More Lightweight, yet Stronger: Revisiting OSCORE's Replay Protection

Konrad-Felix Krentz* and Thiemo Voigt*†
*Uppsala University, Department of Electrical Engineering
†RISE Computer Science
Email: {konrad.krentz,thiemo.voigt}@angstrom.uu.se

***Abstract*—Object Security for Constrained RESTful Environments (OSCORE) is an end-to-end security solution for the Constrained Application Protocol (CoAP), which, in turn, is a lightweight application layer protocol for the Internet of things (IoT). The recently standardized Echo option allows OSCORE servers to check if a request was created recently. Previously, OSCORE only offered a counter-based replay protection, which is why delayed OSCORE requests were accepted as fresh. However, the Echo-based replay protection entails an additional round trip, thereby prolonging delays, increasing communication overhead, and deteriorating reliability. Moreover, OSCORE remains vulnerable to a denial-of-sleep attack. In this paper, we propose a version of OSCORE with a revised replay protection, namely OSCORE next-generation (OSCORE-NG). OSCORE-NG fixes OSCORE's denial-of-sleep vulnerability and provides freshness guarantees that surpass those of the Echo-based replay protection, while dispensing with an additional round trip. Furthermore, in long-running sessions, OSCORE-NG incurs even less communication overhead than OSCORE's counter-based replay protection. OSCORE-NG's approach is to entangle timestamps in nonces. Except during synchronization, CoAP nodes truncate these timestamps in outgoing OSCORE-NG messages. Receivers fail to restore a timestamp if and only if an OSCORE-NG message is delayed by more than 7.848s in our implementation by default. In effect, older OSCORE-NG messages get rejected.**

I. Introduction

In the Internet of things (IoT), traffic often goes via middleboxes, such as brokers or proxies. This ensues a fragmentation of the end-to-end security since, e.g., Datagram Transport Layer Security (DTLS) or IPsec connections terminate at middleboxes. As a remedy, Object Security for Constrained RESTful Environments (OSCORE) implements end-to-end security at the application layer, thereby providing "true" end-to-end security, i.e., across middleboxes [18].

OSCORE builds upon the Constrained Application Protocol (CoAP), which is a lightweight application layer protocol for the IoT [19]. A CoAP message comprises a header, zero or more options, and payload. An OSCORE message is also a CoAP message with (i) the masked header of an unprotected CoAP message, (ii) unencrypted options of the unprotected CoAP message, (iii) an inserted OSCORE option, and (iv) a payload comprising encrypted parts of the unprotected CoAP message, as well as a message integrity code (MIC). The OSCORE option imparts data to receivers that they require for restoring the unprotected CoAP message, as well as for checking its freshness and authenticity.

This work was supported by the Swedish Foundation for Strategic Research.

Workshop on Security and Privacy in Standardized IoT (SDIoTSec) 2024
26 February 2024, San Diego, CA, USA
ISBN 979-8-9894372-6-9
https://dx.doi.org/10.14722/sdiotsec.2024.23003
www.ndss-symposium.org

Until recently, OSCORE only offered a counter-based replay protection, which works as follows. Clients insert an incrementing sequence number into the OSCORE option of each OSCORE request. When retransmitting an OSCORE request, the originally assigned sequence number must be reused. Servers can hence detect replayed OSCORE requests, but cannot distinguish between retransmitted and replayed OSCORE requests. Therefore, upon receiving an apparently replayed OSCORE request, servers have to send a cached OSCORE response. Caching is dispensable in the case of so-called idempotent OSCORE requests [3]. These are OSCORE requests that can be processed multiple times without side effects. In OSCORE responses, OSCORE options usually elide sequence numbers. Despite being suppressed, the sequence number of the corresponding OSCORE request becomes part of the nonce for the agreed upon Authenticated Encryption with Associated Data (AEAD) algorithm. A client looks up an elided sequence number via the 'Token' field of the CoAP header, which must match that of the corresponding OSCORE request. If a client finds an OSCORE response authentic, the client deletes the looked up sequence number. Thus, a lookup only succeeds once and replayed OSCORE responses get discarded as a result.

OSCORE's counter-based replay protection has two vulnerabilities.

1) First, the fact that servers cannot remain silent upon receiving replayed OSCORE requests constitutes a denial-of-sleep vulnerability. This is because it enables attackers to cause an increased energy consumption by replaying OSCORE requests. A non-compliant fix is to set new sequence numbers in retransmitted OSCORE requests [11]. However, this fix is inapplicable to non-idempotent OSCORE requests. For supporting non-idempotent OSCORE requests, it is key to recognize retransmissions. Actually, the 'Message ID' field of the CoAP header is intended for this purpose, but it is left unauthenticated by OSCORE and hence malleable. Authenticating it may conflict with CoAP proxies as they may modify message IDs to coordinate concurrent requests toward the same destination endpoint.

2) Second, servers cannot detect if an OSCORE request was delayed. This can, e.g., become critical when interacting with a door lock [14]. Specifically, an attacker may delay a command to open the door until the owner moves out of sight. Even if the door lock receives other OSCORE requests in the meantime, a delay attack may still succeed as OSCORE maintains a replay window to allow for out-of-order delivery. A defense is to use the newly introduced Echo option, which enables servers to ensure the temporal freshness of an OSCORE request [2]. To do so, servers reply with an Echo option enclosed instead of processing an OSCORE request. A client then has to resend its OSCORE request with a copy of the received Echo option. Ultimately, servers can confine the age of an OSCORE request to the age of its Echo option. That said, the Echo-based replay protection harms delays, communication overhead, and reliability due to the added round trip.

In this paper, we propose, analyze, and evaluate OSCORE next-generation (OSCORE-NG), a version of OSCORE with a revised replay protection. OSCORE-NG has three benefits:

- First, OSCORE-NG resists denial-of-sleep attacks, while providing support for non-idempotent requests.
- Second, OSCORE-NG detects shorter delays than the Echo-based replay protection and not only protects requests against delay attacks, but also responses.
- Third, OSCORE-NG works without an additional round trip, thereby avoiding detrimental effects on delays, communication overhead, and reliability. In long-running sessions, OSCORE-NG has an even lower communication overhead than OSCORE's counter-based replay protection. This is despite of learning and maintaining clock differences since OSCORE-NG realizes this by piggybacking synchronization data on messages that are being sent anyway.

