processing and low communication overhead. CoAP can be transported over several transports, including UDP as the default option, and supports the use of transport intermediaries such as proxies. As for communication security, CoAP originally indicated the use of DTLS [3] for protecting exchanged messages at the transport layer.

Object Security for Constrained RESTful Environments (OSCORE) [4] is a lightweight security protocol that protects CoAP messages *end-to-end* at the application layer. OSCORE protects CoAP messages all the way from the original message producer to the final message consumer, also in the presence of intermediaries. OSCORE provides confidentiality, integrity, source authentication and replay protection of CoAP messages.

This work was partly funded by the European Commission and Stiftelsen för Strategisk Forskning.

To protect communications with OSCORE, two peers rely on a shared OSCORE Security Context, including parameters and keying material used for message protection. As it focuses on message protection and secure communication, OSCORE considers the establishment and update of the OSCORE Security Context to be addressed separately, while providing an optional method for key update in its specification [4].

However, the availability of an effective and efficient procedure to update the OSCORE keying material is vital in order to ensure long-term, secure communication. In particular, the current keying material eventually becomes invalid and has to be renewed. This is the case, for instance, when it reaches its expiration time, or when approaching well-known limits of the used cryptographic algorithms. Furthermore, in order to ensure good performance and address the constrained nature of most IoT devices and IoT-based network environments, it is of great importance that an update procedure displays limited complexity and communication overhead, and that it does not resort to a full-fledged re-establishment of the keying material.

This paper overviews and compares the following methods to perform key establishment and key update for OSCORE, with special focus on the novel KUDOS procedure [5].

- Keying material (re-)provisioning from a device operator.
- The authenticated, key establishment protocol Ephemeral Diffie-Hellman Over COSE (EDHOC) [6], under standardization in the IETF LAKE Working Group.
- The "OSCORE" [7] and "EDHOC and OSCORE" [8] profiles of the standard ACE framework for access control in the IoT [9], leveraging a trusted Authorization Server.
- The OSCORE-based device bootstrapping procedure provided by the standard management framework OMA Lightweight Machine-to-Machine (LwM2M) [10], leveraging a Bootstrap Server and a Device Manager Server.
- The peer-to-peer update procedure originally defined in the OSCORE specification [4], in its Appendix B.2.
- The peer-to-peer update procedure KUDOS [5], under standardization in the IETF CoRE Working Group.

We compare the different methods as to their functioning, security properties, and performance in terms of communication overhead and required message round-trips. Also, we provide a detailed, functional description of the novel key update procedure KUDOS, focusing on its rationale and design choices. Furthermore, we present a formal verification of KUDOS, using the tool Tamarin Prover [11]. The formal verification confirms that KUDOS satisfies its important security properties, including *keying material convergence* and *keying material confidentiality*, also in the presence of an adversary.

To the best of our knowledge, this is the first contribution providing a comprehensive overview of the key establishment and key update methods for the OSCORE protocol, and a formal verification of the novel key update procedure KUDOS.

The paper is organized as follows. Section II provides the background. Section III discusses what motivates key update. Section IV presents key update and key establishment methods for OSCORE. Section V provides a detailed and reasoned presentation of the novel key update procedure KUDOS. Section VI compares the different methods. Section VII presents our formal verification of KUDOS. Section VIII, presents the relevant related work. Section IX provides our conclusions.

II. Background

The following presents relevant technologies and concepts.

A. CoAP

The Constrained Application Protocol (CoAP) [2] is an application-layer, web-transfer protocol suitable for constrained devices and networks, especially in terms of lightweight message processing and low communication overhead. Like the widely used HTTP protocol, CoAP relies on the same REST paradigm, thus enabling a client peer to retrieve and manipulate the representation of resources at a server peer.

CoAP relies on a request/response message model with optional reliability, and is typically transported over UDP, but also supports other transports such as TCP. Furthermore, CoAP supports communication via intermediaries (e.g., proxies), caching of response messages, one-to-many group communication (e.g., over IP multicast), and translation to/from HTTP.

A CoAP message includes a header with mandatory fields (e.g., a request or response code, and a Message ID for message deduplication) and a Token field to enforce request-response correlation. The header can also include CoAP *options* that signal protocol features or extensions applied to the message. A notable example is the Observe option that allows a CoAP client to "subscribe" to a resource at a server, which will unsolicitedly send back a notification response when the resource representation changes. Finally, a CoAP message can include a payload conveying application data.

CoAP originally indicated the DTLS suite [3] for providing secure communication at the transport layer. Recently, the OSCORE security protocol has enabled end-to-end protection of CoAP messages at the application layer (see Section II-C).

B. CBOR and COSE

Concise Binary Object Representation (CBOR) [12] is a data format for compact, binary encoding. It is based on the JavaScript Object Notation (JSON) data model, but uses binary encoding for compactness, and is designed for small code size, extensibility, and lightweight encoding and decoding.

