CoRE Working Group
Internet-Draft
Updates: 8613, 8768 (if approved)
Intended status: Standards Track
Expires: 4 September 2025

M. Tiloca
R. Höglund
RISE AB

3 March 2025

OSCORE-capable Proxies
draft-ietf-core-oscore-capable-proxies-04

Abstract

Object Security for Constrained RESTful Environments (OSCORE) can be used to protect CoAP messages end-to-end between two endpoints at the application layer, also in the presence of intermediaries such as proxies. This document defines how to use OSCORE for protecting CoAP messages also between an origin application endpoint and an intermediary, or between two intermediaries. Also, it defines rules to escalate the protection of a CoAP option, in order to encrypt and integrity-protect it whenever possible. Finally, it defines how to secure a CoAP message by applying multiple, nested OSCORE protections, e.g., both end-to-end between origin application endpoints, and between an application endpoint and an intermediary or between two intermediaries. Therefore, this document updates RFC 8613. Furthermore, this document updates RFC 8768, by explicitly defining the processing with OSCORE for the CoAP option Hop-Limit. The approach defined in this document can be seamlessly used with Group OSCORE, for protecting CoAP messages when group communication is used in the presence of intermediaries.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the Constrained RESTful Environments Working Group mailing list (core@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/core/.

Source for this draft and an issue tracker can be found at https://github.com/core-wg/oscore-capable-proxies.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

This Internet-Draft will expire on 4 September 2025.

Copyright Notice

Table of Contents

1. Introduction

The Constrained Application Protocol (CoAP) [RFC7252] supports the presence of intermediaries, such as forward-proxies and reverse-proxies, which assist origin clients by performing requests to origin servers on their behalf, and forwarding back the corresponding responses.

CoAP supports also group communication scenarios [I-D.ietf-core-groupcomm-bis], where clients can send a one-to-many request targeting all the servers in the group, e.g., by using IP multicast. Like for one-to-one communication, group settings can also rely on intermediaries [I-D.ietf-core-groupcomm-proxy].

The security protocol Object Security for Constrained RESTful Environments (OSCORE) [RFC8613] can be used to protect CoAP messages between two endpoints at the application layer, especially achieving end-to-end security in the presence of (non-trusted) intermediaries. When CoAP group communication is used, the same can be achieved by means of the security protocol Group OSCORE [I-D.ietf-core-oscore-groupcomm].

For a number of use cases (see Appendix A), it is required and/or beneficial that communications are secured also between an application endpoint (i.e., a CoAP origin client/server) and an intermediary, as well as between two adjacent intermediaries in a chain. This especially applies to the communication leg between the CoAP origin client and the adjacent intermediary acting as next hop towards the CoAP origin server.

In such cases, and especially if the origin client already uses OSCORE to achieve end-to-end security with the origin server, it would be convenient that OSCORE is used also to secure communications between the origin client and its next hop.

However, the original specification [RFC8613] does not define how OSCORE can be used to protect CoAP messages in that communication leg, or how to generally process CoAP messages with OSCORE at an intermediary. In fact, this would require to consider also an intermediary as an "OSCORE endpoint".

This document fills this gap, and updates [RFC8613] as follows.

* It defines how to use OSCORE for protecting a CoAP message in the communication leg between: i) an origin client/server and an intermediary; or ii) two adjacent intermediaries in an intermediary chain. That is, besides origin clients/servers, it allows also intermediaries to be "OSCORE endpoints".

* It defines rules to escalate the protection of a CoAP option that is originally meant to be unprotected or only integrity-protected by OSCORE. This results in both encrypting and integrity-protecting a CoAP option whenever it is possible.

* It admits a CoAP message to be secured by multiple, nested OSCORE protections applied in sequence. For instance, this is the case when the message is OSCORE-protected end-to-end between the origin client and origin server, and the result is further OSCORE-protected over the leg between the current and next hop (e.g., the origin client and the adjacent intermediary acting as next hop towards the origin server).

Furthermore, this document updates [RFC8768], as it explicitly defines the CoAP option Hop-Limit to be of Class U for OSCORE (see Section 3). In the case where the Hop-Limit option is first added to a request by an origin client instead of an intermediary, this update avoids undesired overhead in terms of message size and ensures that the first intermediary in the chain enforces the intent of the origin client in detecting forwarding loops.

This document does not specify any new signaling method to guide the message processing on the different endpoints. In particular, every endpoint is always able to understand what steps to take on an incoming message, depending on the presence of the OSCORE option and of other CoAP options intended for an intermediary.

The approach defined in this document can be seamlessly adopted also when Group OSCORE is used, for protecting CoAP messages in group communication scenarios that rely on intermediaries.

1.1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Readers are expected to be familiar with the terms and concepts related to CoAP [RFC7252], OSCORE [RFC8613], and Group OSCORE [I-D.ietf-core-oscore-groupcomm]. This document especially builds on concepts and mechanics related to intermediaries such as CoAP forward-proxies and reverse-proxies.

In addition, this document uses the following terms.

* Source application endpoint: an origin client producing a request, or an origin server producing a response.

* Destination application endpoint: an origin server intended to consume a request, or an origin client intended to consume a response.

* Application endpoint: a source or destination application endpoint.

* Source OSCORE endpoint: an endpoint protecting a message with OSCORE or Group OSCORE.

* Destination OSCORE endpoint: an endpoint unprotecting a message with OSCORE or Group OSCORE.

* OSCORE endpoint: a source or destination OSCORE endpoint. An OSCORE endpoint is not necessarily also an application endpoint with respect to a certain message.

* Hop: an endpoint in the end-to-end path between two application endpoints included.

* Proxy-related options: either of the following (set of) CoAP options that a proxy can use to understand where to forward a CoAP request. These CoAP options are defined in [RFC7252] and [I-D.ietf-core-href].

  - The Proxy-Uri option or the Proxy-Cri option. These are relevant when using a forward-proxy.

  - The set of CoAP options comprising the Proxy-Scheme option or the Proxy-Scheme-Number option, together with any of the Uri-* options. This is relevant when using a forward-proxy.

  - The set of CoAP options comprising any of the Uri-Host, Uri-Port, and Uri-Path options, when those are not used together with the Proxy-Scheme option or the Proxy-Scheme-Number option. This is relevant when using a reverse-proxy.

2. Message Processing

This section defines the processing of CoAP messages with OSCORE.

Appendix B provides a number of examples where the approach defined in this document is used to protect message exchanges.

2.1. Deviations from the Original Message Processing

This document introduces the following two main deviations from the original OSCORE specification [RFC8613].

* An "OSCORE endpoint", as a producer/consumer of an OSCORE option, can be not only an application endpoint (i.e., an origin client or server), but also an intermediary such as a proxy.

  Hence, OSCORE can be used between an origin client/server and a proxy, as well as between two proxies in an intermediary chain.

* A CoAP message can be secured by multiple OSCORE protections applied in sequence. In such a case, the final result is a message with nested OSCORE protections. Hence, following a decryption, the resulting message might legitimately include an OSCORE option, and thus have in turn to be decrypted.

  The most common case is expected to consider a message protected with up to two OSCORE layers, i.e.: i) an inner layer, protecting the message end-to-end between the origin client and the origin server acting as application endpoints; and ii) an outer layer, protecting the message between a certain OSCORE endpoint and the other OSCORE endpoint adjacent in the intermediary chain.

However, a message can also be protected with a higher, arbitrary number of nested OSCORE layers, e.g., in scenarios relying on a longer chain of intermediaries. For instance, the origin client can sequentially apply multiple OSCORE layers to a request, each of which to be consumed and removed by one of the intermediaries in the chain, until the origin server is reached and it consumes the innermost OSCORE layer.

An OSCORE endpoint SHOULD define the maximum number of OSCORE layers that it is able to apply (remove) when processing an outgoing (incoming) CoAP message. The defined limit has to appropriately reflect the security requirements of the application. At the same time, such a limit is typically bounded by the maximum number of OSCORE Security Contexts that can be active at the endpoint, and also by the number of intermediary OSCORE endpoints that have been explicitly set up by the communicating parties.

If its defined limit is reached when processing a CoAP message, an OSCORE endpoint MUST NOT perform any further OSCORE processing on that message. If the message is an outgoing request and it requires further OSCORE processing beyond the set limit, the endpoint MUST abort the message sending. If the message is an incoming request and it requires further OSCORE processing beyond the set limit, the endpoint MUST reply with a 4.01 (Unauthorized) error response. The endpoint protects such a response by applying the same OSCORE layers that it successfully removed from the corresponding incoming request, but in the reverse order than the one according to which they were removed (see Section 2.5).

2.2. Protection of CoAP Options

Let us consider a sender endpoint that, when protecting an outgoing message M, applies the i-th OSCORE layer in sequence, by using the OSCORE Security Context shared with another OSCORE endpoint X.

As usual, the sender endpoint encrypts and integrity-protects the CoAP options included in M that are processed as Class E for OSCORE, as per Sections 4.1.1 and 4.1.3 of [RFC8613].

Per the update made by this document, the sender endpoint MUST perform the procedure defined below for each CoAP option OPT that is included in M and is originally specified only as an outer option (Class U or I) for OSCORE. This procedure does not apply to options that are specified (also) as Class E. Depending on the outcome of this procedure, the sender endpoint processes OPT as per its original Class U or I, or instead as Class E.

Before protecting M by using the OSCORE Security Context shared with another OSCORE endpoint X and applying the i-th OSCORE layer in sequence, the sender endpoint performs the following steps, for each CoAP option OPT that is included in M and is originally specified only as an outer option (Class U or I) for OSCORE. Appendix C provides an overview of these steps through a state diagram.

Note that the sender endpoint can assess some conditions only "to the best of its knowledge". This is due to the possible presence of a reverse-proxy standing for X and whose presence as reverse-proxy is, by definition, expected to be unknown to the sender endpoint.

1. If the sender endpoint has added OPT to M, then this algorithm moves to Step 2. Otherwise, this algorithm moves to Step 4.

2. If, to the best of the sender endpoint’s knowledge, X is a consumer of OPT, then this algorithm moves to Step 3. Otherwise, this algorithm moves to Step 4.

3. If, to the best of the sender endpoint’s knowledge, X is the immediately next consumer of OPT, then this algorithm moves to Step 5. Otherwise, this algorithm moves to Step 9.

4. If any of the following conditions holds, then this algorithm moves to Step 6. Otherwise, this algorithm moves to Step 9.

   * To the best of the sender endpoint’s knowledge, X is the next hop for the sender endpoint; or

   * To the best of the sender endpoint’s knowledge, the next hop for the sender endpoint is not the immediately next consumer of OPT.

5. If X needs to access OPT before having removed the i-th OSCORE layer or in order to remove the i-th OSCORE layer, then this algorithm moves to Step 9. Otherwise, this algorithm moves to Step 6.

6. If OPT is the Uri-Host or Uri-Port option, then this algorithm moves to Step 7. Otherwise, this algorithm moves to Step 8.

7. If M includes the Proxy-Scheme or Proxy-Scheme-Number option, then this algorithm moves to Step 8. Otherwise, this algorithm moves to Step 9.

8. The sender endpoint determines that OPT will be processed as Class E for OSCORE, i.e., both encrypted and integrity-protected. Then, the sender endpoint terminates this algorithm.

9. The sender endpoint determines that OPT will be processed as per its original Class U or I for OSCORE. Then, the sender endpoint terminates this algorithm.

Compared to what is defined in Section 5.7.1 of [RFC7252], a new requirement is introduced for a proxy that acts as OSCORE endpoint. That is, for each CoAP option OPT included in an outgoing message M that the proxy protects with OSCORE, the proxy has to be able to recognize OPT and thus be aware of the original Class of OPT for OSCORE.

If a proxy does not recognize a CoAP option included in M, then the proxy MUST stop processing M and performs the following actions.

* If M is a request, then the proxy MUST respond with a 4.02 (Bad Option) error response to (the previous hop towards) the origin client.

* If M is a response, then the proxy MUST send a 5.02 (Bad Gateway) error response to (the previous hop towards) the origin client.

In either case, this may result in protecting the error response over that communication leg, as per Section 2.5.

2.3. Processing of an Outgoing Request

The rules from Section 2.2 apply when processing an outgoing request message, with the following additions.

When a source application endpoint applies multiple OSCORE layers in sequence to protect an outgoing request, and it uses an OSCORE Security Context shared with the other application endpoint, then the first OSCORE layer MUST be applied by using that Security Context.

After that, the source application endpoint further protects the outgoing request, by applying one OSCORE layer for each intermediary with which it shares an OSCORE Security Context. When doing so, the source application endpoint applies those OSCORE layers in the same order according to which those intermediaries are positioned in the chain, starting from the one closest to the other application endpoint and moving backwards towards the one closest to the source application endpoint.

2.4. Processing of an Incoming Request

Upon receiving a request REQ, the recipient endpoint performs the actions described in the following steps. Appendix D provides an overview of these steps through a state diagram.

1. If REQ includes proxy-related options, the endpoint moves to Step 2. Otherwise, the endpoint moves to Step 3.

2. The endpoint proceeds as defined below, depending on which of the two following conditions holds.

   * REQ includes either of the following (set) of CoAP options: the Proxy-Uri option; the Proxy-Cri option; the Proxy-Scheme option or the Proxy-Scheme-Number option, together with any of the Uri-* options.

     If the endpoint is not configured to be a forward-proxy, it MUST stop processing the request and MUST respond with a 5.05 (Proxying Not Supported) error response to (the previous hop towards) the origin client, as per Section 5.10.2 of [RFC7252]. This may result in protecting the error response over that communication leg, as per Section 2.5.

     Otherwise, the endpoint MUST check whether forwarding this request to (the next hop towards) the origin server is an acceptable operation to perform, according to the endpoint's configuration and a possible authorization enforcement. This check can be based, for instance, on the specific OSCORE Security Context that the endpoint used to decrypt the incoming message, before performing this step.

     In case the check fails, the endpoint MUST stop processing the request and MUST respond with a 4.01 (Unauthorized) error response to (the previous hop towards) the origin client, as per Section 5.10.2 of [RFC7252]. This may result in protecting the error response over that communication leg, as per Section 2.5.