II. Related Work

Counter-based replay protection is ill-suited for the IoT. For one, its communication overhead is undesirable in IoT scenarios since low bit rates and energy constraints are typical there. To reduce its communication overhead, Gouda et al. proposed the last bits (LB) optimization, which only conveys lower order bits of counters and restores higher order bits via anti-replay data [5], [13]. Yet, the LB optimization suffers from desynchronization and may cost random-access memory (RAM) [8], [10]. Even without the LB optimization, RAM consumption can become problematic for constrained IoT devices as the number of communication partners increases. To this end, Luk et al. suggested using Bloom filters [13], but they incur a high energy consumption [7]. After all, a fundamental issue with counter-based replay protection is that it only provides sequential freshness, i.e., merely ensures that a message is not accepted more than once [16]. In short-running sessions with high packet rates, sequential freshness often suffices as old messages quickly fall out of the anti-replay window. However, IoT devices receive rather rarely and may, e.g., connect to a cloud backend for long. Hence, strong freshness is desirable. It allows a receiver to not only detect duplicate messages, but also if a message was sent within a limited time span prior to its reception [16].

There exist three ways to achieve strong freshness. First, one can use wall clock timestamps, but they create various complications [14]. For instance, after correcting a host's time, the host may accept old or reject fresh messages. Moreover, wall clock timestamps may reveal information, such as that a host accepts expired certificates. Lastly, attacks on time synchronization also impact replay protection. Second, the side that requires strong freshness can generate a random nonce and ask the communication partner to echo that nonce [4]. That approach is actually now standardized for OSCORE [2]. Such Echo-based replay protection compromises on delays, communication overhead, and reliability. Third, special ways to achieve strong freshness emerged in Layer 2 security. There, the general idea is to reuse Layer 2 synchronization efforts for replay protection. In the time-slotted channel hopping (TSCH) medium access control (MAC) protocol, for example, transmissions happen in timeslots whose indices are known to sender and receiver [1]. By deriving nonces from these timeslot indices, an authentic frame can also be considered fresh, provided that session keys are in use. Similarly, in the coordinated sampled listening (CSL) MAC protocol, transmissions are scheduled around wake ups of the receiver [1]. By assigning incrementing counters to each wake up and using these wake-up counters for generating nonces, strong freshness can be achieved with little initial communication overhead [8]–[10]. Analogous to wall clock timestamps, however, vulnerabilities in the Layer 2 synchronization undermine replay protection. In CSL, such attack paths are excluded through the adoption of the proven Secure Pairwise Synchronization (SPS) protocol [4], [20]. OSCORE-NG tailors the third way to Layer 7 security and also adopts the SPS protocol. More specifically, OSCORE-NG adopts Sun et al.'s version of the SPS protocol [20]. Their version has a reduced communication overhead compared to the original one.

Preuß Mattsson et al. pointed out the possibility of delay attacks against OSCORE-, DTLS-, TLS-, and IPsec-protected CoAP traffic [14]. Additionally, they explained so-called mismatch attacks against DTLS-, TLS-, and IPsec-protected CoAP traffic. A mismatch attack tricks a client into accepting a CoAP response that was originally sent in reply to a different CoAP request. Besides, Preuß Mattsson et al. go into vulnerabilities that are specific to CoAP's block-wise transfers. All vulnerabilities found by Preuß Mattsson et al. were fixed in RFC 9175 [2]. Unfortunately, its defense against mismatch attacks entails communication overhead, whereas OSCORE and OSCORE-NG thwart mismatch attacks by themselves. This further motivates the use of OSCORE or OSCORE-NG instead of DTLS, TLS, or IPsec for securing CoAP traffic.

Krentz et al. discussed a remote denial-of-sleep attack against OSCORE, where a remote attacker injects or replays OSCORE messages [11]. Basically, since OSCORE only filters out inauthentic and replayed OSCORE messages at the destination host, injected and replayed OSCORE messages incur an increased energy consumption. Hence, they put forward a middlebox that filters out unwanted OSCORE messages in a trusted execution environment (TEE). Furthermore, their tiny remote attestation protocol (TRAP) establishes trust in the TEE, as well as sets up an OSCORE session for communica-

TABLE I. Time Values of OSCORE-NG and CoAP with Default Values of our Implementation and CoAP, Respectively

Name	Default	Description
`TIMESTAMP_UNIT`	0.01s	Precision of timestamps
`TIMESLOT`	10.24s	Duration of a timeslot
`PHASE_UNIT`	0.16s	Precision of the phase field
`MAX_DRIFT`	$30\frac{\mu s}{s}$	Maximum divergence of clocks
`SYNC_INTERVAL`	6h	Resynchronization interval
`ACK_TIMEOUT`	2s	Waiting period before the first retransmission
`MAX_TRANSMIT_SPAN`	45s	Maximum time span between the first transmission and the last retransmission
`PROCESSING_DELAY`	2s	Maximum turnaround time

tion with the TEE. One feature of their middlebox is the prevention of remote denial-of-sleep attacks, including the denial-of-sleep attack we described in the introduction. However, they adopted the non-compliant fix of setting new sequence numbers in retransmitted OSCORE requests, thus limiting their solution to idempotent OSCORE requests. OSCORE-NG overcomes this limitation. Another feature of their middlebox is that it frees IoT devices of setting up and maintaining individual sessions for communication with remote hosts. Instead, IoT devices just set up and maintain a single OSCORE session for communicating with the TEE. In this manner, the resource requirements of OSCORE are much reduced.

III. OSCORE-NG

This section first outlines the threat model and approach of OSCORE-NG. Next, it details the format of OSCORE-NG options, OSCORE-NG's realization of SPS, and OSCORE-NG's adaptations to the replay protection-related processing rules of OSCORE. Finally, this section gives various correctness and security proofs. For a summary of important configuration constants of OSCORE-NG and CoAP, see Table I.

A. Threat Model

OSCORE-NG assumes an attacker who can sniff, drop, replay, inject, modify, and delay any OSCORE-NG message exchanged between a client and a server. Yet, the attacker neither controls the communicating endpoints themselves nor any TEE attested by one of the communicating endpoints. Further, preventing the attacker from tampering with unencrypted options is out of scope. Also, OSCORE-NG's threat model excludes traffic analysis attacks. Finally, OSCORE-NG's threat model includes denial-of-sleep attacks, yet leaves it up to key establishment schemes to protect themselves against other relevant denial-of-service attacks, such as amplification attacks.

Altogether, there are two differences between OSCORE-NG's and OSCORE's threat model [18]. First, OSCORE-NG's threat model is concerned about delay attacks against all OSCORE-NG messages. OSCORE's threat model, by contrast, considers delay attacks against specific requests benign and settles for accepting a delayed response as long as its echoed token is active. Second, OSCORE-NG's threat model augments OSCORE's threat model by denial-of-sleep attacks.