CBOR Object Signing and Encryption (COSE) [13] defines the creation and processing of representations for cryptographic keys, key exchange, signatures, message authentication codes, and ciphertexts. COSE is based on JSON Object Signing and Encryption, but it yields a more compact representation by using CBOR for serialization.

C. OSCORE

The standard Object Security for Constrained RESTful Environments (OSCORE) [4] is a security protocol providing *end-to-end* protection of CoAP messages at the application layer. Even in the presence of (untrusted) intermediaries, OSCORE protects a CoAP message all the way from the original message producer to the final message consumer. In particular, OSCORE ensures confidentiality, integrity, source authentication and replay protection of CoAP messages.

OSCORE relies on CBOR and COSE as its main building blocks, thus resulting in lightweight message processing and small communication overhead. Specifically, OSCORE takes a CoAP message as input, selectively protects different message parts by encrypting as much information as possible (while allowing intermediaries to read fields necessary for message forwarding), and produces as outcome a new, OSCORE-protected CoAP message. The latter includes the produced, authenticated ciphertext as payload, as well as an OSCORE CoAP option, which signals that the message is protected with OSCORE and enables the recipient to correctly decrypt and verify it, thus retrieving the original CoAP message.

Therefore, OSCORE can be used wherever CoAP works, irrespective of the underlying transport protocol. Also, OSCORE preserves features and extensions of CoAP (e.g., Observe) as well as the possible use of intermediaries, which are not required or expected to be OSCORE-aware.

To use OSCORE, two peers need a shared OSCORE Security Context, which can be established in many ways (see Section IV), e.g., through (re-)provisioning. However, OSCORE considers the establishment and update of the OSCORE Security Context to be addressed separately, while providing an optional method for key update in its specification [4].

The OSCORE Security Context of one peer specifies: the cryptographic algorithms; a Master Secret and Master Salt; the Sender ID (Recipient ID) of this peer corresponding to the Recipient ID (Sender ID) of the other peer; an optional ID Context to disambiguate different OSCORE Security Contexts; a Sender Sequence Number used for outgoing messages and incremented at each use; a Sender Key and a Recipient Key, which this peer derives from the material above, and uses to encrypt outgoing messages addressed to the other peer and to decrypt incoming messages from the other peer.

A client protects an outgoing request by using the OSCORE Security Context associated with the target resource at the server. A server protects an outgoing response by using the same OSCORE Security Context used for decrypting the corresponding request.

D. ACE framework

The standard ACE framework [9] enforces fine-grained access control in constrained, IoT environments. ACE builds on the widely used OAuth 2.0 [14], while using CoAP for message transport, and CBOR and COSE for compact encoding. The ACE workflow considers the three entities Client, Resource Server (RS), and Authorization Server (AS).

In particular, a Client that wants to access a protected resource at an RS must first request an Access Token from the AS in a trust relation with the RS. Consistent with the access policies to enforce, the AS can issue an Access Token to the Client, specifying the granted permissions. Then, the Client uploads the Access Token to the RS, after which the Client can accordingly access protected resources at the RS.

The ACE framework entrusts *profile* documents to specify how the parties securely communicate. The profiles especially define how a Client and an RS securely communicate with one another, and how they establish their secure association based on security material facilitated by the AS and specified in the Access Token. For example, existing ACE profiles indicates DTLS or OSCORE as the security protocol to use.

E. LwM2M

OMA Lightweight Machine-to-Machine (LwM2M) [10] is a framework for management of IoT devices. It defines how to perform provisioning of credentials, and how to transfer data between a managed and a managing device. It also defines several data models that devices can use to expose information for remote access. This can be, for example, the definition of which resources a temperature meter should make available.

LwM2M relies on defined roles including a Client (i.e., an IoT device to deploy and manage), a Bootstrap Server and a Management Server. The registration workflow consists of the Client securely connecting to the Bootstrap server (possibly using pre-configured security material), from which the Client receives the security material to use for securely connecting to the Management Server. In turn the Management server is provided with the same security material out-of-band.

Once a secure association is established, the Client registers at the Management Server, after which the two peers can securely exchange protected messages. LwM2M supports a number of protocols, including CoAP for message exchange, and DTLS and OSCORE for message protection.

F. Forward secrecy

Forward secrecy is a property of key management and key derivation protocols, where compromising long-term keys or current session keys does not compromise past session keys. For example, compromising a long-term, private authentication key used to establish a secure session does not compromise session keys previously established from the same private key.

Forward secrecy ensures that communications occurred in the past between two peers cannot later be revealed or tampered with due to a compromise of long-term keys. Without forward secrecy, an adversary can access the content of old, saved communications if long-term keys get compromised.