     Instead, in case the check succeeds, the endpoint consumes the proxy-related options as per Section 5.7.2 of [RFC7252]. In particular, the endpoint checks whether the authority (host and port) of the request URI identifies the endpoint itself. In such a case, the endpoint moves to Step 1.

     Otherwise, the endpoint forwards REQ to (the next hop towards) the origin server according to the request URI, unless differently indicated in REQ, e.g., by means of any of its CoAP options. For instance, a forward-proxy does not forward a request that includes proxy-related options together with the Listen-To-Multicast-Notifications option (see Section 12 of [I-D.ietf-core-observe-multicast-notifications]).

If the endpoint forwards REQ to (the next hop towards) the origin server, this may result in (further) protecting REQ over that communication leg, as per Section 2.3.

After that, the endpoint does not take any further action.

* REQ does not include the Proxy-Scheme option or the Proxy-Scheme-Number option, but it includes one or more Uri-Path options, and/or the Uri-Host option, and/or the Uri-Port option.

  If the endpoint is not configured to be a reverse-proxy, or what is targeted by the value of the Uri-Path, Uri-Host, and Uri-Port options is not intended to support reverse-proxy functionalities, then the endpoint proceeds to Step 3.

  Otherwise, the endpoint MUST check whether forwarding this request to (the next hop towards) the origin server is an acceptable operation to perform, according to the endpoint's configuration and a possible authorization enforcement. This check can be based, for instance, on the specific OSCORE Security Context that the endpoint used to decrypt the incoming message, before performing this step.

  In case the check fails, the endpoint MUST stop processing the request and MUST respond with a 4.01 (Unauthorized) error response to (the previous hop towards) the origin client, as per Section 5.10.2 of [RFC7252]. This may result in protecting the error response over that communication leg, as per Section 2.5.

  Otherwise, the endpoint consumes the present Uri-Path, Uri-Host, and Uri-Port options, and forwards REQ to (the next hop towards) the origin server, unless differently indicated in REQ, e.g., by means of any of its CoAP options.

  If the endpoint forwards REQ to (the next hop towards) the origin server, this may result in (further) protecting REQ over that communication leg, as per Section 2.3.

  After that, the endpoint does not take any further action.

  Note that, when forwarding REQ, the endpoint might not remove all the Uri-Path options originally present, e.g., in case the next hop towards the origin server is a reverse-proxy.

3. The endpoint proceeds as defined below, depending on which of the two following conditions holds.

* REQ does not include an OSCORE option.

  If the endpoint does not have an application to handle REQ, it MUST stop processing the request and MAY respond with a 4.00 (Bad Request) error response to (the previous hop towards) the origin client. This may result in protecting the error response over that communication leg, as per Section 2.5.

  Otherwise, the endpoint delivers REQ to the application.

* REQ includes an OSCORE option.

  If REQ includes any Uri-Path options, the endpoint MUST stop processing the request and MAY respond with a 4.00 (Bad Request) error response to (the previous hop towards) the origin client. This may result in protecting the error response over that communication leg, as per Section 2.5.

  Otherwise, the endpoint MUST check whether decrypting the request is an acceptable operation to perform, according to the endpoint's configuration and a possible authorization enforcement, and in view of the (previous hop towards the) origin client being the alleged request sender. This check can be based, for instance, on considering the source addressing information of the request, and then asserting whether the OSCORE Security Context indicated by the OSCORE option is not only available to use, but also present in a local list of OSCORE Security Contexts that are usable to decrypt a request from the alleged request sender.

  In case the check fails, the endpoint MUST stop processing the request and MUST respond with a 4.01 (Unauthorized) error response to (the previous hop towards) the origin client, as per Section 5.10.2 of [RFC7252]. This may result in protecting the error response over that communication leg, as per Section 2.5.

  Instead, in case the check succeeds, the endpoint decrypts REQ using the OSCORE Security Context indicated by the OSCORE option, which results in the decrypted request REQ*. The possible presence of an OSCORE option in REQ* is not treated as an error situation.

  If the OSCORE processing results in an error, the endpoint MUST stop processing the request and performs error handling as per Section 8.2 of [RFC8613] or Sections 7.2 and 8.4 of [I-D.ietf-core-oscore-groupcomm], in case OSCORE or Group OSCORE is used, respectively. In case the endpoint sends an

error response to (the previous hop towards) the origin client, this may result in protecting the error response over that communication leg, as per Section 2.5.

Otherwise, REQ takes REQ*, and the endpoint moves to Step 1.

2.5. Processing of an Outgoing Response

The rules from Section 2.2 apply when processing an outgoing response message, with the following additions.

When a source application endpoint applies multiple OSCORE layers in sequence to protect an outgoing response, and it uses an OSCORE Security Context shared with the other application endpoint, then the first OSCORE layer MUST be applied by using that Security Context.

The sender endpoint protects the response by applying the same OSCORE layers that it removed from the corresponding incoming request, but in the reverse order than the one according to which they were removed.

In case the response is an error response, the sender endpoint protects it by applying the same OSCORE layers that it successfully removed from the corresponding incoming request, but in the reverse order than the one according to which they were removed.

2.6. Processing of an Incoming Response

The recipient endpoint removes the same OSCORE layers that it added when protecting the corresponding outgoing request, but in the reverse order than the one according to which they were added.

When doing so, the possible presence of an OSCORE option in the decrypted response following the removal of an OSCORE layer is not treated as an error situation, unless it occurs after having removed as many OSCORE layers as were added in the corresponding outgoing request. In such a case, the endpoint MUST stop processing the response.

3. OSCORE Processing of the Hop-Limit Option

The CoAP option Hop-Limit is defined in [RFC8768] and can be used to detect forwarding loops through a chain of proxies. The first proxy in the chain that understands the option can include it in a received request (if not present already), then sets a proper integer value specifying the desired maximum number of hops, and finally forward the request to the next hop. Any following proxy that understands the option decrements the option value and forwards the request if

the new value is different from zero, or returns a 5.08 (Hop Limit Reached) error response otherwise.

[RFC8768] does not define how the Hop-Limit option is processed by OSCORE. As a consequence, the default behavior specified in Section 4.1 of [RFC8613] applies, i.e., the Hop-Limit option has to be processed as Class E for OSCORE.

However, this results in additionally and unjustifiably increasing the size of OSCORE-protected CoAP messages, in case the origin client is the first endpoint to add the Hop-Limit option in a CoAP request. In the typical scenario where the origin client and the origin server share an OSCORE Security Context, the origin client including the Hop-Limit option in a request will also protect that option when protecting the request end-to-end for the origin server, per the default processing mentioned above. After that, the origin client sends the request to its adjacent proxy in the chain, which will add an outer Hop-Limit option to be effectively considered from then on as the message is forwarded towards the origin server.

This undesirably prevents the first proxy in the chain from enforcing the intent of the origin client, which was presumably in the position to specify a better initial value for the Hop-Limit option. While this does not fundamentally prevent the detection of forwarding loops, it is conducive to deviations from the intention of the origin client. Moreover, it results in undesired overhead due to the presence of the inner Hop-Limit option included by the client. That inner option will not be visible by the proxies in the chain and therefore will serve no practical purpose, but it will still be conveyed within the request as this traverses each hop towards the origin server.

In order to prevent that by construction, this section updates [RFC8768] by explicitly defining the Hop-Limit option to be of Class U for OSCORE.

Therefore, with reference to the scenario discussed above, the origin client does not protect the Hop-Limit option when protecting the request end-to-end for the origin server, thus allowing the first proxy in the chain to see and process the Hop-Limit option as expected.

When OSCORE is used at proxies like defined in this document, the process defined in Section 2.2 seamlessly applies also to the Hop-Limit option. Therefore, in a scenario where the origin client also shares an OSCORE Security Context with the first proxy in the chain, the origin client does not protect the Hop-Limit option end-to-end for the origin server, but it does protect the option when protecting the request for that proxy by means of their shared OSCORE Security Context.

4. Caching of OSCORE-Protected Responses

Although it is not possible as per the original OSCORE specification [RFC8613], effective cacheability of OSCORE-protected responses at proxies can be achieved. To this end, the approach defined in [I-D.amsuess-core-cachable-oscore] can be used, as based on Deterministic Requests protected with the pairwise mode of Group OSCORE [I-D.ietf-core-oscore-groupcomm] used end-to-end between an origin client and an origin server. The applicability of this approach is limited to requests that are safe (in the REST sense) to process and do not yield side effects at the origin server.

In particular, this approach requires both the origin client and the origin server to have already joined the correct OSCORE group. Then, starting from the same plain CoAP request, different clients in the OSCORE group are able to deterministically generate a same Deterministic Request protected with Group OSCORE, which is sent to a proxy for being forwarded to the origin server. The proxy can effectively cache the resulting OSCORE-protected response from the server, since the same plain CoAP request will result again in the same Deterministic Request and thus will produce a cache hit at the proxy.

When using this approach, the following also applies in addition to what is defined in Section 2.4 and Section 2.6, when processing incoming messages at a proxy that implements caching of responses.

* Upon receiving a request from (the previous hop towards) the origin client, the proxy checks if specifically the message available during the execution of Step 2 in Section 2.4 produces a cache hit.

  That is, such a message: i) is exactly the one to be forwarded to (the next hop towards) the origin server, in case no cache hit occurs; and ii) is the result of an OSCORE decryption at the proxy, in case OSCORE is used on the communication leg between the proxy and (the previous hop towards) the origin client.

* Upon receiving a response from (the next hop towards) the origin server, the proxy first removes the same OSCORE layers that it added when protecting the corresponding outgoing request, as defined in Section 2.6.

  Then, the proxy stores specifically that resulting response message in its cache. That is, such a stored message is exactly the one to be forwarded to (the previous hop towards) the origin client.

The specific rules about serving a request with a cached response are defined in Section 5.6 of [RFC7252], as well as in Section 7 of [I-D.ietf-core-groupcomm-proxy] for group communication scenarios.

5. Establishment of OSCORE Security Contexts

Like the original OSCORE specification [RFC8613], this document is not devoted to any particular approach that two OSCORE endpoints use for establishing an OSCORE Security Context.

At the same time, the following applies, depending on the two peers using OSCORE or Group OSCORE [I-D.ietf-core-oscore-groupcomm] to protect their communications.

* When using OSCORE, the establishment of the OSCORE Security Context can rely on the authenticated key exchange protocol Ephemeral Diffie-Hellman Over COSE (EDHOC) [RFC9528].

  Assuming that OSCORE has to be used both between the two origin application endpoints as well as between the origin client and the first proxy in the chain, it is expected that the origin client first runs EDHOC with the first proxy in the chain, and then with the origin server through the chain of proxies (see the example in Appendix B.4).

  Furthermore, the additional use of the combined EDHOC + OSCORE request defined in [RFC9668] is particularly beneficial in this case (see the example in Appendix B.5), and especially when relying on a long chain of proxies.

* The use of Group OSCORE is expected to be limited between the origin application endpoints, e.g., between the origin client and multiple origin servers. In order to join the same OSCORE group and obtain the corresponding Group OSCORE Security Context, those endpoints can use the approach defined in [I-D.ietf-ace-key-groupcomm-oscore] and based on the ACE framework for Authentication and Authorization in constrained environments [RFC9200].

For the purposes of this document, there is no need for a proxy to also be a member of the OSCORE group whose Group OSCORE Security Context is used by the origin application endpoints for protecting communications end-to-end.

6. CoAP Header Compression with SCHC

The method defined in this document enables and results in the possible protection of the same CoAP message with multiple, nested OSCORE layers. Especially when this happens, it is desirable to compress the header of protected CoAP messages, in order to improve performance and ensure that CoAP is usable also in Low-Power Wide-Area Networks (LPWANs).

To this end, it is possible to use the Static Context Header Compression and fragmentation (SCHC) framework [RFC8724]. In particular, [I-D.ietf-schc-8824-update] specifies how to use SCHC for compressing headers of CoAP messages, also when messages are protected with OSCORE. The SCHC Compression/Decompression is applicable also in the presence of CoAP proxies, and especially to the two following cases.

* In case OSCORE is not used at all, the SCHC processing occurs hop-by-hop, by relying on SCHC Rules that are consistently shared between two adjacent hops.

* In case OSCORE is used only end-to-end between the application endpoints, then an Inner SCHC Compression/Decompression and an Outer SCHC Compression/Decompression are performed (see Section 8.2 of [I-D.ietf-schc-8824-update]). In particular, the following holds.

  The SCHC processing occurs end-to-end as to the Inner SCHC Compression/Decompression. This relies on Inner SCHC Rules that are shared between the two application endpoints, which act as OSCORE endpoints and share the used OSCORE Security Context.

  The SCHC processing occurs hop-by-hop as to the Outer SCHC Compression/Decompression. This relies on Outer SCHC Rules that are shared between two adjacent hops.

When using the method defined in this document, and thus enabling also an intermediary proxy to be an OSCORE endpoint, the SCHC processing above is generalized as specified below.

When processing an outgoing CoAP message, a sender endpoint proceeds as follows.

* The sender endpoint performs one Inner SCHC Compression for each OSCORE layer applied to the outgoing message.

  Each Inner SCHC Compression occurs before protecting the message with that OSCORE layer, and relies on the SCHC Rules that are shared with the other OSCORE endpoint.

* The sender endpoint performs exactly one Outer SCHC Compression.

  This occurs after having performed all the intended OSCORE protections of the outgoing message, and relies on the SCHC Rules that are shared with the (next hop towards the) destination application endpoint.

That is, with respect to the SCHC Compression/Decompression processing, the following holds.

An Inner SCHC Compression is intended for a destination OSCORE endpoint, which performs the following steps.

1. It decrypts an incoming message with the OSCORE Security Context shared with the other OSCORE endpoint.

2. It performs the corresponding Inner SCHC Decompression, by relying on the SCHC Rules shared with the other OSCORE endpoint.

An Outer SCHC Compression is intended for the (next hop towards the) destination application endpoint, which performs the following steps.