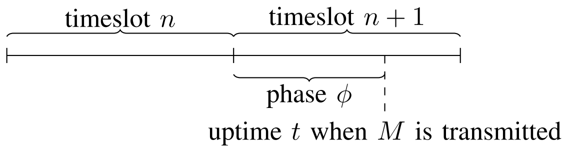


Fig. 1. Relation of transmission time, timeslot, and phase

B. Overview

OSCORE-NG uses SPS to learn and maintain clock differences. Except when running SPS, OSCORE-NG options just carry a truncated transmission timestamp. The portion that is elided in transit is the so-called *timeslot*. Let t be the uptime of a sender at the time of transmitting an OSCORE-NG message M, as shown in Fig. 1. The timeslot n at time t is determined by taking a configurable number of the higher order bits of t. The number of retained bits determines the duration `TIMESLOT` of a timeslot. For example, our implementation defaults to `TIMESLOT` $= 10.24$s. *Phase* ϕ, on the other hand, is the time span between the beginning of the transmission timeslot and t, i.e., the truncated bits of t. Both timeslot and phase become part of the AEAD nonce that is used for securing M.

Receivers restore elided transmission timeslots based on clock differences, which they learn through SPS and subsequently maintain by re-running SPS on occasion. Crucially, if an attacker delays an OSCORE-NG message by longer than a configurable delay, the restoration of the transmission timeslot will fail. In such a case, a wrong AEAD nonce is restored, causing the delayed OSCORE-NG messages to be considered inauthentic. As a result, OSCORE-NG attains strong freshness. To put things into perspective, our OSCORE-NG implementation rejects a message if it was delayed by more than 7.848s by default, whereas OSCORE's counter-based replay protection provides no such guarantees and the Echo-based replay protection leaves OSCORE requests valid for at least 51s, presuming a default configuration of CoAP [19].

To attain sequential freshness, OSCORE-NG mainly relies on end-to-end message IDs and includes them in AEAD nonces, too. If a CoAP proxy needs to modify the 'Message ID' CoAP header field for disambiguation, it moves the original message ID to the OSCORE-NG option.

Similarly, to prevent mismatch attacks, OSCORE-NG authenticates original tokens and offers the possibility for CoAP proxies to extend the 'Token' CoAP header field. Often, CoAP proxies can avoid to do so as active tokens are distinct already.

C. Option Format

The format of OSCORE-NG options is shown in Fig. 2. After various flags, the 'phase' field contains the phase in units of `PHASE_UNIT` $\overset{\text{def}}{=} \frac{\texttt{TIMESLOT}}{2^4}$ seconds, e.g., 0.16s in our implementation by default. During synchronization and resynchronization, the 'phase' field is replaced with timestamps. When initiating a synchronization or resynchronization, a client or server only fills out the 'tx timestamp' field. Its value comprises the transmission timeslot and phase, where phase is measured in units of `TIMESTAMP_UNIT` seconds, e.g., 0.01s in our implementation. When replying, both other timestamp fields are also present. The 'corresponding tx timestamp' field

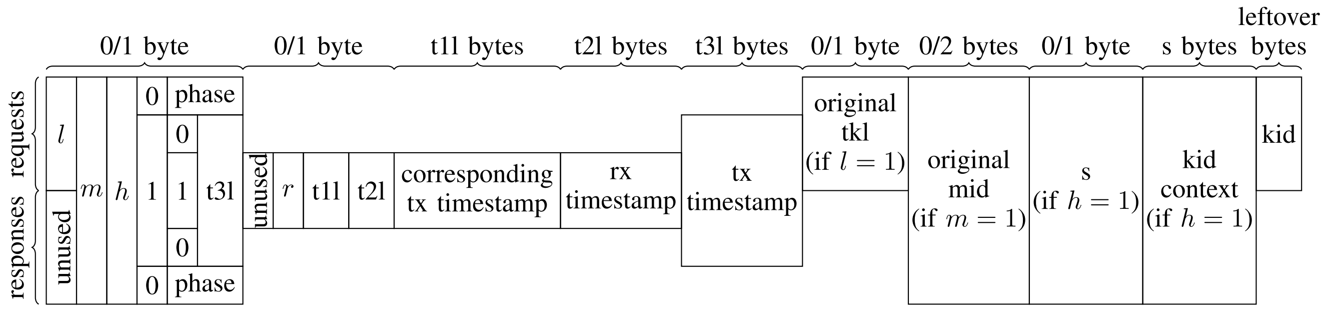
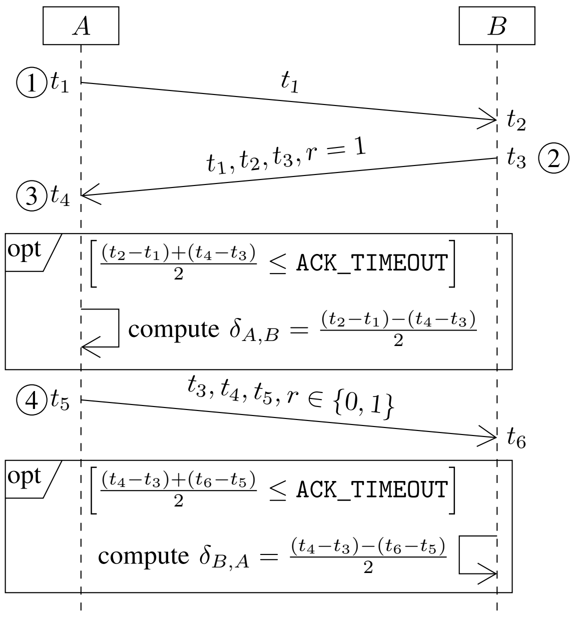


Fig. 2. Format of OSCORE-NG options. OSCORE-NG retains OSCORE's 's', 'kid context', and 'kid' fields. The new 'phase' field allows receivers to restore transmission timestamps, which are entangled in AEAD nonces. During synchronization and resynchronization, up to three timestamps replace the 'phase' field. The 'original tkl' field carries the length of the original token. The 'original mid' field serves for deduplication and for generating AEAD nonces.