III. Motivation for Doing Key Update

Figure 1 shows a common scenario for deployment of devices communicating with OSCORE. That is, two peers are

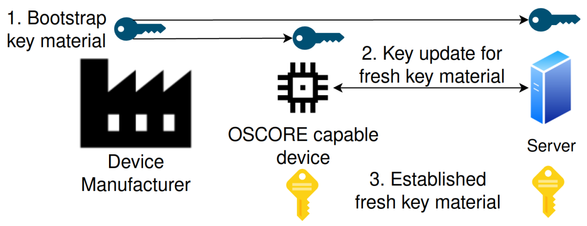

1. Bootstrap key material
2. Key update for fresh key material
Device Manufacturer
OSCORE capable device
Server
3. Established fresh key material

Fig. 1. Scenario with manual provisioning of keying material.

pre-provisioned by the device manufacturer with static keying material in read-only memory. To ensure secure communication, devices may not want to begin exchanging protected application data until they have performed a key update, as the current keying material is known to the manufacturer. Thus, the peers can first perform a secure key update to transition from the bootstrap keying material, and agree on new keying material to use. In this manner, the peers avoid communicating using the keying material initially provided to them.

More generally, two OSCORE peers may need to update their OSCORE keying material for different reasons. These include, for instance: enforcing local application policies that mandate periodic key update; or getting close to exhausting the OSCORE Sender Sequence Number space; or the expiration of the currently used OSCORE Security Context.

Another reason to perform key update is approaching limits for the number of times the keying material can be safely used. Cryptographic analysis has shown that, for Authenticated Encryption with Associated Data (AEAD) algorithms such as the different variants of AES, excessive key usage endangers certain security properties of the algorithms and/or reduces the security level of secure communication [15].

Since OSCORE uses AEAD algorithms for message integrity and confidentiality, it is critical to follow the recommended, specific limits. These concern the number of encryptions performed using the same Sender Key, and the number of failed decryptions using the same Recipient Key. Specifically, no more than 2^{20} encryptions or failed decryptions should occur with a specific key. If that happens, the peers should perform a key update or cease communication.

When using OSCORE, performing a key update is primarily about updating the OSCORE Master Secret and Master Salt, which in turn are used to derive new Sender and Recipient keys. After that, the peers share new keying material for which the maximum number of uses is safely available again, per the algorithm limits. Furthermore, since a new OSCORE Security Context is derived, the OSCORE Sender Sequence Number will be reset, hence the peers can continue communicating with the full sequence number space available again.

IV. Key Management Methods for OSCORE

Several methods exist for performing key establishment or key update (hereafter, "key management") for OSCORE.

A key management protocol should not only enable secure establishment or update of keying material, but also be lightweight in terms of computing and communication overhead. As CoAP and OSCORE are designed with the IoT and resource-constrained devices in mind, a key management protocol for OSCORE should follow this design philosophy, and ideally result in lightweight message processing, low communication overhead, and few round-trips.

The threat model considered for a key management protocol should take into account an on-path adversary that can capture, delete, or change any message exchanged between the peers running that protocol. This adversary can also replay previously captured messages and inject newly crafted ones. It is also assumed that the used cryptographic functions are secure, and that the adversary does not hold the keying material used for establishing the actual secure communication associations.

A. Manual (re-)provisioning from device operator

A basic method for key management consists in a device operator that hard-codes an OSCORE Security Context on both peers. This allows for both key establishment and key update. When the set of peers is well-known and not dynamically changing, this can be a feasible method.

However, it lacks flexibility and adds manual work for configuring the system. If the peers need to perform a key update regularly and do not support other key update methods, this method is not suitable as it entails another manual setup phase before communication can continue. This may be infeasible when IoT devices are deployed in hard to reach locations.

Thus, if a peer often needs to communicate with new network peers or exchanges a high number of messages, a manual re-provisioning of keying material is not suitable.

B. EDHOC Key establishment protocol

The EDHOC protocol [6], being defined within the LAKE IETF working group, is a lightweight key establishment protocol that can be used for deriving keys for OSCORE.

It is based on the SIGMA-I protocol and relies on an authenticated Diffie-Hellman key exchange using ephemeral keys. For peer authentication, it can use Message Authentication Codes or signatures. When used for OSCORE, an EDHOC execution yields an OSCORE Security Context with keying material and Sender/Recipient IDs for the two peers.

One benefit of EDHOC is the use of public peer authentication credentials (e.g., certificates) as starting point, from which symmetric keys are ultimately derived. EDHOC provides security properties such as forward secrecy, peer aliveness, mutual authentication, and secure algorithm negotiation.

EDHOC supports both key establishment and key update, although performing key establishment requires using asymmetric cryptographic operations, at least for Diffie-Hellman secret derivation. To perform key update, EDHOC defines the EDHOC-KeyUpdate function. However, how to practically use the EDHOC-KeyUpdate function in a synchronized way between two peers is not defined in the EDHOC specification.

C. OSCORE profile of the ACE framework

The OSCORE profile [7] of the ACE framework [9] (see Section II-D) can be used, with the ACE Client and RS acting as OSCORE peers. After the Client has obtained an Access Token from the AS, the Client uploads it to the RS, and at the same time exchanges two nonces with the RS. Then, both the Client and the RS use the two nonces and information specified in the Access Token as input for deriving the OSCORE Security Context. This OSCORE Security Context can be used by the Client and RS for subsequent communication.