1. It performs a corresponding Outer SCHC Decompression on an incoming message, by relying on the SCHC Rules shared with the previous hop towards the destination application endpoint.

2. Unless it is exactly the destination application endpoint, it performs a new Outer SCHC Compression on the result from the previous step, by relying on the SCHC Rules shared with the (next hop towards the) destination application endpoint. Then, it sends the result to the (next-hop towards the) destination application endpoint.

Note that the generalization above does not alter the core approach, design choices, and features of the SCHC Compression/Decompression applied to CoAP headers.

7. Security Considerations

The same security considerations about CoAP [RFC7252] and group communication for CoAP [I-D.ietf-core-groupcomm-bis] apply to this document. The same security considerations from [RFC8613] and [I-D.ietf-core-oscore-groupcomm] apply to this document, when using OSCORE or Group OSCORE to protect exchanged messages.

Further security considerations to take into account are inherited from the specifically used CoAP options, extensions, and methods employed when relying on OSCORE or Group OSCORE.

This document does not change the security properties of OSCORE and Group OSCORE. That is, given any two OSCORE endpoints, the method defined in this document provides them with the same security guarantees that OSCORE and Group OSCORE provide in the case where such endpoints are specifically application endpoints.

If Group OSCORE is used over a communication leg and the group mode is used to apply a protection layer to a message over that leg (see Section 7 of [I-D.ietf-core-oscore-groupcomm]), then all the members of the OSCORE group that support the group mode are able to remove that protection layer, i.e., to accordingly decrypt and verify the message. Therefore, the OSCORE group should only include OSCORE endpoints for which that is acceptable.

7.1. Preserving Location Anonymity

Before decrypting an incoming request (see Step 3 in Section 2.4), the recipient endpoint checks whether decrypting the request is an acceptable operation to perform, according to the endpoint’s configuration and a possible authorization enforcement, and in the light of the alleged request sender and the OSCORE Security Context to use.

This is particularly relevant for an origin server that expects to receive messages protected end-to-end by origin clients, but only if sent by a reverse-proxy as its adjacent hop.

In such a setup, that check prevents a malicious sender endpoint C from associating the addressing information of the origin server S with the OSCORE Security Context CTX that C and S are sharing. Making such an association would compromise the location anonymity of the origin server, as otherwise afforded by the reverse-proxy.

That is, if C gains knowledge of some addressing information ADDR, then C might send a request directly addressed to ADDR and protected with CTX. A response protected with CTX would prove that ADDR is in fact the addressing information of S.

However, after performing and failing the check on the received request, S replies with a 4.01 (Unauthorized) error response that is not protected with CTX, hence preserving the location anonymity of the origin server.

7.2. Hop-Limit Option

Section 3 of this document defines that the Hop-Limit option [RFC8768] is of Class U for OSCORE. This overrides the default behavior specified in Section 4.1 of [RFC8613], according to which the option would be processed as Class E for OSCORE.

As discussed in Section 3, applying the default behavior would result in the Hop-Limit option added by the origin client being protected end-to-end for the origin server. That is, the intention of the client about performing a detection of forwarding loops would be hidden even from the first proxy in chain, which in turn adds an outer Hop-Limit option and thus further contributes to increasing the message size (see Section 3).

Instead, having defined the Hop-Limit option as Class U for OSCORE, the following holds by virtue of the procedure defined in Section 2.2.

* If the origin client and the origin server share an OSCORE Security Context, the client protects the option end-to-end for the server only when sending a request to the server directly (i.e., not via a proxy).

* If the origin client and the first proxy in the chain share an OSCORE Security Context, then the client protects the option for the proxy, while also avoiding the downsides resulting from the default behavior mentioned above.

  Otherwise, unless the communication leg between the origin client and the first proxy in the chain relies on another secure association (e.g., a DTLS connection), the Hop-Limit option included in a request sent to the proxy will be unprotected.

Fundamentally, this is not worse then when applying the default behavior mentioned above. In that case, the origin client would not be able to provide the proxy with its intention as to detecting forwarding loops, while an active on-path adversary would be able to tamper with the request and add an outer Hop-Limit option with a fraudulent value for the proxy to use.

More generally, if any two adjacent hops share an OSCORE Security Context, then the Hop-Limit option will be protected with OSCORE in the communication leg between those two hops.

If the Hop-Limit option is transported unprotected over the communication leg between two hops, then the following applies.

* A passive on-path adversary can read the option value. By possibly relying on other information such as the option value read in other communication legs, the adversary might be able to infer the topology of the network and the path used for delivering requests from the origin client.

* An active on-path adversary can add or remove the option, or alter its value. Adding the option allows the adversary to trigger an otherwise undesired process for detecting forwarding loops, e.g., as an attempt to probe the topology of the network. Removing the option results in undetectably interrupting the ongoing process for detecting forwarding loops, while altering the option value undetectably interferes with the natural unfolding of such an ongoing process.

8. IANA Considerations

This document has the following actions for IANA.

8.1. CoAP Option Numbers Registry

IANA is asked to add this document as an additional reference for the Hop-Limit option in the "CoAP Option Numbers" registry within the "Constrained RESTful Environments (CoRE) Parameters" registry group.

9. References

9.1. Normative References

[I-D.ietf-core-href]
Bormann, C. and H. Birkholz, "Constrained Resource Identifiers", Work in Progress, Internet-Draft, draft-ietf-core-href-18, 3 February 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-core-href-18>.

[I-D.ietf-core-oscore-groupcomm]
Tiloca, M., Selander, G., Palombini, F., Mattsson, J. P., and R. H旦glund, "Group Object Security for Constrained RESTful Environments (Group OSCORE)", Work in Progress, Internet-Draft, draft-ietf-core-oscore-groupcomm-24, 8 February 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-core-oscore-groupcomm-24>.

[I-D.ietf-schc-8824-update]
Tiloca, M., Toutain, L., Martinez, I., and A. Minaburo, "Static Context Header Compression (SCHC) for the Constrained Application Protocol (CoAP)", Work in Progress, Internet-Draft, draft-ietf-schc-8824-update-04, 3 March 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-schc-8824-update-04>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <https://www.rfc-editor.org/rfc/rfc7252>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8613] Selander, G., Mattsson, J., Palombini, F., and L. Seitz, "Object Security for Constrained RESTful Environments (OSCORE)", RFC 8613, DOI 10.17487/RFC8613, July 2019, <https://www.rfc-editor.org/rfc/rfc8613>.

[RFC8724] Minaburo, A., Toutain, L., Gomez, C., Barthel, D., and JC. Zuniga, "SCHC: Generic Framework for Static Context Header Compression and Fragmentation", RFC 8724, DOI 10.17487/RFC8724, April 2020, <https://www.rfc-editor.org/rfc/rfc8724>.

[RFC8768] Boucadair, M., Reddy.K, T., and J. Shallow, "Constrained Application Protocol (CoAP) Hop-Limit Option", RFC 8768, DOI 10.17487/RFC8768, March 2020, <https://www.rfc-editor.org/rfc/rfc8768>.

9.2. Informative References

[I-D.amsuess-core-cachable-oscore]
Amsüss, C. and M. Tiloca, "Cacheable OSCORE", Work in Progress, Internet-Draft, draft-amsuess-core-cachable-oscore-10, 8 January 2025, <https://datatracker.ietf.org/doc/html/draft-amsuess-core-cachable-oscore-10>.

[I-D.amsuess-t2trg-onion-coap]
Amsüss, C., Tiloca, M., and R. Hålglund, "Using onion routing with CoAP", Work in Progress, Internet-Draft, draft-amsuess-t2trg-onion-coap-03, 17 November 2024, <https://datatracker.ietf.org/doc/html/draft-amsuess-t2trg-onion-coap-03>.

[I-D.ietf-ace-coap-est-oscore]
Selander, G., Raza, S., Furuhed, M., Vuini, M., and T. Claeys, "Protecting EST Payloads with OSCORE", Work in Progress, Internet-Draft, draft-ietf-ace-coap-est-oscore-06, 21 October 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-ace-coap-est-oscore-06>.

[I-D.ietf-ace-key-groupcomm-oscore]
Tiloca, M., Park, J., and F. Palombini, "Key Management for OSCORE Groups in ACE", Work in Progress, Internet-Draft, draft-ietf-ace-key-groupcomm-oscore-16, 6 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-ace-key-groupcomm-oscore-16>.

[I-D.ietf-core-coap-pm]
Fioccola, G., Zhou, T., Nilo, M., and F. Bulgarella, "Constrained Application Protocol (CoAP) Performance Measurement Option", Work in Progress, Internet-Draft, draft-ietf-core-coap-pm-03, 3 October 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-core-coap-pm-03>.

[I-D.ietf-core-coap-pubsub]
Jimenez, J., Koster, M., and A. Kernen, "A publish-subscribe architecture for the Constrained Application Protocol (CoAP)", Work in Progress, Internet-Draft, draft-

ietf-core-coap-pubsub-18, 28 February 2025,
<https://datatracker.ietf.org/doc/html/draft-ietf-core-coap-pubsub-18>.

[I-D.ietf-core-groupcomm-bis]
Dijk, E. and M. Tiloca, "Group Communication for the Constrained Application Protocol (CoAP)", Work in Progress, Internet-Draft, draft-ietf-core-groupcomm-bis-13, 24 February 2025,
<https://datatracker.ietf.org/doc/html/draft-ietf-core-groupcomm-bis-13>.

[I-D.ietf-core-groupcomm-proxy]
Tiloca, M. and E. Dijk, "Proxy Operations for CoAP Group Communication", Work in Progress, Internet-Draft, draft-ietf-core-groupcomm-proxy-04, 3 March 2025,
<https://datatracker.ietf.org/doc/html/draft-ietf-core-groupcomm-proxy-04>.

[I-D.ietf-core-observe-multicast-notifications]
Tiloca, M., Hglund, R., Amsss, C., and F. Palombini, "Observe Notifications as CoAP Multicast Responses", Work in Progress, Internet-Draft, draft-ietf-core-observe-multicast-notifications-11, 3 March 2025,
<https://datatracker.ietf.org/doc/html/draft-ietf-core-observe-multicast-notifications-11>.

[I-D.ietf-core-transport-indication]
Amsss, C. and M. S. Lenders, "CoAP Transport Indication", Work in Progress, Internet-Draft, draft-ietf-core-transport-indication-07, 21 October 2024,
<https://datatracker.ietf.org/doc/html/draft-ietf-core-transport-indication-07>.

[LwM2M-Core]
Open Mobile Alliance, "Lightweight Machine to Machine Technical Specification - Core, Approved Version 1.2, OMA-TS-LightweightM2M_Core-V1_2-20201110-A", November 2020,
<http://www.openmobilealliance.org/release/LightweightM2M/V1_2-20201110-A/OMA-TS-LightweightM2M_Core-V1_2-20201110-A.pdf>.

[LwM2M-Gateway]
Open Mobile Alliance, "Lightweight Machine to Machine Gateway Technical Specification - Approved Version 1.1, OMA-TS-LWM2M_Gateway-V1_1-20210518-A", May 2021,
<https://www.openmobilealliance.org/release/LwM2M_Gateway/V1_1-20210518-A/OMA-TS-LWM2M_Gateway-V1_1-20210518-A.pdf>.

[LwM2M-Transport]
Open Mobile Alliance, "Lightweight Machine to Machine Technical Specification - Transport Bindings, Approved Version 1.2, OMA-TS-LightweightM2M_Transport-V1_2-20201110-A", November 2020, <http://www.openmobilealliance.org/release/LightweightM2M/V1_2-20201110-A/OMA-TS-LightweightM2M_Transport-V1_2-20201110-A.pdf>.

[RFC7030] Pritikin, M., Ed., Yee, P., Ed., and D. Harkins, Ed., "Enrollment over Secure Transport", RFC 7030, DOI 10.17487/RFC7030, October 2013, <https://www.rfc-editor.org/rfc/rfc7030>.

[RFC7641] Hartke, K., "Observing Resources in the Constrained Application Protocol (CoAP)", RFC 7641, DOI 10.17487/RFC7641, September 2015, <https://www.rfc-editor.org/rfc/rfc7641>.

[RFC8742] Bormann, C., "Concise Binary Object Representation (CBOR) Sequences", RFC 8742, DOI 10.17487/RFC8742, February 2020, <https://www.rfc-editor.org/rfc/rfc8742>.

[RFC9200] Seitz, L., Selander, G., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "Authentication and Authorization for Constrained Environments Using the OAuth 2.0 Framework (ACE-OAuth)", RFC 9200, DOI 10.17487/RFC9200, August 2022, <https://www.rfc-editor.org/rfc/rfc9200>.

[RFC9528] Selander, G., Preu Mattsson, J., and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)", RFC 9528, DOI 10.17487/RFC9528, March 2024, <https://www.rfc-editor.org/rfc/rfc9528>.

[RFC9668] Palombini, F., Tiloca, M., Hglund, R., Hristozov, S., and G. Selander, "Using Ephemeral Diffie-Hellman Over COSE (EDHOC) with the Constrained Application Protocol (CoAP) and Object Security for Constrained RESTful Environments (OSCORE)", RFC 9668, DOI 10.17487/RFC9668, November 2024, <https://www.rfc-editor.org/rfc/rfc9668>.

[TOR-SPEC] Tor Project, "Tor Specifications", <https://spec.torproject.org/>.

Appendix A. Use Cases

The approach defined in this document has been motivated by a number of use cases, which are summarized below.

A.1. CoAP Group Communication with Proxies

CoAP supports also one-to-many group communication, e.g., over IP multicast [I-D.ietf-core-groupcomm-bis], which can be protected end-to-end between origin client and origin servers by using Group OSCORE [I-D.ietf-core-oscore-groupcomm].

This communication model can be assisted by intermediaries such as a CoAP forward-proxy or reverse-proxy, which relays a group request to the origin servers. If Group OSCORE is used, the proxy is intentionally not a member of the OSCORE group. Furthermore, [I-D.ietf-core-groupcomm-proxy] defines a signaling protocol between origin client and proxy, to ensure that responses from the different origin servers are forwarded back to the origin client within a time interval set by the client, and that they can be distinguished from one another.