A
B
1 t_1
t_1
t_2
$t_1, t_2, t_3, r = 1$
t_3 2
3 t_4
opt
$\left[\frac{(t_2-t_1)+(t_4-t_3)}{2} \leq \texttt{ACK\_TIMEOUT}\right]$
compute $\delta_{A,B} = \frac{(t_2-t_1)-(t_4-t_3)}{2}$
4 t_5
$t_3, t_4, t_5, r \in \{0, 1\}$
t_6
opt
$\left[\frac{(t_4-t_3)+(t_6-t_5)}{2} \leq \texttt{ACK\_TIMEOUT}\right]$
compute $\delta_{B,A} = \frac{(t_4-t_3)-(t_6-t_5)}{2}$

Fig. 3. For learning and maintaining clock differences, OSCORE-NG adopts Sun et al.'s lightweight variant of the SPS protocol [20]

echoes the received 'tx timestamp'. The 'rx timestamp' field contains the timeslot and phase when having received the echoed 'tx timestamp'. Again, phase is measured in units of TIMESTAMP_UNIT seconds here. Thereafter, OSCORE-NG requests carry the length of the original token and the original message ID in the 'original tkl' and 'original mid' field, respectively. These fields are dispensable when the 'Token' and 'Message ID' CoAP header fields are relayed unmodified. Lastly, OSCORE-NG retains the 's', 'kid context', and 'kid' fields of OSCORE options. While the 'kid' field carries the client's ID, the 's'-byte 'kid context' field can provide inputs for the generation of session keys and session constants.

D. Secure Pairwise Synchronization

Fig. 3 shows the operation of SPS in OSCORE-NG. At ①, A initiates the protocol by populating the 'tx timestamp' field in an OSCORE-NG message to B. That message can be a request or a response. At ②, B creates a response or request, in which the 'corresponding tx timestamp', 'rx timestamp', and 'tx timestamp' fields, contain t_1, t_2, and t_3,

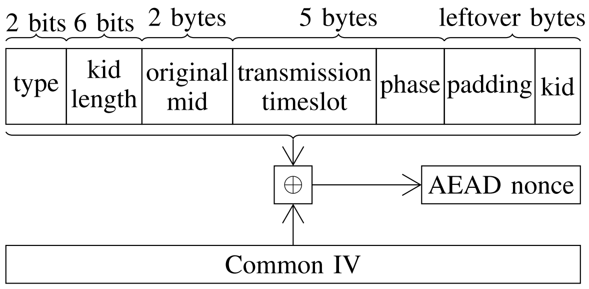


Fig. 4. OSCORE-NG derives an AEAD nonce from the CoAP message type, transmission timestamp, original message ID, and client ID, also called 'kid'

respectively. B also sets the r flag, which asks A to return fully populated timestamp fields at the next occasion. At ③, A first checks if $\frac{(t_2-t_1)+(t_4-t_3)}{2} \leq \texttt{ACK\_TIMEOUT}$. This check confines the repercussions of delay attacks. Next, A computes its clock difference $\delta_{A,B}$ compared to B. At ④, A adds t_3, t_4, and t_5 to the next OSCORE-NG message to B. The r flag remains unset if $\frac{(t_2-t_1)+(t_4-t_3)}{2} \leq \texttt{ACK\_TIMEOUT}$, i.e., if A was able to initialize $\delta_{A,B}$. Upon reception, B analogously checks the round-trip time and computes its clock difference $\delta_{B,A}$ compared to A. For the security of SPS, it is necessary to authenticate all timestamps. Hence, OSCORE-NG expands the authenticated contents to include all timestamp fields.

Resynchronizations must happen early enough so that the restoration works correctly throughout a session. As we will show shortly, a resynchronization must happen before:

$$\texttt{SYNC\_INTERVAL} \leq \frac{\frac{\texttt{TIMESLOT}}{2} - 2\texttt{ACK\_TIMEOUT} - \frac{\texttt{PHASE\_UNIT}}{2}}{\texttt{MAX\_DRIFT}} \tag{1}$$

, where MAX_DRIFT is the maximum divergence of clocks per second in seconds and ACK_TIMEOUT is when a retransmission happens at the earliest. Only configurations with $\texttt{SYNC\_INTERVAL} > 0$ are reasonable.

E. Replay Protection

1) Protecting a CoAP Request: OSCORE-NG forms the AEAD nonce for protecting a CoAP request in two steps, as shown in Fig. 4. First, OSCORE-NG concatenates the CoAP message type, the client's ID, the original message ID, the

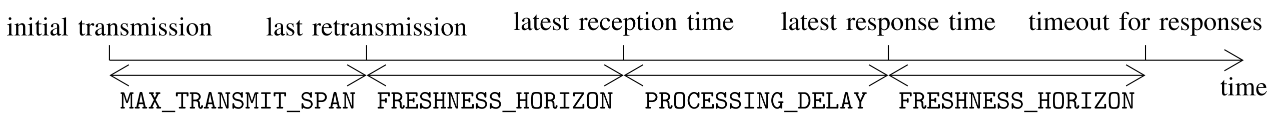


Fig. 5. Message IDs may only repeat after a backoff period so that servers do not falsely detect a request as a duplicate. Additionally, to prevent mismatch attacks, tokens and message IDs must not be reused before previous responses have become inauthentic due to reaching an age of `FRESHNESS_HORIZON`.

transmission timeslot, and phase. Second, like in OSCORE, the result is XORed with the so-called Common IV. The Common IV is a secret session constant, thus making AEAD nonces unpredictable for attackers.

Crucially, OSCORE-NG constraints when a retransmission can happen at the earliest and on when a message ID can be reused. OSCORE-NG requires that $\texttt{ACK\_TIMEOUT} \geq \texttt{PHASE\_UNIT}$ and that message IDs do not repeat before the timeout for responses, as shown in Fig. 5. Therein, `MAX_TRANSMIT_SPAN` is the maximum time between an initial OSCORE-NG request and its last retransmission, `PROCESSING_DELAY` is the maximum turnaround time, and `FRESHNESS_HORIZON` is defined as follows:

$$\texttt{FRESHNESS\_HORIZON} \overset{\text{def}}{=} \frac{\texttt{TIMESLOT}}{2} + \frac{\texttt{PHASE\_UNIT}}{2} + \texttt{ACK\_TIMEOUT} + \texttt{MAX\_DRIFT} \times \texttt{SYNC\_INTERVAL} \quad (2)$$

It shall turn out that a receiver will find an OSCORE-NG message inauthentic if it is delayed by more than `FRESHNESS_HORIZON`. Thus, for servers it suffices to keep anti-replay data for $\texttt{MAX\_TRANSMIT\_SPAN} + \texttt{FRESHNESS\_HORIZON}$.

After the authenticated encryption, OSCORE-NG inserts a corresponding OSCORE-NG option. Normally, the ‘phase’ field is present and imparts the phase of the AEAD nonce. When synchronizing or resynchronizing, the ‘tx timestamp’ field appears instead and conveys the timeslot and phase entangled in the AEAD nonce. Other timestamp fields may also be included as required for SPS. The ‘original tkl’ and ‘original mid’ fields are elided. CoAP proxies add them when adapting the ‘Token’ or ‘Message ID’ CoAP header field.