To perform a key update, the Client can upload the Access Token again, while also exchanging two new nonces with the RS, and use them to derive new keying material. Thus, the ACE OSCORE profile can be used for both key establishment and key update. However, it requires the AS acting as trusted third party and it does not provide forward secrecy.

D. EDHOC and OSCORE profile of the ACE framework

The EDHOC and OSCORE profile [8] of the ACE framework is under development in the IETF ACE Working Group and can also be used, with the ACE Client and RS acting as EDHOC and OSCORE peers. After the Client has obtained an Access Token from the AS, the Client uploads it to the RS.

After that, the Client and the RS run EDHOC, using the peer authentication credentials facilitated by the AS. Based on the result of the EDHOC execution, the two peers derive an OSCORE Security Context. This method provides the full security properties of EDHOC, including forward secrecy, also when using the ACE framework. To perform a key update, the two peers can rely on the EDHOC-KeyUpdate function.

E. LwM2M bootstrapping procedure

When relying on OSCORE for secure communication, the OMA LwM2M framework [10] overviewed in II-E can also be used to perform key management. This occurs during the bootstrapping procedure, where the Bootstrap Server provides the LwM2M Client with the OSCORE Security Context to use with the Management Server. The Management Server will be provided out-of-band with the OSCORE Security Context to use for communicating with the Client.

LwM2M mandates the use of the key update procedure from Appendix B.2 of the OSCORE specification (see Section IV-F), for the Client and the Management Server to update their OSCORE Security Context (e.g., after a reboot).

Another way to update the keying material when using LwM2M is to simply perform the bootstrapping procedure again, assuming that the Bootstrap Server has updated OSCORE keying material to provide to the Client.

F. Appendix B.2 of the OSCORE specification

Appendix B.2 of the OSCORE specification [4] defines a method for performing key update.

The peers perform two exchanges of nonces, which are taken as input to derive a new OSCORE Security Context. Specifically, the two peers derive two temporary OSCORE Security Contexts for protecting the key update messages. Finally, they derive the new OSCORE Security Context to use for secure communication, by relying on the exchanged nonces for the key derivation. The different OSCORE Security Contexts also use different ID Contexts.

Since this method cannot perform key establishment, the peers must start from an already established OSCORE Security Context, which can be established through any method above.

G. *KUDOS*

KUDOS is a novel procedure for OSCORE key update, under development in the IETF CoRE Working Group. Compared to the method in Section IV-F, KUDOS is more lightweight, has better security properties, and allows to maintain CoAP observations beyond a key update.

Two peers executing KUDOS perform one exchange of nonces, which are then taken as input to derive a new OSCORE Security Context. KUDOS derives only a single temporary OSCORE Security Context during its execution, and does not change the used ID Context.

KUDOS is specifically designed to perform key update. If key establishment is desired, another method has to be used first. More details on KUDOS are presented in V.

V. KUDOS KEY UPDATE FOR OSCORE

KUDOS is a novel, efficient and lightweight method for key update for OSCORE. It displays a low communication overhead and completes the key update after only a single round trip, i.e., after only two KUDOS messages. When using KUDOS, the two peers exchange two nonces from which, together with the old keying material, they derive new keying material using one-way key derivation functions. KUDOS requires the existence of a shared OSCORE Security Context between the communicating peers before it can be initiated.

In order to derive a new OSCORE Security Context, KUDOS relies on the internal function *updateCtx()*. This takes as input the exchanged nonce values and the current, immediately previously derived OSCORE Security Context. The input material is further passed to the internal function KUDOS-Expand(), which generates a new OSCORE Master Secret. The new OSCORE Master Salt is simply the concatenation of the exchanged nonces. Finally, the new OSCORE Security Context is derived from this Master Secret and Master Salt.

KUDOS supports two modes of operation. In *stateful* mode, KUDOS achieves forward secrecy as it uses one-way key derivation functions, and takes as input to the key derivation the current, immediately previously derived keying material.

The *stateless* mode of operation is intended for constrained devices that are unable to dynamically store information to persistent memory. That is, they cannot permanently store the current, immediately previously derived keying material for retrieval after a loss of state, e.g., due to a device reboot. This means that, after every loss of state, a peer will use the same long-term, bootstrap keying material (e.g., as provided at manufacturing time) as the old keying material to update. Thus, the stateless mode does not provide forward secrecy.

KUDOS also displays the following properties: i) it can be initiated by either a client or server taking the role of "Initiator", with the other peer acting as "Responder"; ii) it is secure even if case a peer reboots and loses state information not stored in persistent memory; iii) only a single intermediate OSCORE Security context is derived throughout the KUDOS execution; iv) it does not change OSCORE identifiers; v) it allows preserving ongoing CoAP observations; vi) it is extensible to use algorithms defined for OSCORE in the future.

A. *Message Exchange*

The exchange of KUDOS messages is protected with OSCORE, thus inheriting the OSCORE security properties. Both the value and the length of the exchanged nonces are integrity protected by design, since they are input to the key derivation.