In particular, it is required that the proxy identifies the origin client as allowed-listed, before forwarding a group request to the servers (see Section 4 of [I-D.ietf-core-groupcomm-proxy]). This requires a security association between the origin client and the proxy, which would be convenient to provide with a dedicated OSCORE Security Context between the two, since the client is possibly using also Group OSCORE with the origin servers.

A.2. CoAP Observe Notifications over Multicast

The Observe extension for CoAP [RFC7641] allows a client to register its interest in "observing" a resource at a server. The server can then send back notification responses upon changes in the resource representation, all matching with the original observation request.

In some applications, such as pub-sub [I-D.ietf-core-coap-pubsub], multiple clients are interested in observing the same resource at the same server. Hence, [I-D.ietf-core-observe-multicast-notifications] defines a method that allows the server to send a multicast notification to all the observer clients at once, e.g., over IP multicast. To this end, the server synchronizes the clients by providing them with a common "phantom observation request", against which the following multicast notifications will match.

In case the clients and the server use Group OSCORE for end-to-end security and a proxy is also involved, an additional step is required (see Section 12 of [I-D.ietf-core-observe-multicast-notifications]). That is, clients are in turn required to provide the proxy with the obtained "phantom observation request", thus enabling the proxy to receive the multicast notifications from the server.

Therefore, it is preferable to have a security association also between each client and the proxy, in order to ensure the integrity of that information provided to the proxy (see Section 15.3 of [I-D.ietf-core-observe-multicast-notifications]). Like for the use case in Appendix A.1, this would be conveniently achieved with a dedicated OSCORE Security Context between a client and the proxy, since the client is also using Group OSCORE with the origin server.

A.3. LwM2M Client and External Application Server

The Lightweight Machine-to-Machine (LwM2M) protocol [LwM2M-Core] enables a LwM2M Client device to securely bootstrap and then register at a LwM2M Server, with which it will perform most of its following communication exchanges. As per the transport bindings specification of LwM2M [LwM2M-Transport], the LwM2M Client and LwM2M Server can use CoAP and OSCORE to secure their communications at the application layer, including during the device registration process.

Furthermore, Section 5.5.1 of [LwM2M-Transport] specifies that:

> OSCORE MAY also be used between LwM2M endpoint and non-LwM2M endpoint, e.g., between an Application Server and a LwM2M Client via a LwM2M server. Both the LwM2M endpoint and non-LwM2M endpoint MUST implement OSCORE and be provisioned with an OSCORE Security Context.

In such a case, the LwM2M Server can practically act as forward-proxy between the LwM2M Client and the external Application Server. At the same time, the LwM2M Client and LwM2M Server must continue protecting communications on their leg using their OSCORE Security Context. Like for the use case in Appendix A.1, this also allows the LwM2M Server to identify the LwM2M Client, before forwarding its request outside the LwM2M domain and towards the external Application Server.

A.4. LwM2M Gateway

The specification [LwM2M-Gateway] extends the LwM2M architecture by defining the LwM2M Gateway functionality. That is, a LwM2M Server can manage end IoT devices that are deployed "behind" the LwM2M Gateway. While it is outside the scope of that specification, it is possible for the LwM2M Gateway to use any suitable protocol with its connected end IoT devices, as well as to carry out any required protocol translation.

Practically, the LwM2M Server can send a request to the LwM2M Gateway, asking to forward it to an end IoT device. With particular reference to CoAP and the related transport binding specified in [LwM2M-Transport], the LwM2M Server acting as CoAP client sends its request to the LwM2M Gateway acting as CoAP server.

If CoAP is used in the communication leg between the LwM2M Gateway and the end IoT devices, then the LwM2M Gateway fundamentally acts as a CoAP reverse-proxy (see Section 5.7.3 of [RFC7252]). That is, in addition to its own resources, the LwM2M Gateway serves the resources hosted by each end IoT device standing behind it, as exposed by the LwM2M Gateway under a dedicated URI path. As per [LwM2M-Gateway], the first URI path segment is used as "prefix" to identify the specific IoT device, while the remaining URI path segments specify the target resource at the IoT device.

As per Section 7 of [LwM2M-Gateway], message exchanges between the LwM2M Server and the LwM2M Gateway are secured using the LwM2M-defined technologies, while the LwM2M protocol does not provide end-to-end security between the LwM2M Server and the end IoT devices. However, the approach defined in this document makes it possible to achieve both goals, by allowing the LwM2M Server to use OSCORE for protecting a message both end-to-end with the targeted end IoT device and with the LwM2M Gateway acting as reverse-proxy.

A.5. Further Use Cases

The approach defined in this document can be useful also in the following use cases relying on a proxy.

* A server aware of a suitable cross-proxy can rely on it as a third-party service, in order to indicate transports for CoAP available to that server (see Section 4 of [I-D.ietf-core-transport-indication]).

  From a security point of view, it would be convenient if the proxy could provide suitable credentials to the client, as a general trusted proxy for the system. At the same time, it can be desirable to limit the use of such a proxy to a set of clients which have permission to use it, and that the proxy can identify through a secure communication association.

  However, in order for OSCORE to be an applicable security mechanism for this scenario, OSCORE has to be terminated at the proxy. That is, it would be required for a client and the proxy to share a dedicated OSCORE Security Context and to use it for protecting their communication leg.

* The method specified in [I-D.ietf-core-coap-pm] relies on the Performance Measurement option to enable network telemetry for CoAP communications. This makes it possible to efficiently measure Round-Trip Time and message losses, both end-to-end and hop-by-hop. In particular, on-path probes such as intermediary proxies can be deployed to perform measurements hop-by-hop.

  When OSCORE is used in deployments including on-path probes, an inner Performance Measurement option is protected end-to-end between the two application endpoints and enables end-to-end measurements between those. At the same time, an outer Performance Measurement option allows also hop-by-hop measurements to be performed by relying on an on-path probe.

  Therefore, it is preferable to have a secure association with an on-path probe, in order to also ensure the integrity of the hop-by-hop measurements exchanged with the probe.

* The method specified in [I-D.ietf-ace-coap-est-oscore] enables public-key certificate enrollment for Internet of Things deployments. This leverages payload formats defined in Enrollment over Secure Transport (EST) [RFC7030], while relying on CoAP for message transfer and on OSCORE for message protection.

  In real-world deployments, an EST server issuing public-key certificates may reside outside a constrained network that includes devices acting as EST clients. In particular, the EST clients are expected to support only CoAP, while the EST server in a non-constrained network is expected to support only HTTP. This requires a CoAP-to-HTTP proxy to be deployed between the EST clients and the EST server, in order to map CoAP messages with HTTP messages across the two networks.

  Even in such a scenario, the EST server and every EST client can still effectively use OSCORE to protect their communications end-to-end. At the same time, it is desirable to have an additional secure association between the EST client and the CoAP-to-HTTP proxy, especially in order for the proxy to identify the EST client before forwarding EST messages out of the CoAP boundary of the constrained network and towards the EST server.

* A proxy may be deployed to act as an entry point to a firewalled network that only authenticated clients can join. In particular, authentication can rely on the used secure communication association between a client and the proxy. If the proxy could share a different OSCORE Security Context with each different client, then the proxy can rely on it to identify a client before forwarding messages from that client to other members of the firewalled network.

* The approach defined in this document does not pose a limit to the number of OSCORE protections applied to the same CoAP message.

  This enables more privacy-oriented scenarios based on proxy chains, where the origin client protects a CoAP request first by using the OSCORE Security Context shared with the origin server, and then by using different OSCORE Security Contexts shared with the different hops in the chain. Once received at a chain hop, the request would be stripped of the OSCORE protection associated with that hop before being forwarded to the next one.

  Building on that, it is also possible to enable the operation of hidden services and clients through onion routing with CoAP [I-D.amsuess-t2trg-onion-coap], similarly to how Tor (The Onion Router) [TOR-SPEC] enables it for TCP-based protocols.

Appendix B. Examples of Message Exchanges

This section provides a number of examples where the approach defined in this document is used to protect message exchanges.

The presented examples build on the example shown in Appendix A.1 of [RFC8613], which illustrates an origin client requesting the alarm status from an origin server through a forward-proxy.

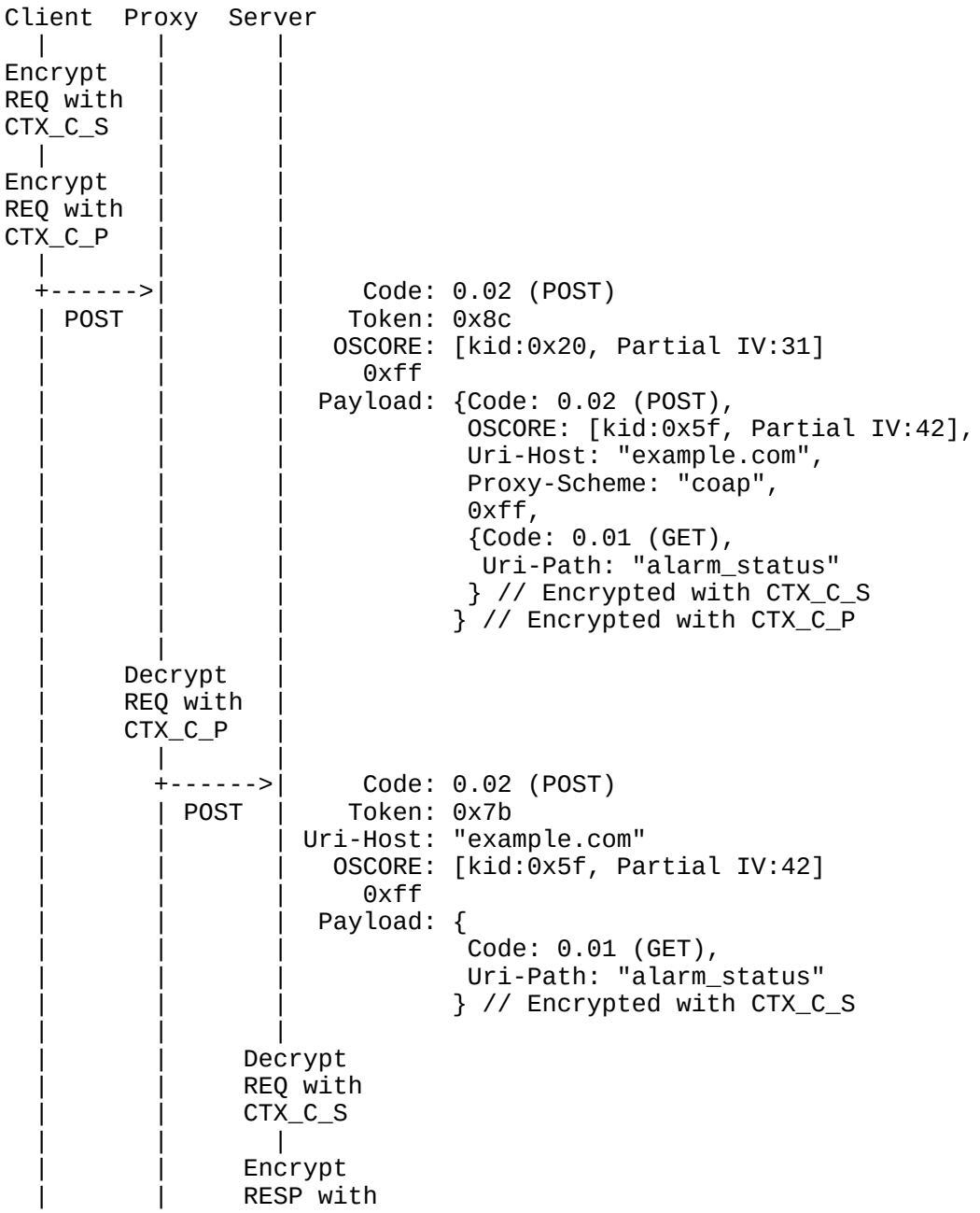
The abbreviations "REQ" and "RESP" are used to denote a request message and a response message, respectively.

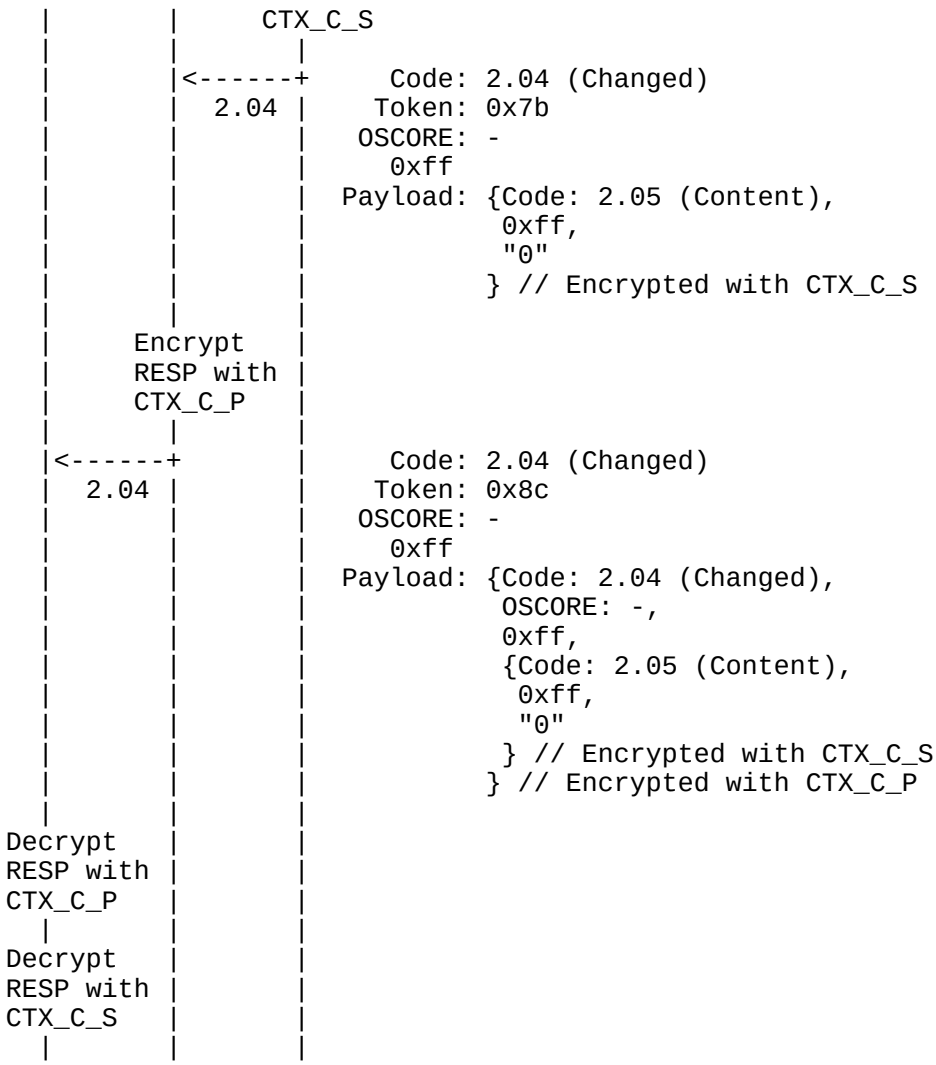
B.1. With Forward-Proxy; OSCORE: C-S, C-P

In the example shown in Figure 1, message exchanges are protected with OSCORE as follows.