2) Verifying an OSCORE-NG Request: On reception of an OSCORE-NG request at time t, a first step is to generate the AEAD nonce. As for an OSCORE-NG request with a suppressed transmission timeslot, the server uses its learned clock difference δ compared to the client to reconstruct the transmission timeslot of the AEAD nonce as follows. Let ϕ denote the product of the value of the ‘phase’ field and `PHASE_UNIT`. The server rounds $t + \delta - \phi$ to the nearest beginning of a transmission timeslot and takes that timeslot as the transmission timeslot. As for an OSCORE-NG request with timestamps, there are two cases. First, if δ is not yet in place, the server derives timeslot and phase from the ‘tx timestamp’ field. Second, if a server is aware of its clock difference δ compared to the client, the server estimates the transmission timeslot using only the phase portion of the ‘tx timestamp’ field. This is crucial to attain strong freshness for all OSCORE-NG requests in a session (but the very first) that only have a ‘tx timestamp’ field. Note that if other timestamps are present, SPS already ensures strong freshness, but poses stricter requirements. By suppressing the received tx timestamp and restoring it via the latest δ, OSCORE-NG detects if an OSCORE-NG request is fresh enough for upper layers, while it might not be fresh enough for updating δ.

With the AEAD nonce in place, the server decrypts and checks the AEAD MIC. If the OSCORE-NG request turns out authentic, the server moves on to check if the original message ID reoccurred within $\texttt{MAX\_TRANSMIT\_SPAN} + \texttt{FRESHNESS\_HORIZON}$. Additionally, if less than two OSCORE-NG messages were accepted in the current session, the server checks if the original message ID matches that of the first OSCORE-NG request in the session. If the original message ID reoccurred together with a previously seen transmission timeslot and phase, the server suspects a replay attack and silently discards the OSCORE-NG request. Otherwise, if the original message ID reoccurred together with a previously unseen transmission timeslot or phase, the server suspects a retransmission and either re-protects a cached unprotected CoAP response or, in the case of an idempotent OSCORE-NG request, may alternatively process the request again. Unprotected CoAP responses to a non-idempotent OSCORE-NG request have to be cached for $\texttt{MAX\_TRANSMIT\_SPAN} + \texttt{FRESHNESS\_HORIZON}$ and until a second OSCORE-NG message has been accepted. Finally, if all checks pass, the server accepts the OSCORE-NG request.

3) Protecting a CoAP Response: The AEAD nonce of an OSCORE-NG response is formed in the same manner as that of OSCORE-NG requests. To prevent a nonce reuse, the combination of CoAP message type, message ID, and client ID may not repeat within `PHASE_UNIT`. However, if all other AEAD inputs are also equal, such a nonce reuse will be benign as this yields the same ciphertext. Thus, an implementation may alternatively ensure that all AEAD inputs are equal. In particular, it is safe to re-protect a cached unprotected CoAP response. Besides, to allow for deduplication, the combination of CoAP message type, message ID, and client ID may not repeat within $\texttt{MAX\_TRANSMIT\_SPAN} + 2\texttt{FRESHNESS\_HORIZON} + \texttt{PROCESSING\_DELAY}$, unless when retransmitting a response.

In the OSCORE-NG option of an OSCORE-NG response, the ‘original tkl’ field remains omitted on the whole reverse path. This is because the original token needs to be restored on the last leg anyway, except in Empty Acknowledgements (ACKs) whose ‘Token’ field remains empty on the whole reverse path, obviating the ‘original tkl’ field, too. Likewise, the original message ID must be restored on the last leg if a response’s CoAP message type is ACK or Reset (RST). Else, if a response is of type Non-confirmable (NON), a CoAP proxy may need to adapt the ‘Message ID’ CoAP header field for mapping an RST reply to a NON response. Finally, either SPS-related fields or the ‘phase’ field are filled as appropriate.

4) Verifying an OSCORE-NG Response: The receiver of an OSCORE-NG response extracts the original message ID from the CoAP header or OSCORE-NG option.

Subsequently, the restoration of the transmission timeslot works like in the case of OSCORE-NG requests. If an OSCORE-NG response turns out authentic, the receiver proceeds with checking if the combination of message ID and type reoccurred within $\texttt{MAX\_TRANSMIT\_SPAN} + 2\texttt{FRESHNESS\_HORIZON} + \texttt{PROCESSING\_DELAY}$, i.e., within the time frame where authentic responses can come in, as shown in Fig. 5. If so, the OSCORE-NG response is silently discarded. Else, the receiver accepts the OSCORE-NG response.

F. Correctness Properties

1) Restoration of Transmission Timeslots: OSCORE-NG may measure clock differences that deviate significantly from the real ones for two reasons. First, there is a high variation in the end-to-end delays when communicating with energy-constrained IoT devices because of their sleep cycles. Second, delay attacks may already happen during synchronization. However, inaccuracies do not negatively affect OSCORE-NG as long as the restoration of the elided transmission timeslots works. Recall that this restoration works by rounding $t + \delta - \phi$ to the closest transmission timeslot, where t is the time of reception, δ denotes the latest measured clock difference, and ϕ is the received phase. Let t^* denote the real transmission time, δ^* denote the real clock difference, and ϕ^* denote the unrounded value of ϕ. Below, we show that if the communication delay is below $\texttt{ACK\_TIMEOUT}$, i.e., $|t^* - (t + \delta^*)| < \texttt{ACK\_TIMEOUT}$, the restoration of the transmission timeslot works, i.e., $|(t^* - \phi^*) - (t + \delta - \phi)| < \frac{\texttt{TIMESLOT}}{2}$.

We can confine ϕ^* like follows:

$$\phi^* \in \left[\phi - \frac{\texttt{PHASE\_UNIT}}{2}, \phi + \frac{\texttt{PHASE\_UNIT}}{2}\right) \quad (3)$$

$$\Rightarrow |\phi^* - \phi| \leq \frac{\texttt{PHASE\_UNIT}}{2} \quad (4)$$

Furthermore, at the time of synchronization, SPS ensures:

$$\begin{aligned}
&|\delta^* - \delta| \\
=&\left|\delta^* - \frac{(t_2 - t_1) - (t_4 - t_3)}{2}\right| \\
=&\left|\frac{2\delta^*}{2} - \underbrace{\frac{t_2 - t_1}{2}}_{\leq \texttt{ACK\_TIMEOUT} - \frac{t_4 - t_3}{2}} + \underbrace{\frac{t_4 - t_3}{2}}_{\leq \texttt{ACK\_TIMEOUT} - \frac{t_2 - t_1}{2}}\right| \\
=&\left|- \underbrace{\frac{t_2 - t_1 - \delta^*}{2}}_{\in [0, \texttt{ACK\_TIMEOUT} - \frac{t_4 - t_3 + \delta^*}{2}]} + \underbrace{\frac{t_4 - t_3 + \delta^*}{2}}_{\in [0, \texttt{ACK\_TIMEOUT} - \frac{t_2 - t_1 - \delta^*}{2}]}\right| \\
\leq&\texttt{ACK\_TIMEOUT}
\end{aligned} \quad (5)$$