In order to enable a KUDOS execution, the OSCORE CoAP option has been extended as follows. The newly defined bit 'd' is set to indicate that an OSCORE message is a KUDOS message. Also, the newly defined 'x' byte specifies the length of the exchanged nonce value, and additional bits signaling how the present KUDOS execution should work (e.g., if in stateless or stateful mode, and if CoAP observations should be preserved). Finally, a new field specifies the nonce value conveyed in the present KUDOS message.

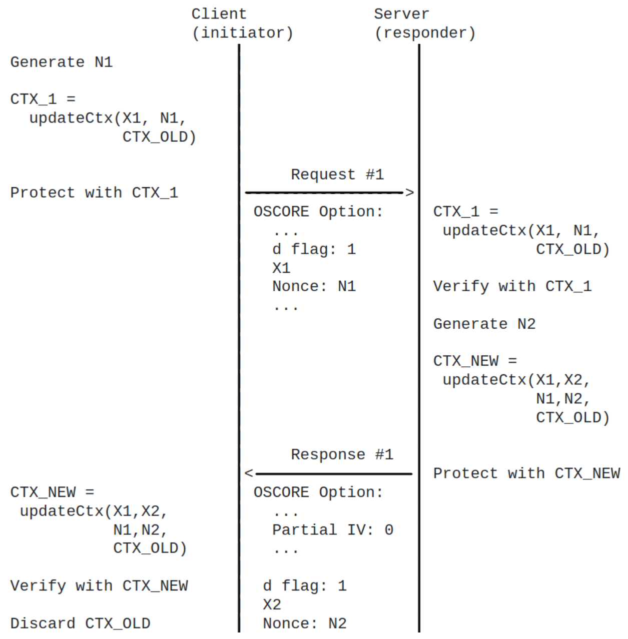


Fig. 2. KUDOS message flow.

We denote as CTX_OLD the OSCORE Security Context shared before the key update starts, and as Xi the content of the byte 'x' of the OSCORE option in the i-th KUDOS message. We focus on the client-initiated version of KUDOS as shown in Figure 2, which consists of the following steps.

First, the client prepares a CoAP Request #1 as first KUDOS message, and generates a nonce $N1$ and the value $X1$. Then, the client takes $N1$, $X1$ and CTX_OLD as input to *updateCtx()*, and derives a temporary OSCORE Security Context CTX_1. The client protects Request #1 using CTX_1, and specifies $N1$ and $X1$ in the OSCORE option. Finally, the client sends Request #1 to the server, and stores $N1$ and $X1$.

The server extracts $N1$ and $X1$ from Request #1, takes those and CTX_OLD as input to *updateCtx()*, and derives the temporary OSCORE Security Context CTX_1. After that, the server decrypts and verifies Request #1 using CTX_1.

The server prepares a CoAP Response #1 as second KUDOS message, and generates a nonce $N2$ and the value $X2$. Then, it takes $X1$, $X2$ $N1$, $N2$ and CTX_OLD as input to *updateCtx()*, and derives the OSCORE Security Context CTX_NEW. Next, it protects Response #1 using CTX_NEW, and specifies $N2$ and $X2$ in the OSCORE option. The server sends Response #1 to the client, and uses CTX_NEW for protecting its following communications with the client.

The client extracts $N2$ and $X2$ from Response #1, takes those and $N1$, $X1$, and CTX_OLD as input to *updateCtx()*, and derives the OSCORE Security Context CTX_NEW. Then, the client decrypts and verifies Response #1 using CTX_NEW.

Hereafter, the two peers communicate using the newly established OSCORE Security Context CTX_NEW.

B. Preserving CoAP observations

KUDOS allows retaining the ongoing CoAP observations that the two peers have with each other. To this end, the peers can set a specific bit of the 'x' byte in the KUDOS message, to indicate that they wish to preserve those observations beyond the key update. If both peers express that wish, then all those observations are preserved, otherwise they are all terminated.

Securely preserving observations is challenging, as it requires preventing an incoming OSCORE response from cryptographically matching multiple, different OSCORE requests.

This is due to the fact that, when a client derives a new OSCORE Security Context, it resets its Sender Sequence Number (SSN) to 0. If the client uses the same SSN value both in a request that started a CoAP observation and was protected with CTX_OLD, as well as in a request protected with CTX_NEW, then a response to either request would cryptographically match both requests.

KUDOS prevents this as follows. After having derived CTX_NEW and reset its SSN to 0, a peer determines SSN^* as the highest SSN value of previously sent requests associated with ongoing observations, for which this peer acts as client. Then, this peer updates its current SSN value, setting it to the value $SSN^* + 1$. By construction this prevents a later OSCORE response from the other peer cryptographically matching two OSCORE requests sent by this peer.

C. Stateless mode of operation

The stateful mode is the main mode of operation in KUDOS, as it retains forward secrecy for the peers' keying material. At the price of sacrificing forward secrecy, the stateless mode is intended for very constrained devices that cannot store information to persistent memory.