* End-to-end, between the client and the server, using the OSCORE Security Context CTX_C_S. The client uses the OSCORE Sender ID 0x5f when using OSCORE with the server.

* Between the client and the proxy, using the OSCORE Security Context CTX_C_P. The client uses the OSCORE Sender ID 0x20 when using OSCORE with the proxy.





Square brackets [...] indicate content of compressed COSE object.
Curly brackets { ... } indicate encrypted data.

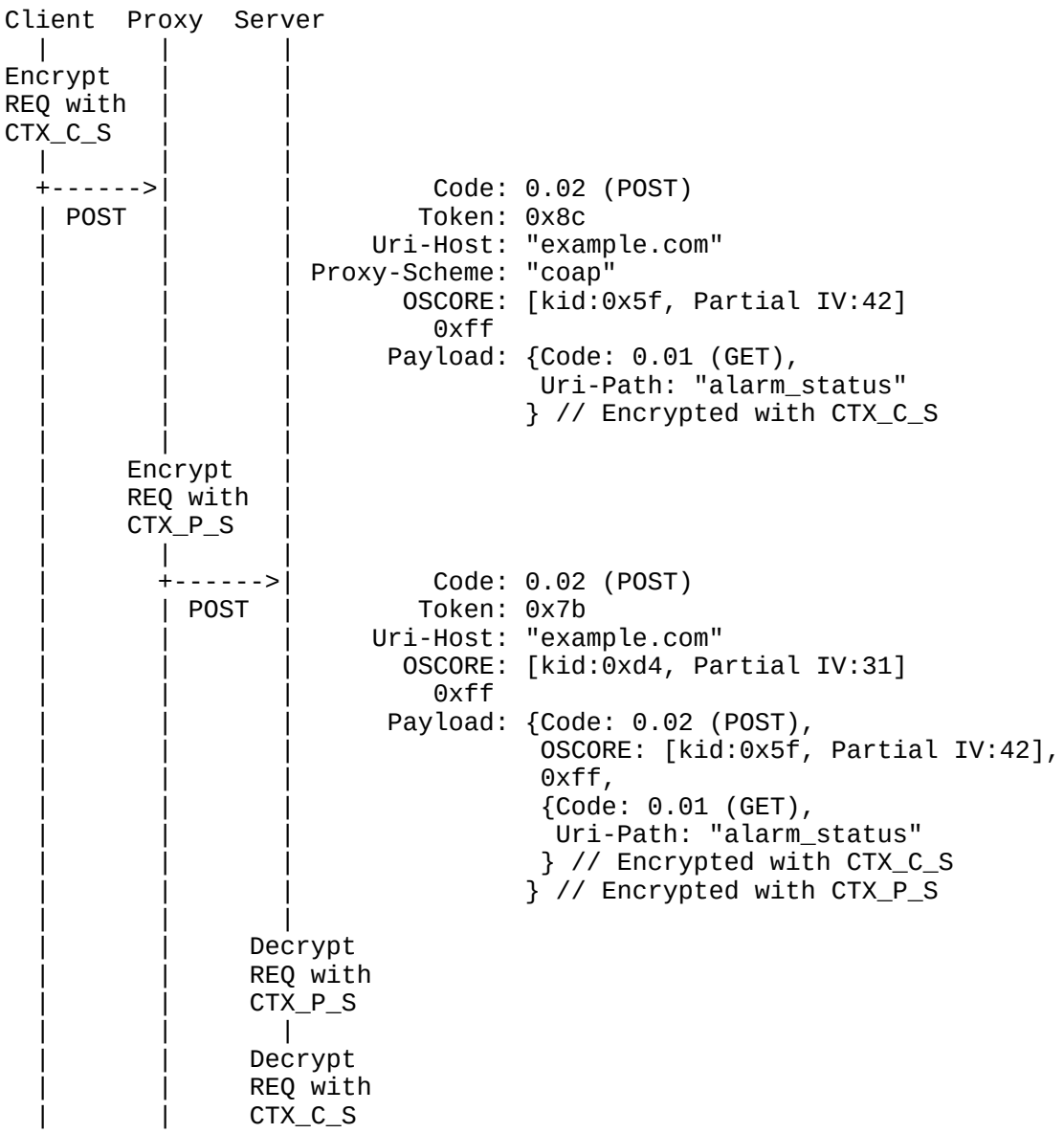
Figure 1: Use of OSCORE between Client-Server and Client-Proxy

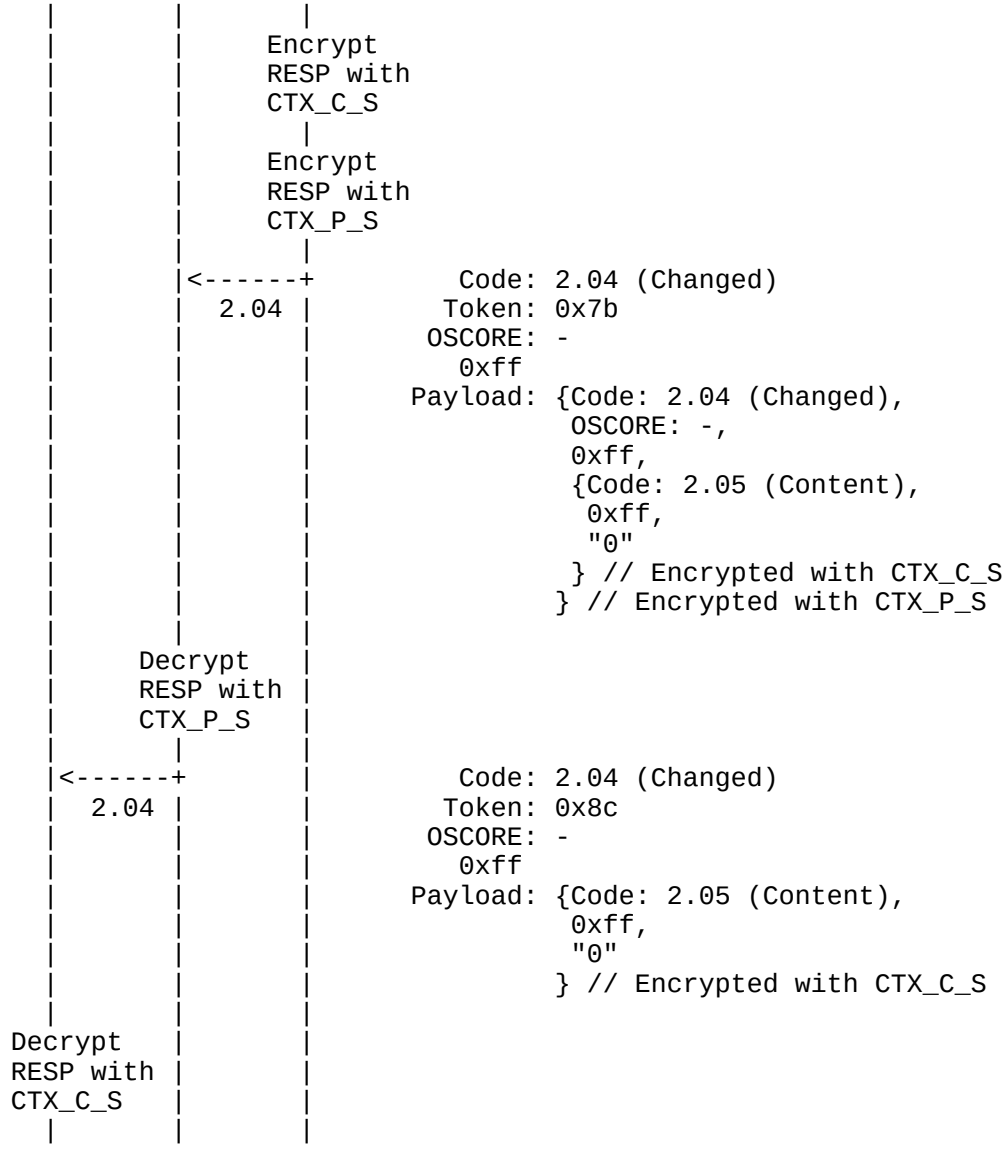
B.2. With Forward-Proxy; OSCORE: C-S, P-S

In the example shown in Figure 2, message exchanges are protected with OSCORE as follows.

* End-to-end between the client and the server, using the OSCORE Security Context CTX_C_S. The client uses the OSCORE Sender ID 0x5f when using OSCORE with the server.

* Between the proxy and the server, using the OSCORE Security Context CTX_P_S. The proxy uses the OSCORE Sender ID 0xd4 when using OSCORE with the server.





Square brackets [...] indicate content of compressed COSE object.
Curly brackets { ... } indicate encrypted data.

Figure 2: Use of OSCORE between Client-Server and Proxy-Server

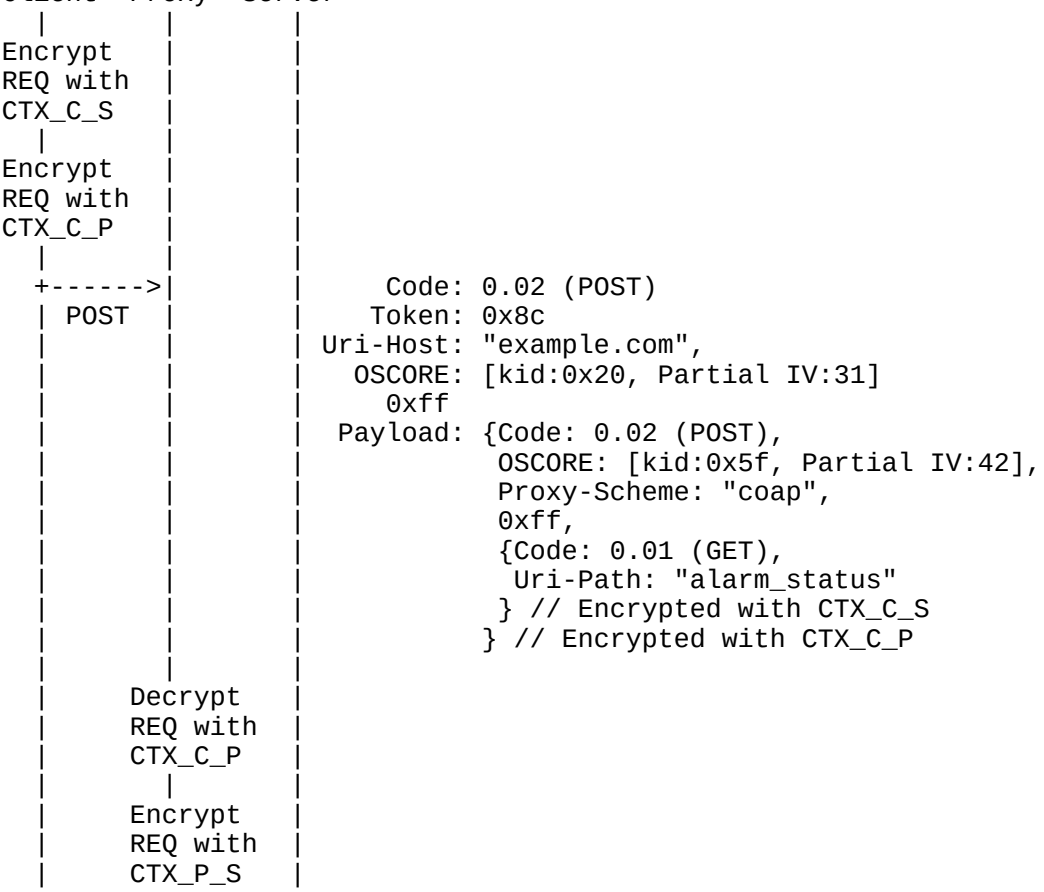
B.3. With Forward-Proxy; OSCORE: C-S, C-P, P-S