Δ seconds after the synchronization, we have:

$$|\delta^* - \delta| \leq \texttt{ACK\_TIMEOUT} + \texttt{MAX\_DRIFT} \times \Delta \quad (6)$$

With Equation 1, 4, and 6 we get:

$$\begin{aligned}
&|(t^* - \phi^*) - (t + \delta - \phi)| \\
=&|(t^* - \phi^*) - (t + \delta - \phi) + \delta^* - \delta^*| \\
=&|(t^* - (t + \delta^*)) + (\phi - \phi^*) + (\delta^* - \delta)| \\
<&\texttt{ACK\_TIMEOUT} + \frac{\texttt{PHASE\_UNIT}}{2} \\
&+ \texttt{ACK\_TIMEOUT} + \texttt{MAX\_DRIFT} \times \Delta \\
<&2\texttt{ACK\_TIMEOUT} + \frac{\texttt{PHASE\_UNIT}}{2} \\
&+ \texttt{MAX\_DRIFT} \times \texttt{SYNC\_INTERVAL} \\
\leq&\frac{\texttt{TIMESLOT}}{2}
\end{aligned} \quad (7)$$

2) Uniqueness of Nonces: Since OSCORE-NG uses special AEAD nonces, it is in order to assure that no such nonce reoccurs together with the same key. Like OSCORE, OSCORE-NG derives separate session keys for the sender and receiver side, which already excludes nonce reuses when OSCORE-NG messages travel in opposite directions. Thus, it suffices to ensure that no nonce reuse occurs among all outgoing OSCORE-NG messages. Observe that the general format of AEAD nonces is equal and always contains the CoAP message type, the original message ID, the transmission timeslot, the 4-bit phase, and the client's ID. Since requests use different client IDs than responses, this disambiguates the AEAD nonces of requests and responses. Thus, it remains to separately ensure for requests and responses that the concatenation of the other values is unique. As for requests, because original message IDs may not repeat within $\texttt{PHASE\_UNIT}$, at least phase will change before an original message ID gets reused. Likewise, retransmissions only happen after $\texttt{ACK\_TIMEOUT} \geq \texttt{PHASE\_UNIT}$, which is why at least phase differs in retransmissions. As for responses, the combination of a CoAP message type and an original message ID may only occur once within $\texttt{PHASE\_UNIT}$. In lieu of that, an implementation ensures to reuse all other AEAD inputs, too.

G. Security Properties

In the following, we show that, except for the very first OSCORE-NG request in a session, any OSCORE-NG message delayed by longer than $\texttt{FRESHNESS\_HORIZON}$ automatically gets rejected. Next, we show that no OSCORE-NG message is processed more than once, unless it is a retransmitted idempotent OSCORE-NG request. Lastly, we discuss how OSCORE-NG thwarts mismatch attacks.

1) Strong Freshness: Suppose an attacker delays an OSCORE-NG message other than the very first request in a session by $d > \texttt{FRESHNESS\_HORIZON}$. Below, we show that the receiver rejects that delayed OSCORE-NG message. If a measured clock difference δ is not in place, SPS directly checks if $d \leq 2\texttt{ACK\_TIMEOUT} < \texttt{FRESHNESS\_HORIZON}$. Otherwise, a receiver restores a transmission timeslot by rounding $t + \delta - \phi$ to the closest transmission timeslot, where t is the time of reception and ϕ is the received phase. Let t^* denote the real transmission time, δ^* denote the real clock difference, and ϕ^* denote the unrounded value of ϕ. With Equations (4) and (6):

$$\begin{aligned}
&|(t^* - \phi^*) - (t + \delta - \phi)| \\
=&|(t^* - \phi^*) - (t + \delta - \phi) + \delta^* - \delta^*| \\
=&|(t^* - (t + \delta^*)) + (\phi - \phi^*) + (\delta^* - \delta)| \\
=&|d + (\phi - \phi^*) + (\delta^* - \delta)| \\
\geq&|d - \frac{\texttt{PHASE\_UNIT}}{2} - (\texttt{ACK\_TIMEOUT} \\
&+ \texttt{MAX\_DRIFT} \times \texttt{SYNC\_INTERVAL})| \\
>&\frac{\texttt{TIMESLOT}}{2}
\end{aligned} \tag{8}$$

Thus, the receiver restores a different transmission timeslot than the sender used for generating the AEAD nonce and eventually find the delayed OSCORE-NG message inauthentic.

2) Sequential Freshness: Suppose a server receives an authentic non-idempotent OSCORE-NG request in a session at time t. We show that the OSCORE-NG request will not be processed more than once. There are two main cases. First, consider that the server accepted two or more OSCORE-NG messages in the session beforehand. Then, we can upper bound the age of the authentic OSCORE-NG request to FRESHNESS_HORIZON at time t. Let us first look at the subcase that the OSCORE-NG request is a legitimate retransmission. If there was already an authentic OSCORE-NG request with the same original message ID within $[t - \texttt{MAX\_TRANSMIT\_SPAN} - \texttt{FRESHNESS\_HORIZON}, t]$, the OSCORE-NG request is detected as a duplicate and a cached response is sent, thereby not processing the retransmitted OSCORE-NG request again. Else, the retransmitted OSCORE-NG request cannot be a duplicate since the time between an initial transmission and the latest time when an OSCORE-NG request is authentic is $\texttt{MAX\_TRANSMIT\_SPAN} + \texttt{FRESHNESS\_HORIZON}$, as shown in Fig. 5. Next, let us look at the subcase that the OSCORE-NG request was replayed. If the replayed OSCORE-NG request was already received within $[t - \texttt{MAX\_TRANSMIT\_SPAN} - \texttt{FRESHNESS\_HORIZON}, t]$, it would use the same transmission timeslot and phase. Therefore, OSCORE-NG would reject it. Else, the replayed OSCORE-NG request reduces to a retransmitted OSCORE-NG request like in the first subcase. Second, consider that the server accepted less than two OSCORE-NG messages in the session. Since OSCORE-NG keeps anti-replay data at least until accepting a second OSCORE-NG message, retransmissions of the first OSCORE-NG request in a session are answered from the cache if non-idempotent, and replayed ones are discarded.