If a peer supports storing information to persistent memory, it must attempt running KUDOS in stateful mode. If the other peer also supports the stateful mode, then KUDOS will run to completion accordingly. Otherwise, KUDOS will not complete, and the peers can run KUDOS again using the stateless mode. If a peer does not support storing information to persistent memory or is aware that the other peer does not, then this peer runs KUDOS in stateless mode.

To indicate the wish to run KUDOS in stateless mode, a peer sets a specific bit of the 'x' byte in its outgoing KUDOS message. KUDOS can correctly complete if both exchanged KUDOS messages indicate either the use of the stateful mode (if both peers can use it), or the use of the stateless mode.

Since the mode to use is indicated in the integrity-protected 'x' byte, it is practically infeasible to perform a downgrade attack against a KUDOS execution. Therefore, it is ensured that the security properties of the newly derived OSCORE Security Context CTX_NEW are as high as both peers can afford, given their capabilities. That is, forward secrecy is preserved unless it is fundamentally not possible to do otherwise.

VI. Comparison of Key Management Methods

This section provides a discussion and detailed comparison of the key management methods for OSCORE presented in Section IV. We compare the methods with respect to the following aspects: usage of asymmetric cryptography; support, approach and required maximum round-trips for key establishment (KE) and key update (KU); preservation of CoAP observations and OSCORE identifiers in case of key update. Usage of asymmetric cryptography and maximum round-trips are specifically considered as they have a performance impact.

In the following, support for "key update" refers to the ability to perform a lightweight renewing of the current keying material, without undergoing a new, full-fledged key establishment. Table I provides a comparison at-a-glance.

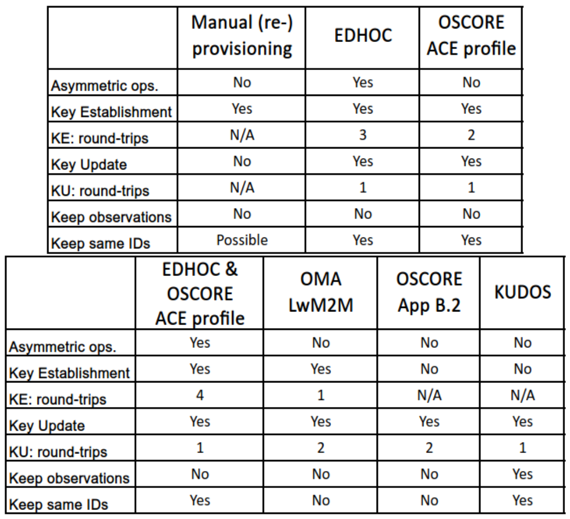
Manual (re-)provisioning of keying material on a device can be used for key establishment, without in-band communication. This procedure being radical and invasive, CoAP observations cannot be preserved. Preserving or changing the OSCORE identifiers is at the discretion of the device operator.

EDHOC can do key establishment and key update, and uses asymmetric cryptographic operations at least for the Diffie-Hellman secret derivation. An EDHOC key establishment consists of exchanging at least 3 messages, followed by a fourth optional message or an application message for key confirmation. Hence, EDHOC requires 2 round trips before the new OSCORE Security Context can be used. Key update is possible by using the EDHOC-KeyUpdate function, but the EDHOC specification does not define exactly how two peers should use it. Preserving observations is not possible.

The OSCORE profile of ACE can do key establishment and key update. Key establishment takes 2 round trips: 1 for the Client to retrieve an Access Token from the AS, and 1 for uploading the Access Token to the RS together with the exchange of nonces. Key update takes 1 round trip, for re-uploading the Access Token and exchanging the nonces to derive a new OSCORE Security Context, while preserving the OSCORE identifiers. Preserving observations is not possible.

The EDHOC and OSCORE profile of ACE can do key establishment and key update. It uses asymmetric cryptographic operations at least for the Diffie-Hellman secret derivation. Key establishment takes 4 round trips: 1 for the Client to

TABLE I
COMPARISON OF KEY MANAGEMENT METHODS.



	Manual (re-) provisioning	EDHOC	OSCORE ACE profile	EDHOC & OSCORE ACE profile	OMA LwM2M	OSCORE App B.2	KUDOS
Asymmetric ops.	No	Yes	No	Yes	No	No	No
Key Establishment	Yes	Yes	Yes	Yes	Yes	No	No
KE: round-trips	N/A	3	2	4	1	N/A	N/A
Key Update	No	Yes	Yes	Yes	Yes	Yes	Yes
KU: round-trips	N/A	1	1	1	2	2	1
Keep observations	No	No	No	No	No	No	Yes
Keep same IDs	Possible	Yes	Yes	Yes	No	No	Yes

retrieve an Access Token from the AS, 1 for uploading the Access Token to the RS, and 2 to run EDHOC with the RS. Key update takes 1 round trip, for re-uploading the Access Token and exchanging the nonces to derive a new OSCORE Security Context, while preserving the OSCORE identifiers. Preserving observations is not possible.