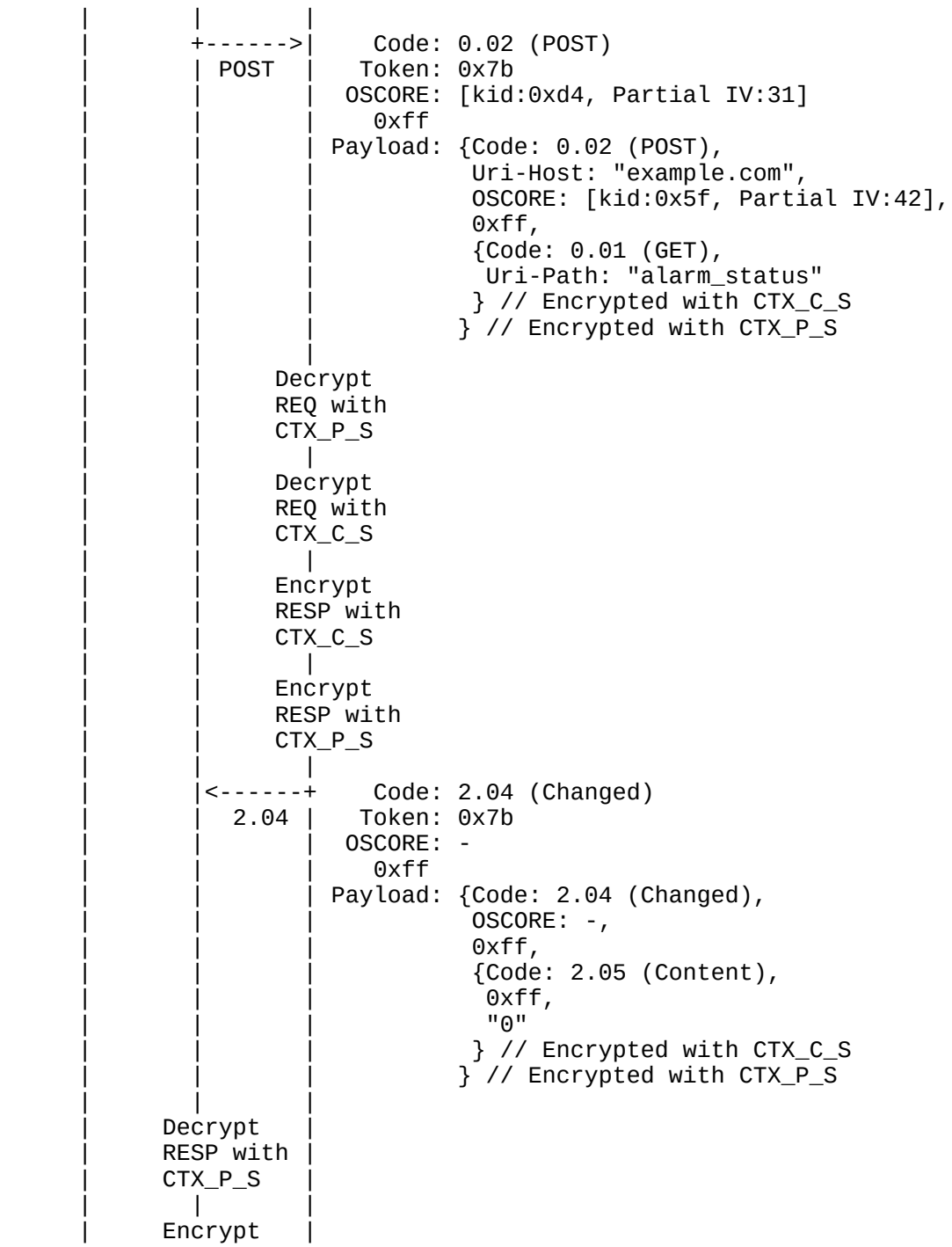
In the example shown in Figure 3, message exchanges are protected with OSCORE as follows.

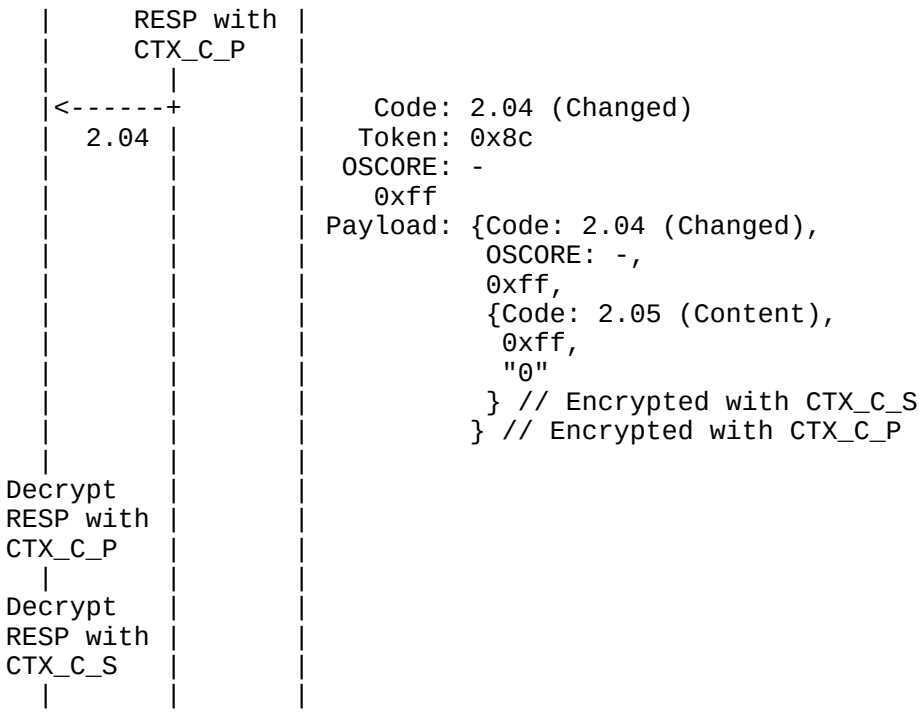
* End-to-end between the client and the server, using the OSCORE Security Context CTX_C_S. The client uses the OSCORE Sender ID 0x5f when using OSCORE with the server.

* Between the client and the proxy, using the OSCORE Security Context CTX_C_P. The client uses the OSCORE Sender ID 0x20 when using OSCORE with the proxy.

* Between the proxy and the server, using the OSCORE Security Context CTX_P_S. The proxy uses the OSCORE Sender ID 0xd4 when using OSCORE with the server.




Code: 0.02 (POST)
Token: 0x7b
OSCORE: [kid:0xd4, Partial IV:31]
0xff
Payload: {Code: 0.02 (POST),
Uri-Host: "example.com",
OSCORE: [kid:0x5f, Partial IV:42],
0xff,
{Code: 0.01 (GET),
Uri-Path: "alarm_status"
} // Encrypted with CTX_C_S
} // Encrypted with CTX_P_S
POST
Decrypt REQ with CTX_P_S
Decrypt REQ with CTX_C_S
Encrypt RESP with CTX_C_S
Encrypt RESP with CTX_P_S
Code: 2.04 (Changed)
Token: 0x7b
OSCORE: -
0xff
Payload: {Code: 2.04 (Changed),
OSCORE: -,
0xff,
{Code: 2.05 (Content),
0xff,
"0"
} // Encrypted with CTX_C_S
} // Encrypted with CTX_P_S
2.04
Decrypt RESP with CTX_P_S
Encrypt



Square brackets [...] indicate content of compressed COSE object.
Curly brackets { ... } indicate encrypted data.

Figure 3: Use of OSCORE between Client-Server, Client-Proxy, and Proxy-Server

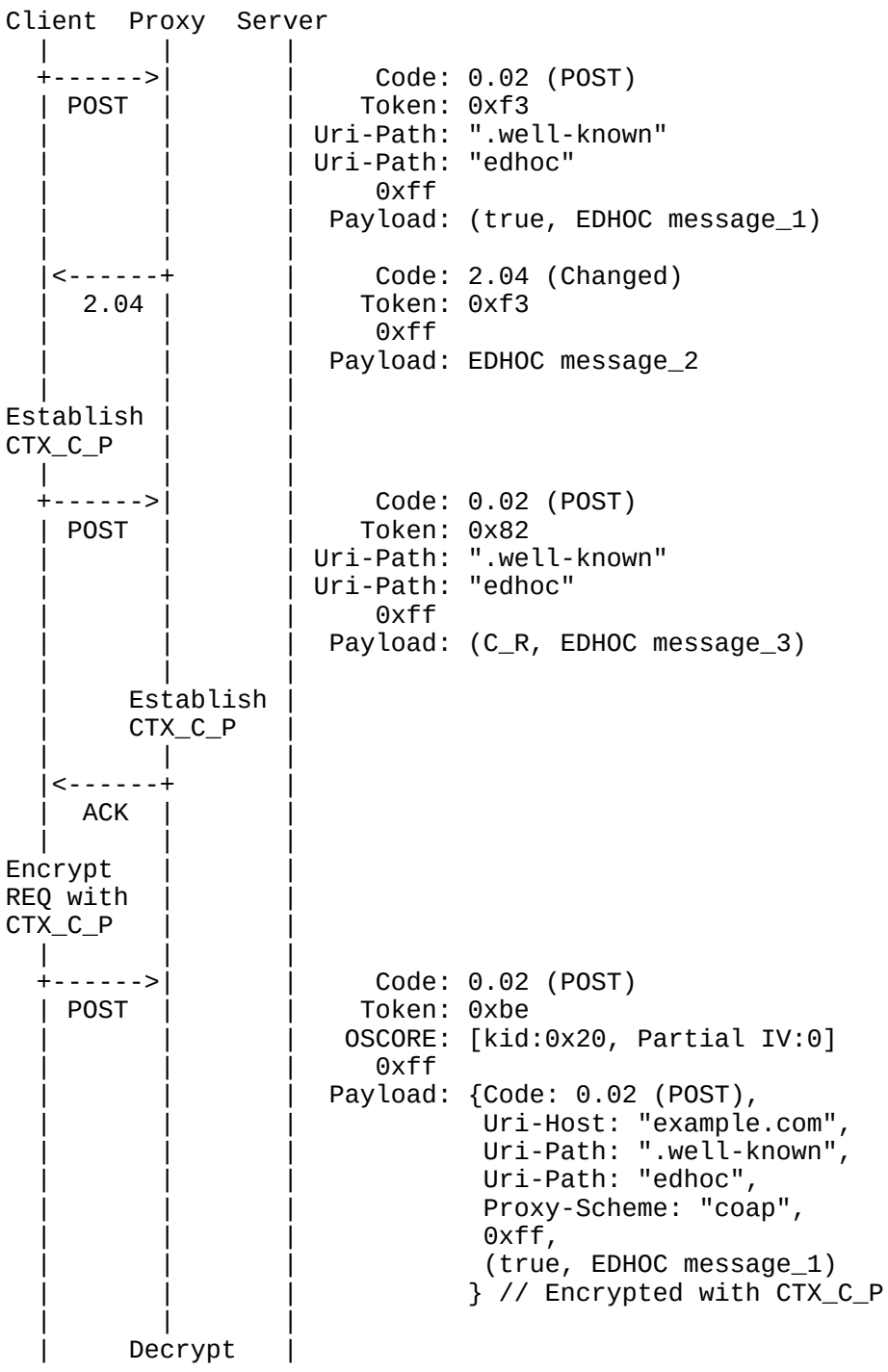
B.4. With Forward-Proxy and EDHOC; OSCORE: C-S, C-P

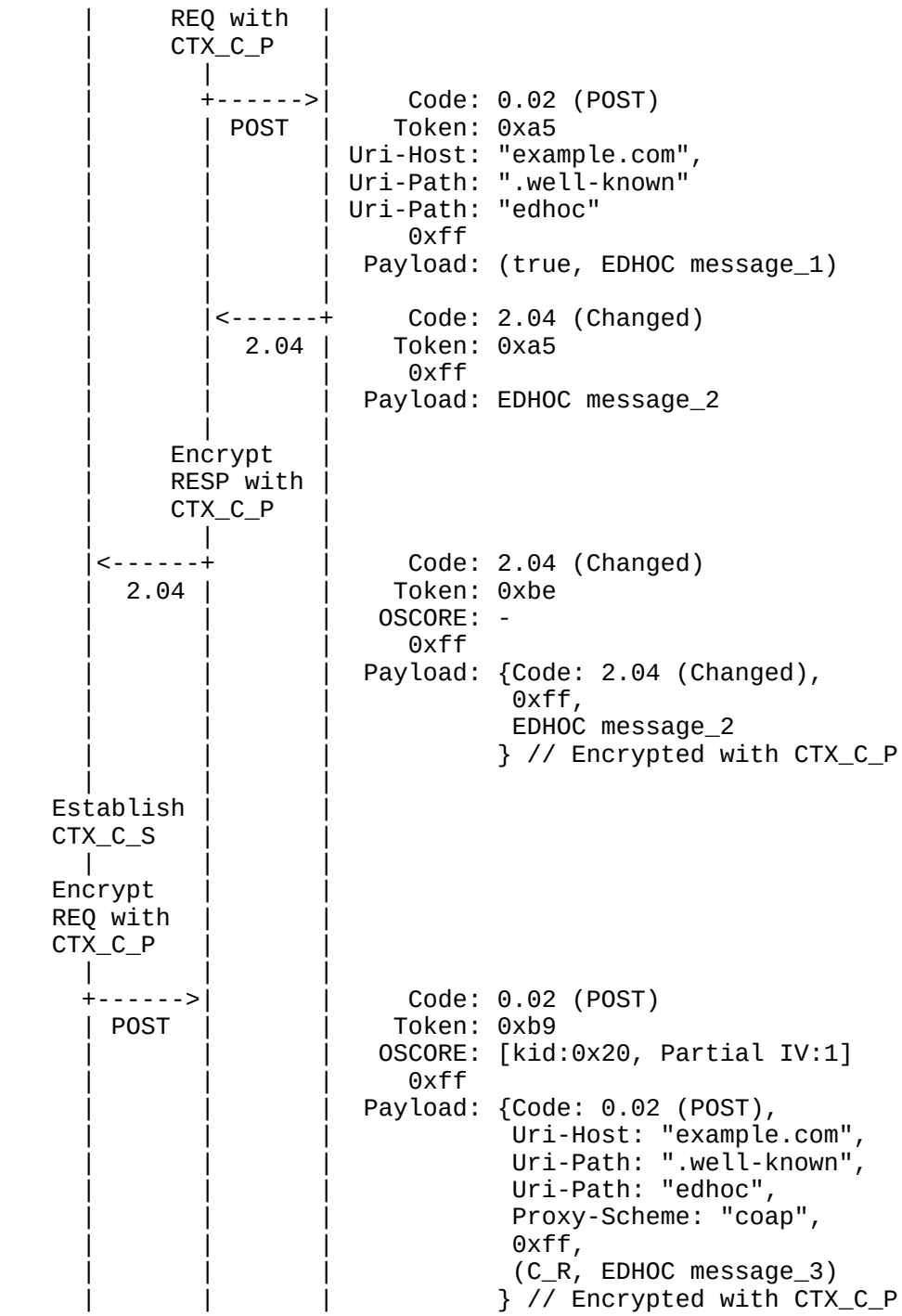
In the example shown in Figure 4, message exchanges are protected as follows.