As for authentic idempotent OSCORE-NG requests, replayed and retransmitted ones are being detected in an analogous manner. Departing from non-idempotent ones, however, RAM-optimized implementations process retransmitted idempotent OSCORE-NG requests multiple times safely.

Next, assume a client receives an authentic OSCORE-NG response. We argue that the OSCORE-NG response will be ignored if that response was received already. This follows from the fact that within the time frame where the combination of CoAP message type and message ID uniquely identifies an OSCORE-NG response, OSCORE-NG maintains anti-replay data to filter out responses with that same combination.

3) Prevention of Mismatch Attacks: To prevent mismatch attacks, OSCORE-NG authenticates the ‘Token’ CoAP header field, which clients use to map responses to requests. This approach also covers so-called notifications. A notification is a CoAP response that conveys a new version of a resource to an observing client [6]. The ‘Token’ CoAP header field of a notification has to echo the ‘Token’ CoAP header field of the initial registration message. This allows clients to identify to which registration an incoming notification corresponds to. Despite of OSCORE-NG’s defense against mismatch attacks, however, applications remain in charge of not reusing tokens while same ones are still active.

IV. Implementation

We integrated OSCORE-NG into libcoap (https://github.com/kkrentz/libcoap), as well as into Krentz et al.’s middlebox (https://github.com/kkrentz/filtering-proxy). Our libcoap implementation runs on Linux hosts, CC2538-based IoT devices [21], as well as in the Cooja network simulator [15]. Our middlebox implementation runs in a Keystone TEE [12].

Both implementations share the same core implementation of OSCORE-NG. To ensure that this core implementation performs consistently with our analytical results, we implemented regression tests. For example, we ensure that if an OSCORE-NG request is delayed by 8s, it will turn out inauthentic.

For OSCORE-NG communication within an IoT network, our libcoap implementation offers to establish session keys as per the protocol specified in Appendix B.2 of OSCORE [18], henceforth called B2 protocol. The B2 protocol derives session keys from a pre-shared Master Secret and a pre-shared Master Salt. It involves four messages, namely Request #1, Response #1, Request #2, and Response #2. All four messages are being secured with OSCORE-NG already. Hence, the client and server learn their clock difference when receiving Response #1 and Request #2, respectively. It is no problem that OSCORE-NG does not provide strong freshness for Request #1 as the B2 protocol does not pass its contents to the upper layer, but first requires the client to confirm a nonce in Request #2.

For OSCORE-NG communication between an IoT device and a remote host, we integrated OSCORE-NG in Krentz et al.’s middlebox. Their middlebox uses the TRAP remote attestation protocol, which is based on Fully Hashed Menezes-Qu-Vanstone with Confirmation (FHMQV-C) [17]. TRAP involves three request-response rounds, namely a `/kno` request and response (short for knock), a `/reg` request and response (short for register), and a `/dis` request and response (short for disclose). The initial `/kno` request-response round, as well as the `/reg` request protect against amplification attacks like in DTLS. Besides, the `/reg` request contains an ephemeral public key of the initiating IoT device. Upon reception, the Keystone security monitor (SM) generates an attestation report, an own ephemeral key pair, as well as FHMQV-C secrets for the TEE. Subsequently, the TEE generates a Master Secret for use in OSCORE-NG, as well as a Layer 2 key, which serves for waking up the initiating IoT device in the future as is essential for the the remote denial-of-sleep protection. The `/reg` response is sent as an unprotected CoAP message and contains the attestation report in a compressed format. This attestation report allows the IoT device to establish trust in the TEE and to generate the same Master Secret and Layer 2 key. The subsequent `/dis` request is the first OSCORE-NG-protected CoAP message in the session between the initiating IoT device and the TEE. Again, it is acceptable that OSCORE-NG does not provide strong freshness until a second OSCORE-NG message gets accepted because stale sessions get deleted anyway. After running TRAP, the middlebox relays OSCORE-NG messages between the IoT device and remote hosts.

V. Evaluation

In this section, we argue that OSCORE-NG offers stronger freshness guarantees than the Echo-based replay protection, while incurring less communication overhead than OSCORE's counter-based replay protection in long-running sessions.

A. Comparison to the Echo-based Replay Protection

As for the freshness guarantees of the Echo-based replay protection, note that an Echo option must at least remain valid for

$$\begin{aligned} &\texttt{ACK\_TIMEOUT} + \texttt{PROCESSING\_DELAY} \\ &+\texttt{MAX\_TRANSMIT\_SPAN} + \texttt{ACK\_TIMEOUT} \end{aligned} \tag{9}$$

, where $\texttt{ACK\_TIMEOUT}+$ accounts for the delay from server to client, `PROCESSING_DELAY` for the turnaround time, `MAX_TRANSMIT_SPAN` for potential retransmissions, and $+\texttt{ACK\_TIMEOUT}$ for the delay from client to server. With the default values in Table I, this yields a minimum validity period of 51s. OSCORE-NG, on the other hand, detects delays greater than $\texttt{FRESHNESS_HORIZON} = 7.848$s by default. Unlike the Echo-based replay protection, OSCORE-NG even ensures this for responses, which is very useful when OSCORE-NG responses are notifications. `FRESHNESS_HORIZON` improves when lowering `SYNC_INTERVAL` at the cost of resynchronizing more often. Tightening `ACK_TIMEOUT` also strengthens freshness guarantees, which is why lower layers should be tuned accordingly.

As for communication overhead, the Echo-based replay protection requires sending each request twice, unless a long validity period is chosen, which would further deteriorate the freshness guarantees. Another response is also conveyed. Plus, all these messages are to be secured with OSCORE, causing further overhead. OSCORE-NG, by contrast, avoids an additional round trip, which is beneficial in terms of communication overhead, reliability, and delays.