The LwM2M framework can do key establishment and key update. Key establishment takes 3 round trips between the Client and the Bootstrap server, of which 2 are for performing the procedure defined in Appendix B.2 of the OSCORE specification, and 1 is for the Client to retrieve the keying material to use with the Management Server. For performing key update, the Client can perform the procedure defined in Appendix B.2 of the OSCORE specification directly with the Management Server, which takes 2 round trips. Preserving observations and OSCORE identifiers is not possible.

The procedure in Appendix B.2 of the OSCORE specification only does key update. It takes 2 round trips for the OSCORE peers to exchange the necessary nonces, while it creates two temporary OSCORE Security Contexts. Preserving observations and OSCORE identifiers is not possible.

KUDOS only does key update, which takes only 1 round trip for the OSCORE peers to exchange the necessary nonces, while creating only one temporary OSCORE Security Context. Preserving observations is possible at the discretion of the two peers, and OSCORE identifiers are preserved.

VII. FORMAL VERIFICATION OF KUDOS

We performed a formal verification of KUDOS using the tool Tamarin Prover [11]. As a symbolic modeling tool, neither concrete values nor any computation is involved in the verification. Instead, it relies solely on abstract relationships between parameters, building its formal proofs as output.

In our verification, we considered the Dolev-Yao adversary model used by default in Tamarin. That is, the adversary has full control of the network and can stop, delay, modify and inject messages, but cannot break cryptographic functions. This is consistent with the threat model in Section IV.

To perform our verification, we produced a Tamarin model of the KUDOS procedure[1]. The model includes the transition rules between states, and lemmas defining the security properties. In addition, we have designed and implemented a custom equation that represents the OSCORE message integrity protection mechanism. This is similar to the built-in signing mechanism of Tamarin, but uses a symmetric key instead of an asymmetric key pair, i.e.: `check(m, prot(m,k), k) = true`, where `m` is the protected message, and `k` is the key used to protect it.

The one-way function `f(secret, nonces)` models the derivation of a new OSCORE Security Context in KUDOS. The one-way function `g(secret, salt, ID, label)` models the derivation of the OSCORE Sender and Recipient Key. As Tamarin works in the symbolic model, these functions do not perform concrete operations, but merely represent the relationship between the input parameters and the output keys.

The Tamarin model considers the two roles ”Initiator” and ”Responder”, and abstracts away from the two peers being CoAP client or server, as not relevant for proving security properties. Also, the model consists of 4 rules: 1 for initializing the system, and 3 for the protocol execution (see Section V).

We defined the following two lemmas related to our model. The *secrecy* lemma, if proved, ensures the confidentiality of the newly derived keying material (*keying material confidentiality*). That is, even in the presence of an on-path active adversary, only the peers executing the KUDOS procedure can derive the new keying material.

```
lemma secrecy: "All s #i.
Secret(s) @i ==> not Ex #j. K(s) @j"
```

The *authenticity* lemma, if proved, ensures that, if the Initiator properly concludes the protocol (i.e., ”Commit”), then the Responder did in fact perform a corresponding execution on its side (i.e., ”Running”), and thus both sides agree on the relevant data for that execution. This, in turn, ensures absolute convergence on the same new keying material now shared between Initiator and Responder (*keying material convergence*).

```
lemma noninjective_agreement:
"All I R t #i. Commit(I,R,t) @i
==> (Ex #j. Running(R,I,t) @j)"
```

The built-in solver managed to automatically prove both lemmas in a few minutes, using the default heuristic. Thus, the corresponding properties are guaranteed to hold in all possible executions of the KUDOS procedure in the model.

VIII. RELATED WORK

The TLS suite [16] provides a Handshake protocol to establish keying material for protecting communication at the transport layer. Authentication of the server peer acting as responder is mandatory, while authentication of the client peer is not. Most authentication methods provide forward secrecy.

[1]Tamarin model code: https://github.com/rikard-sics/kudos-tamarin-model

A peer can locally update the current TLS traffic secrets, derive its new sending keys from those, and notify the other peer about this update by sending a KeyUpdate Handshake message. The other peer updates the TLS traffic secrets and its receiving keys accordingly, and may follow-up in the same way and send a KeyUpdate Handshake message. That is, this key update method is uni-directional, i.e., keys can be updated in a single direction, if desired. After deriving new sending keys, a peer must use those to protect its outgoing messages.

IKEv2 [17] allows the establishment of keying material for protecting communication at the network layer with IPsec [18].

Specifically, IKEv2 allows two peers to establish and update Security Associations (SAs). The peers first exchange two messages to establish an IKE SA, which is used to protect further IKEv2 communications. Next, the peers exchange two further, secured IKEv2 messages and authenticate each other by means of either pre-shared symmetric keys or certificates.

As a result, the peers create a Child SA used to protect subsequent traffic with IPsec. This can provide either integrity-protection of the entire IP packet, and/or encryption, authentication and integrity-protection of the IP packet payload.

IKEv2 optionally supports key update, which the peers perform by exchanging two messages including nonces, as protected with the current IKE SA. To perform key update of the IKE SA, the peers derive a Diffie-Hellman secret to take as input to key derivation. A newly derived IKE SA replaces the current one, and inherits the current one's Child SAs. Instead, to perform key update of a Child SA, the peers optionally derive a Diffie-Hellman secret taken as input to key derivation. A newly derived Child SA replaces the current one.