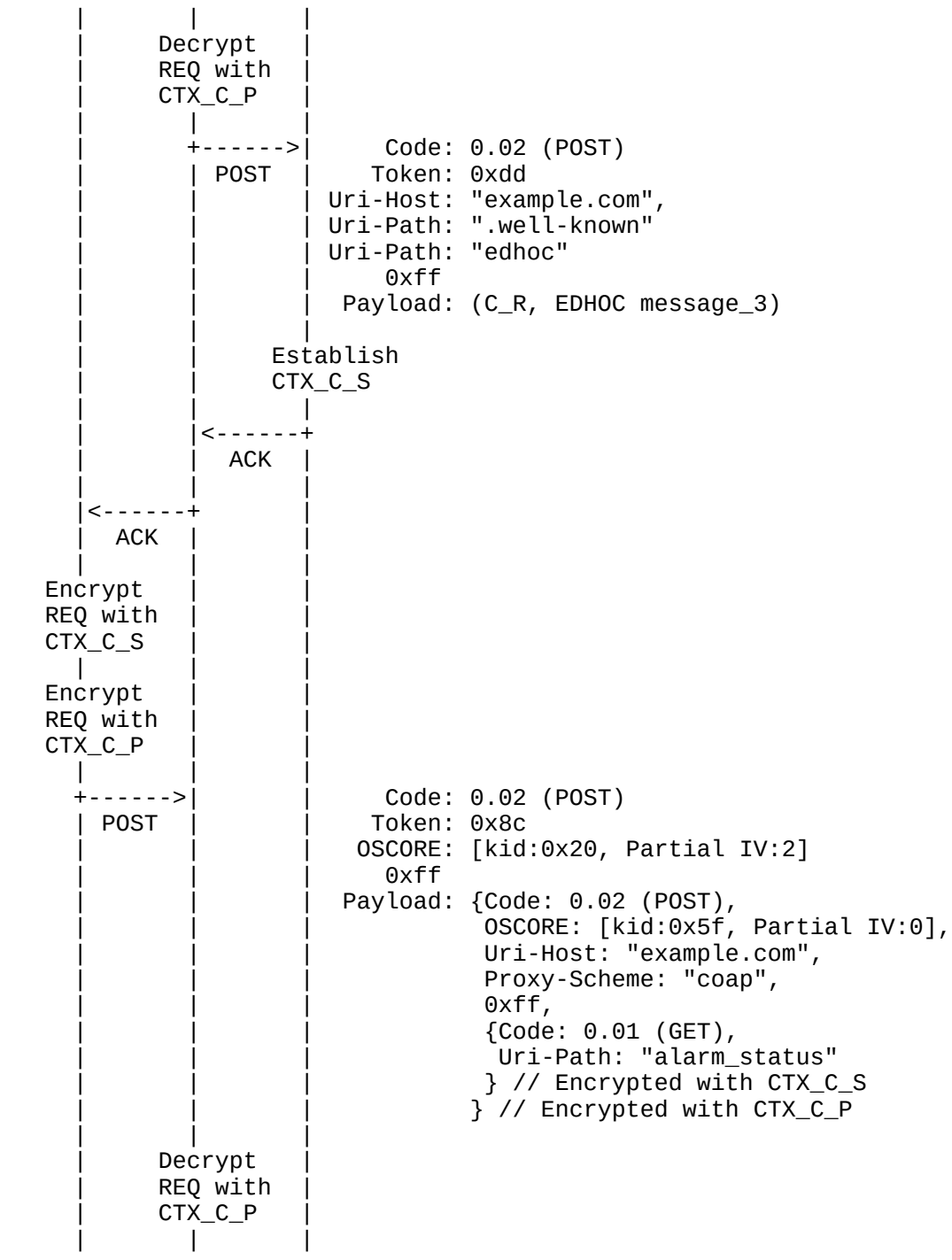
* End-to-end, between the client and the server, using the OSCORE Security Context CTX_C_S. The client uses the OSCORE Sender ID 0x5f when using OSCORE with the server.

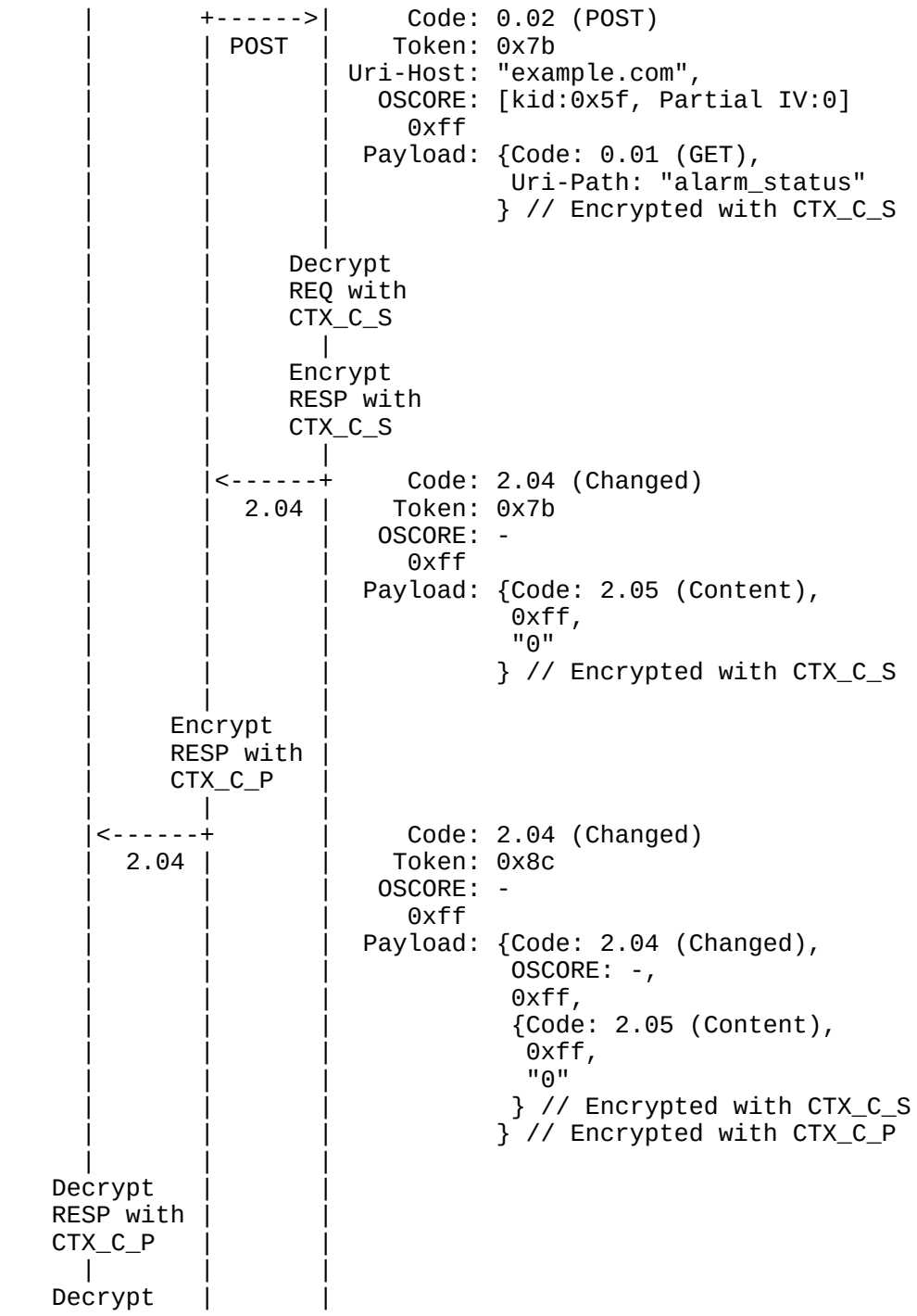
* Between the client and the proxy, using the OSCORE Security Context CTX_C_P. The client uses the OSCORE Sender ID 0x20 when using OSCORE with the proxy.

The example also shows how the client establishes an OSCORE Security Context CTX_C_P with the proxy and CTX_C_S with the server, by using the key exchange protocol EDHOC [RFC9528].


Client Proxy Server
POST
Code: 0.02 (POST)
Token: 0xf3
Uri-Path: ".well-known"
Uri-Path: "edhoc"
0xff
Payload: (true, EDHOC message_1)
2.04
Code: 2.04 (Changed)
Token: 0xf3
0xff
Payload: EDHOC message_2
Establish CTX_C_P
POST
Code: 0.02 (POST)
Token: 0x82
Uri-Path: ".well-known"
Uri-Path: "edhoc"
0xff
Payload: (C_R, EDHOC message_3)
Establish CTX_C_P
ACK
Encrypt REQ with CTX_C_P
POST
Code: 0.02 (POST)
Token: 0xbe
OSCORE: [kid:0x20, Partial IV:0]
0xff
Payload: {Code: 0.02 (POST),
Uri-Host: "example.com",
Uri-Path: ".well-known",
Uri-Path: "edhoc",
Proxy-Scheme: "coap",
0xff,
(true, EDHOC message_1)
} // Encrypted with CTX_C_P
Decrypt


REQ with CTX_C_P
POST
Code: 0.02 (POST)
Token: 0xa5
Uri-Host: "example.com",
Uri-Path: ".well-known"
Uri-Path: "edhoc"
0xff
Payload: (true, EDHOC message_1)
2.04
Code: 2.04 (Changed)
Token: 0xa5
0xff
Payload: EDHOC message_2
Encrypt RESP with CTX_C_P
2.04
Code: 2.04 (Changed)
Token: 0xbe
OSCORE: -
0xff
Payload: {Code: 2.04 (Changed),
0xff,
EDHOC message_2
} // Encrypted with CTX_C_P
Establish CTX_C_S
Encrypt REQ with CTX_C_P
POST
Code: 0.02 (POST)
Token: 0xb9
OSCORE: [kid:0x20, Partial IV:1]
0xff
Payload: {Code: 0.02 (POST),
Uri-Host: "example.com",
Uri-Path: ".well-known",
Uri-Path: "edhoc",
Proxy-Scheme: "coap",
0xff,
(C_R, EDHOC message_3)
} // Encrypted with CTX_C_P


Decrypt
REQ with
CTX_C_P
POST
Code: 0.02 (POST)
Token: 0xdd
Uri-Host: "example.com",
Uri-Path: ".well-known"
Uri-Path: "edhoc"
0xff
Payload: (C_R, EDHOC message_3)
Establish
CTX_C_S
ACK
ACK
Encrypt
REQ with
CTX_C_S
Encrypt
REQ with
CTX_C_P
POST
Code: 0.02 (POST)
Token: 0x8c
OSCORE: [kid:0x20, Partial IV:2]
0xff
Payload: {Code: 0.02 (POST),
OSCORE: [kid:0x5f, Partial IV:0],
Uri-Host: "example.com",
Proxy-Scheme: "coap",
0xff,
{Code: 0.01 (GET),
Uri-Path: "alarm_status"
} // Encrypted with CTX_C_S
} // Encrypted with CTX_C_P
Decrypt
REQ with
CTX_C_P


Code: 0.02 (POST)
Token: 0x7b
Uri-Host: "example.com",
OSCORE: [kid:0x5f, Partial IV:0]
0xff
Payload: {Code: 0.01 (GET),
Uri-Path: "alarm_status"
} // Encrypted with CTX_C_S
POST
Decrypt
REQ with
CTX_C_S
Encrypt
RESP with
CTX_C_S
Code: 2.04 (Changed)
Token: 0x7b
OSCORE: -
0xff
Payload: {Code: 2.05 (Content),
0xff,
"0"
} // Encrypted with CTX_C_S
2.04
Encrypt
RESP with
CTX_C_P
Code: 2.04 (Changed)
Token: 0x8c
OSCORE: -
0xff
Payload: {Code: 2.04 (Changed),
OSCORE: -,
0xff,
{Code: 2.05 (Content),
0xff,
"0"
} // Encrypted with CTX_C_S
} // Encrypted with CTX_C_P
2.04
Decrypt
RESP with
CTX_C_P
Decrypt

```
   RESP with |       |
   CTX_C_S   |       |
     |       |       |
```

Square brackets [...] indicate content of compressed COSE object.
Curly brackets { ... } indicate encrypted data.

(A, B) indicates a CBOR sequence [RFC8742]
of two CBOR data items A and B.

Figure 4: Use of OSCORE between Client-Server and Proxy-Server, with OSCORE Security Contexts established through EDHOC

B.5. With Forward-Proxy and EDHOC (optimized); OSCORE: C-S, C-P

In the example shown in Figure 5, message exchanges are protected as follows.