B. Comparison to the Counter-based Replay Protection

OSCORE's communication overhead can be separated into that caused by OSCORE options, AEADs MICs, the rearrangement of options, as well as key management. Equally, OSCORE-NG requires adding an OSCORE-NG option and an AEAD MIC to an unprotected CoAP message, rearranging the options of the unprotected CoAP message, and key management activities. OSCORE-NG does not normally require scheduling extra messages for resynchronization since keep-alive messages are sent more often than `SYNC_INTERVAL` in practice. Hence, we shall focus on comparing the overhead of OSCORE and OSCORE-NG options. To this end, a Cooja simulation was run for 60 virtual days, in which a client sent 6 CoAP requests to a server per virtual hour. In a

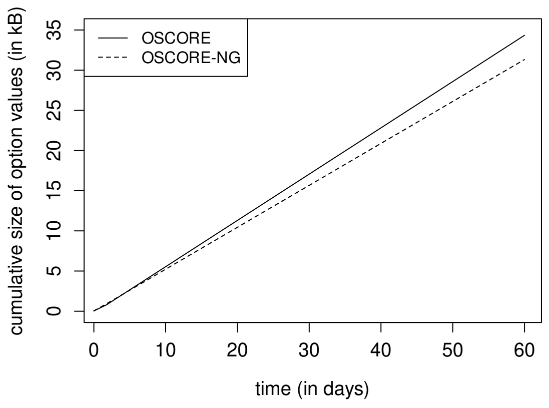


Fig. 6. The communication overhead of OSCORE-NG is even lower than that of OSCORE's counter-based replay protection in long-running sessions

first experimental run, all CoAP messages were secured with libcoap's OSCORE implementaion. In a second experimental run, the CoAP messages were secured with OSCORE-NG. In both runs, the length of the OSCORE(-NG) options was logged.

Fig. 6 shows the results. Initially, the difference in communication overhead is marginal. As soon as OSCORE's sequence number spans two bytes instead of only one, the communication overhead of OSCORE options begins to exceed that of OSCORE-NG options. Similarly, OSCORE-NG's timestamp fields increase with uptime. However, the need for transmitting timestamps arises seldomly with $\texttt{SYNC_INTERVAL} = 6$h.

VI. Conclusion and Future Work

Delay attacks mislead OSCORE into accepting old requests. This vulnerability poses a severe threat to energy-constrained IoT devices because such devices receive rarely and hence old requests only slowly fall out of OSCORE's anti-replay window. As a countermeasure, RFC 9175 standardized the Echo-based replay protection. Unfortunately, its freshness guarantees are quite weak and the added round trip harms delay, communication overhead, as well as reliability. After all, this adjunct leaves OSCORE vulnerable to denial-of-sleep attacks, unless adopting the non-compliant fix of incrementing sequence numbers in retransmissions, which is however inapplicable to non-idempotent requests. OSCORE-NG has overcome all these limitations. As for delay attacks, OSCORE-NG entangles implicit timestamps in AEAD nonces. Presumably, this kind of replay protection has only been applied to Layer 2 security, yet. As for denial-of-sleep attacks, short-term anti-replay data allows receivers to distinguish between replayed and retransmitted OSCORE-NG messages. Thus, receivers remain silent under replay attacks, respond to legitimate retransmissions, and do not process non-idempotent messages more than once. A notable property of OSCORE-NG is that any improvement in delays pays off twice, once in terms of strengthened freshness guarantees and once in terms of shorter cache occupation. Therefore, we aim to shorten the delays of lower layers in our future work.

References

[1] "IEEE Standard 802.15.4-2020," 2020.

[2] C. Amsüss, J. P. Mattsson, and G. Selander, "Constrained Application Protocol (CoAP): Echo, Request-Tag, and Token Processing," RFC 9175, 2023.

[3] C. Amsüss, "OSCORE Implementation Guidance," IETF, Tech. Rep. draft-amsuess-lwig-oscore-00, 2020.

[4] S. Ganeriwal, C. Pöpper, S. Capkun, and M. B. Srivastava, "Secure time synchronization in sensor networks," *ACM Transactions on Information and System Security (TISSEC)*, vol. 11, no. 4, pp. 23:1–23:35, 2008.

[5] M. G. Gouda, Y. ri Choi, and A. Arora, *Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks*. Auerbach, 2005, ch. Antireplay Protocols for Sensor Networks, pp. 561–574.

[6] K. Hartke, "Observing Resources in the Constrained Application Protocol (CoAP)," RFC 7641, 2015.

[7] D. C. Jinwala, D. R. Patel, S. J. Patel, and K. S. Dasgupta, "Optimizing the replay protection at the link layer security framework in wireless sensor networks," *CoRR*, vol. abs/1203.4694, 2012. [Online]. Available: http://arxiv.org/abs/1203.4694

[8] K.-F. Krentz, "A denial-of-sleep-resilient medium access control layer for IEEE 802.15.4 networks," Ph.D. dissertation, Potsdam University, 2019.

[9] K.-F. Krentz and Ch. Meinel, "Denial-of-sleep defenses for IEEE 802.15.4 coordinated sampled listening (CSL)," *Computer Networks*, vol. 148, no. 15, pp. 60–71, 2019.

[10] K.-F. Krentz, Ch. Meinel, and H. Graupner, "More lightweight, yet stronger 802.15.4 security through an intra-layer optimization," in *Proceedings of FPS 2017*. Springer, 2017, pp. 173–188.

[11] K.-F. Krentz and T. Voigt, "Reducing trust assumptions with OSCORE, RISC-V, and Layer 2 one-time passwords," in *Proc. of the 15th International Symposium on Foundations and Practice of Security (FPS 2023)*. Springer, 2023, p. 389–405.

[12] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proc. of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. ACM, 2020.

[13] M. Luk, G. Mezzour, A. Perrig, and V. Gligor, "MiniSec: A secure sensor network communication architecture," in *Proc. of the 6th International Conference on Information Processing in Sensor Networks (IPSN '07)*. ACM, 2007, p. 479–488.

[14] J. P. Mattsson, J. Fornehed, G. Selander, and C. Amsüss, "Attacks on the Constrained Application Protocol (CoAP)," draft-ietf-core-attacks-on-coap-03, 2023.

[15] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with cooja," in *Proc. of the 2006 31st IEEE Conference on Local Computer Networks (LCN 2006)*. IEEE, 2006, pp. 641–648.

[16] D. R. Raymond, R. C. Marchany, and S. F. Midkiff, "Scalable, cluster-based anti-replay protection for wireless sensor networks," in *Proc. of the 2007 IEEE SMC Information Assurance and Security Workshop (IAW 2007)*. IEEE, 2007, pp. 127–134.

[17] A. P. Sarr, P. Elbaz-Vincent, and J.-C. Bajard, "A secure and efficient authenticated diffie–hellman protocol," in *Proc. of the European Public Key Infrastructure Workshop (EuroPKI 2009)*. Springer, 2010, pp. 83–98.

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

[19] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC 7252, 2014.

[20] K. Sun, P. Ning, and C. Wang, "TinySeRSync: Secure and resilient time synchronization in wireless sensor networks," in *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. ACM, 2006, p. 264–277.

[21] *CC2538 SoC for 2.4-GHz IEEE 802.15.4 & ZigBee/ZigBee IP Applications User's Guide (Rev. C)*, Texas Instruments, http://www.ti.com/lit/ug/swru319c/swru319c.pdf.