IX. Conclusion

We have presented an overview and a detailed comparison of different methods to perform key establishment and key update for the standard security protocol OSCORE, which provides end-to-end message protection for the standard, web-transfer protocol CoAP especially intended for IoT devices.

The methods are: i) manual (re-)provisioning of keying material; ii) the key establishment protocol EDHOC; iii) the ACE framework for access control, through its "OSCORE" and "EDHOC and OSCORE" profiles; iv) the OSCORE-based device bootstrapping of the framework LwM2M; v) the key update procedure from Appendix B.2 of the OSCORE specification; and v) the key update procedure KUDOS.

We have also provided an extensive functional description of KUDOS, including its rationale and design choices. Compared to the procedure from Appendix B.2 of the OSCORE specification as its most similar contender, KUDOS displays better security properties and performance, while requiring fewer message exchanges. KUDOS does not change OSCORE identifiers, allows preserving CoAP observations, and can be used also by devices that cannot write to persistent memory.

Finally, we have performed a formal verification of KUDOS using the Tamarin Prover. Our proof confirms that KUDOS satisfies its claimed security properties, including *keying material convergence* and *keying material confidentiality*, also in the presence of an adversary targeting a KUDOS execution.

Future work will experimentally evaluate and compare the performance of different key update methods for OSCORE when running on resource-constrained IoT devices, with particular reference to the procedure from Appendix B.2 of the OSCORE specification and the novel KUDOS procedure.

Acknowledgment

This work was partly supported by the H2020 project SIFIS-Home (Grant agreement 952652), the SSF project SEC4Factory (Grant agreement RIT17-0032), and the H2020 project ARCADIAN-IoT (Grant agreement 101020259).

References

[1] IDC, "Iot growth demands rethink of long-term storage strategies, says idc," https://iotbusinessnews.com/2020/07/29/20898-iot-growth-demands-rethink-of-long-term-storage-strategies-says-idc/, Jul. 2020.

[2] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC 7252 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–112, Jun. 2014, updated by RFC 7959.

[3] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," RFC 6347 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–32, Jan. 2012, updated by RFCs 7507, 7905.

[4] G. Selander, J. P. Mattsson, F. Palombini and L. Seitz, "Object Security for Constrained RESTful Environments (OSCORE)," RFC 8613, Jul. 2019.

[5] R. Höglund and M. Tiloca, "Key Update for OSCORE (KUDOS)," Internet Engineering Task Force, Internet-Draft draft-ietf-core-oscore-key-update, 2022.

[6] G. Selander, J. P. Mattsson and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)," Internet Engineering Task Force, Internet-Draft draft-ietf-lake-edhoc, 2023.

[7] F. Palombini, L. Seitz, G. Selander, M. Gunnarsson, "The Object Security for Constrained RESTful Environments (OSCORE) Profile of the Authentication and Authorization for Constrained Environments (ACE) Framework," RFC 9203, Aug. 2022.

[8] G. Selander , J. P. Mattsson, M. Tiloca , R. Höglund, "Ephemeral Diffie-Hellman Over COSE (EDHOC) and Object Security for Constrained Environments (OSCORE) Profile for Authentication and Authorization for Constrained Environments (ACE)," Internet Engineering Task Force, Internet-Draft draft-ietf-ace-edhoc-oscore-profile, 2022.

[9] L. Seitz , G. Selander, E. Wahlstroem, S. Erdtman , H. Tschofenig, "Authentication and Authorization for Constrained Environments Using the OAuth 2.0 Framework (ACE-OAuth)," RFC 9200, Aug. 2022.

[10] Open Mobile Alliance, "Lightweight machine to machine technical specification: Core - 1.2," http://www.openmobilealliance.org, Nov. 2020.

[11] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The tamarin prover for the symbolic analysis of security protocols," Springer, pp. 696–701, 2013.

[12] C. Bormann and P. E. Hoffman, "Concise Binary Object Representation (CBOR)," RFC 8949, Dec. 2020.

[13] J. Schaad, "CBOR Object Signing and Encryption (COSE): Structures and Process," RFC 9052, Aug. 2022.

[14] D. Hardt, "The OAuth 2.0 Authorization Framework," RFC 6749 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–76, Oct. 2012, updated by RFC 8252.

[15] F. Günther, M. Thomson, C. A. Wood, "Usage Limits on AEAD Algorithms," Internet Engineering Task Force, Internet-Draft draft-irtf-cfrg-aead-limits, 2023.

[16] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018.

[17] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen, "Internet Key Exchange Protocol Version 2 (IKEv2)," RFC 5996 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–138, Sep. 2010, obsoleted by RFC 7296, updated by RFCs 5998, 6989.

[18] S. Kent and K. Seo, "Security Architecture for the Internet Protocol," RFC 4301 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–101, Dec. 2005, updated by RFCs 6040, 7619.