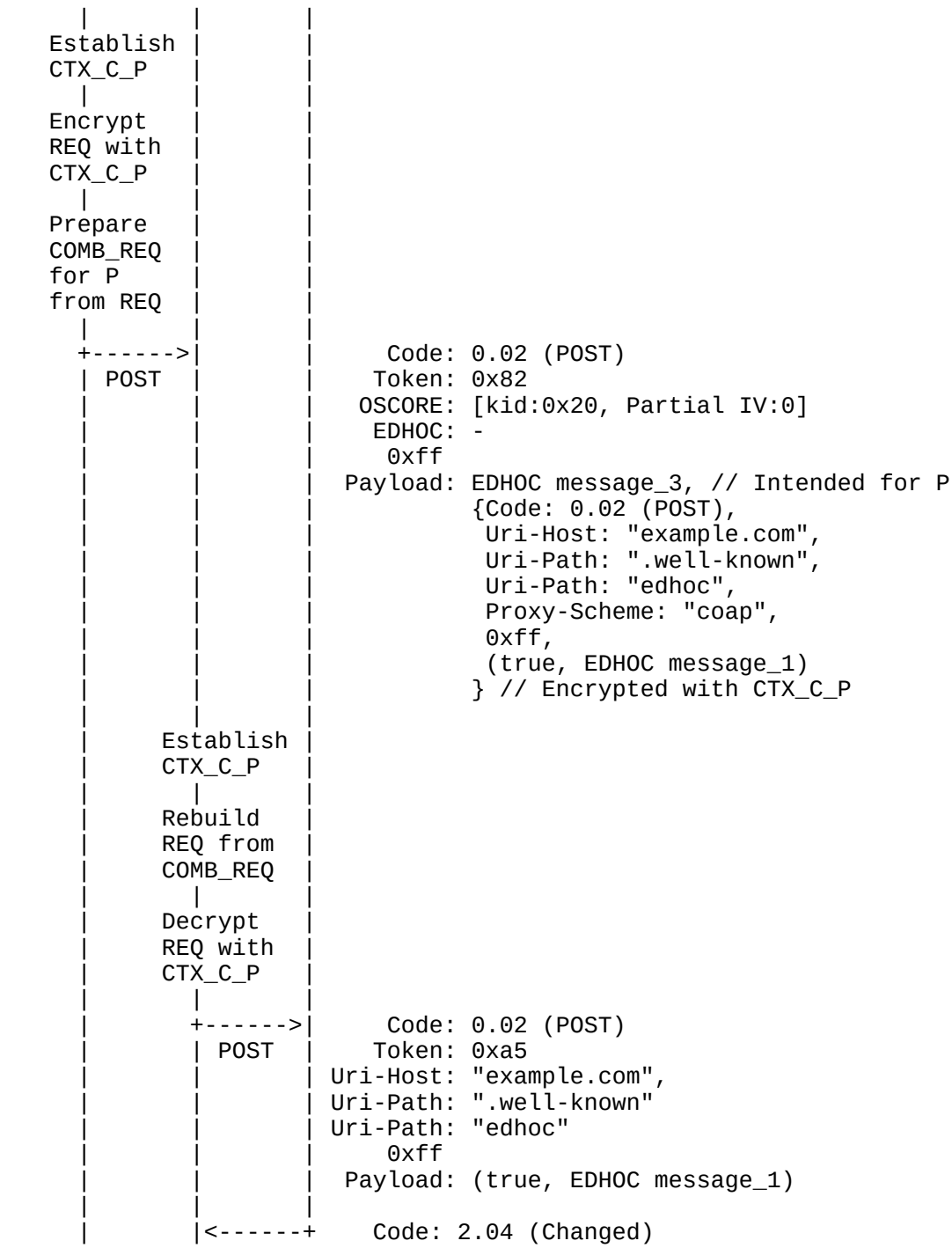
* End-to-end, between the client and the server, using the OSCORE Security Context CTX_C_S. The client uses the OSCORE Sender ID 0x5f when using OSCORE with the server.

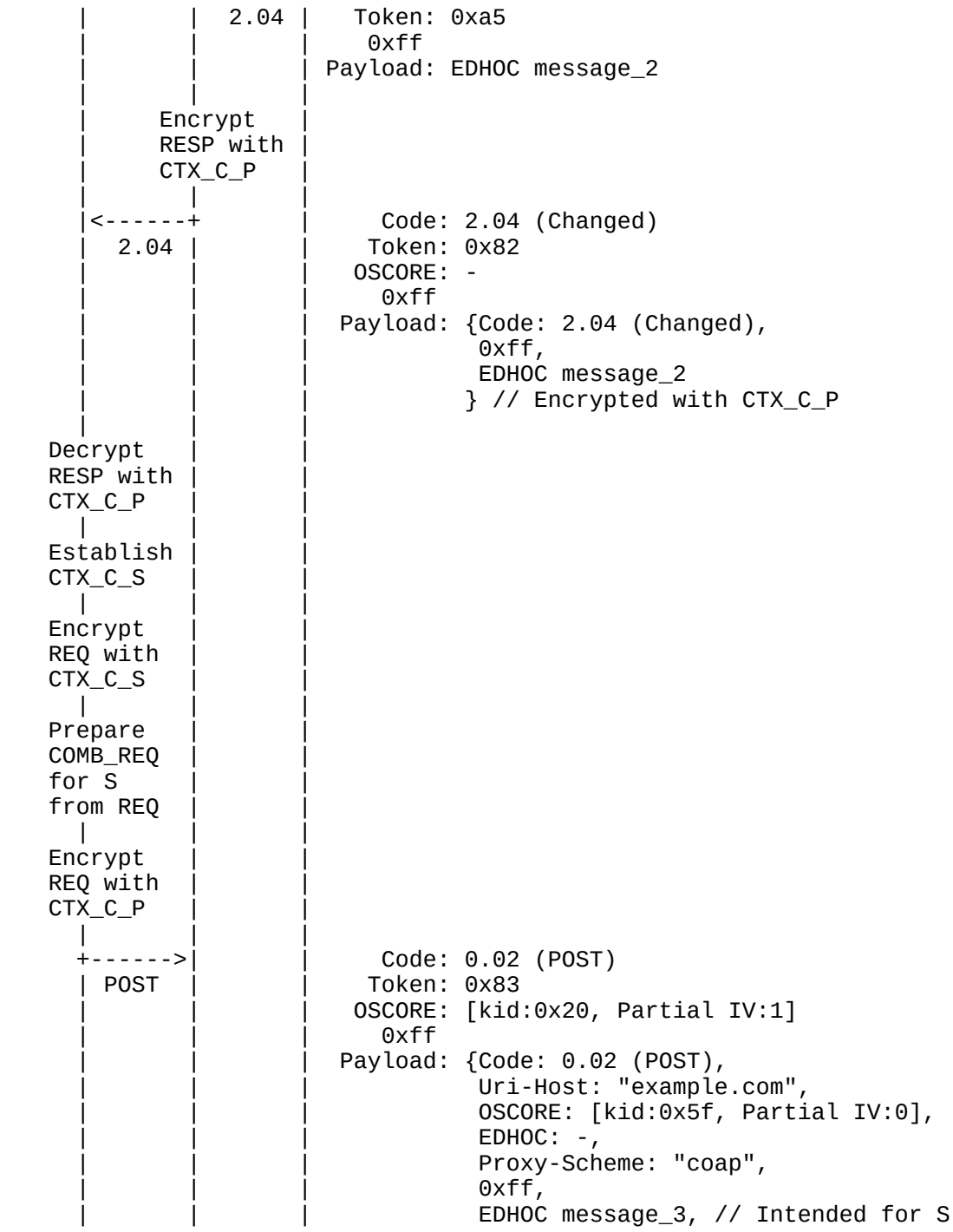
* Between the client and the proxy, using the OSCORE Security Context CTX_C_P. The client uses the OSCORE Sender ID 0x20 when using OSCORE with the proxy.

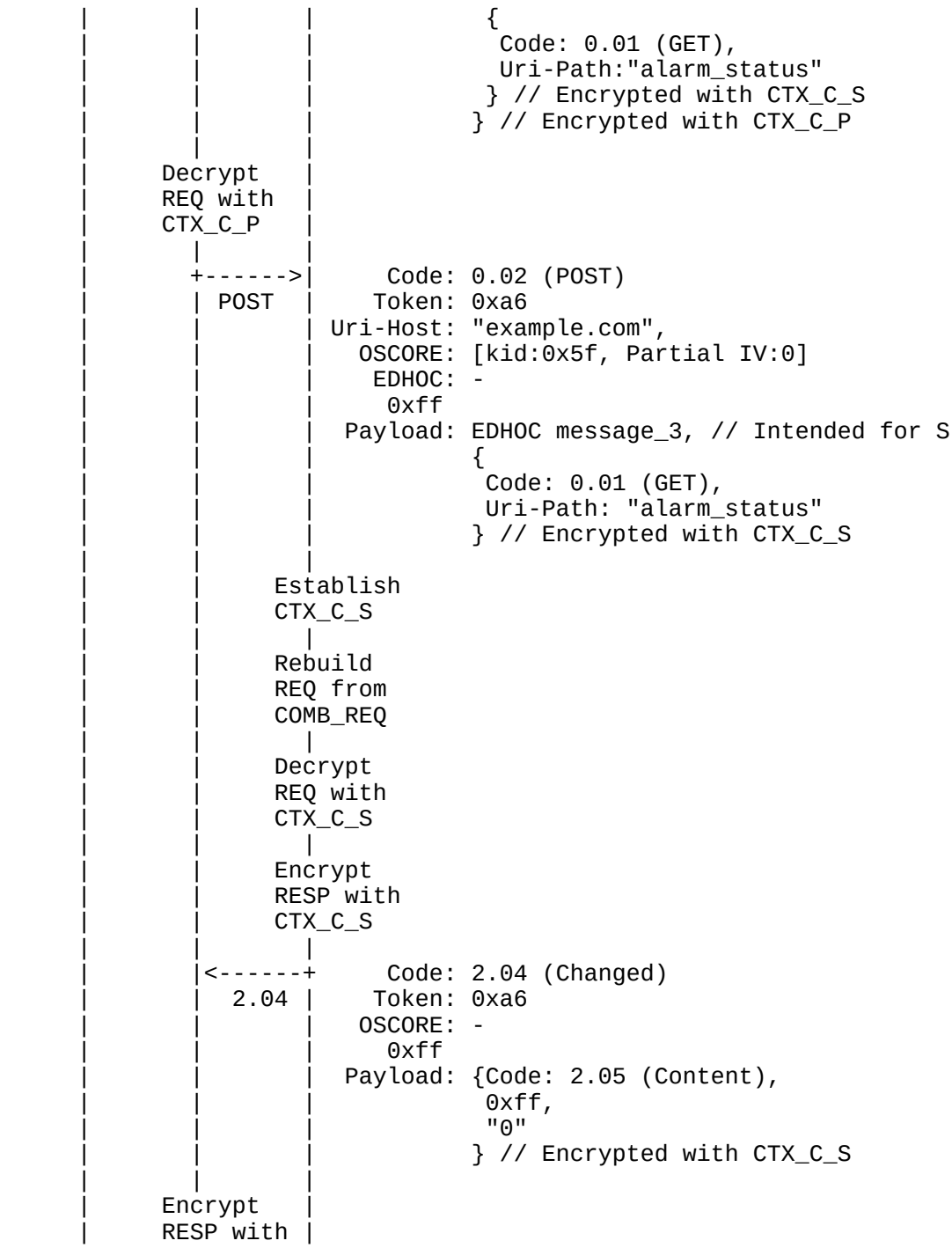
The example also shows how the client establishes an OSCORE Security Context CTX_C_P with the proxy and CTX_C_S with the server, by using the key exchange protocol EDHOC [RFC9528].

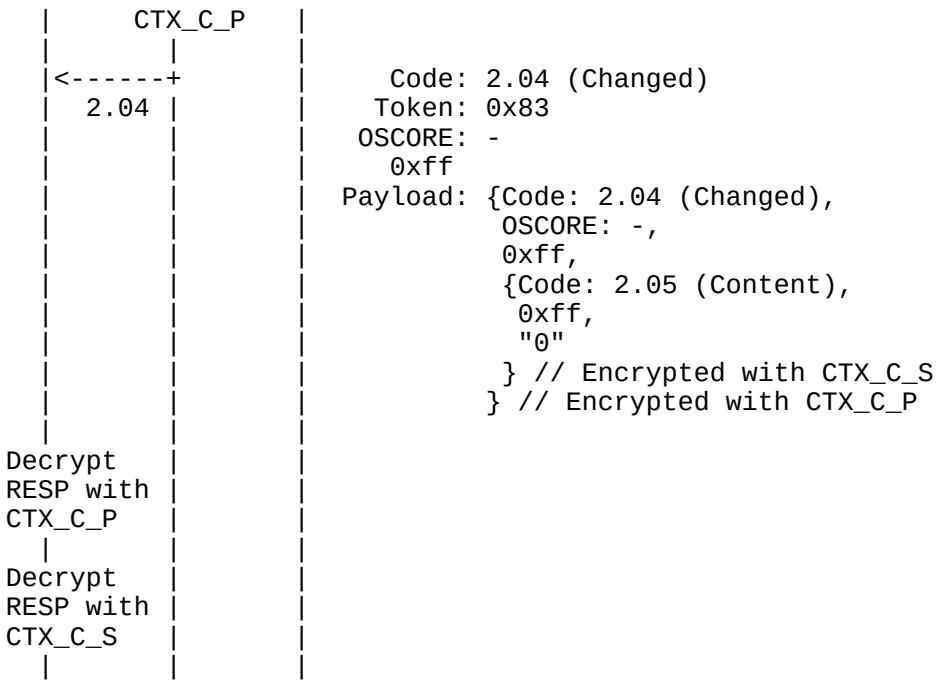
In particular, the client relies on the EDHOC + OSCORE request defined in [RFC9668] and denoted as COMB_REQ, in order to transport the last EDHOC message_3 and the first OSCORE-protected application CoAP request combined together.

```
Client  Proxy  Server
  |       |       |
  +------>|       |     Code: 0.02 (POST)
  | POST  |       |    Token: 0xf3
  |       |       | Uri-Path: ".well-known"
  |       |       | Uri-Path: "edhoc"
  |       |       |     0xff
  |       |       |  Payload: (true, EDHOC message_1)
  |       |       |
  |<------+       |    Code: 2.04 (Changed)
  |  2.04 |       |   Token: 0xf3
  |       |       |    0xff
  |       |       | Payload: EDHOC message_2
```


Establish CTX_C_P
Encrypt REQ with CTX_C_P
Prepare COMB_REQ for P from REQ
POST
Code: 0.02 (POST)
Token: 0x82
OSCORE: [kid:0x20, Partial IV:0]
EDHOC: - 0xff
Payload: EDHOC message_3, // Intended for P
{Code: 0.02 (POST),
Uri-Host: "example.com",
Uri-Path: ".well-known",
Uri-Path: "edhoc",
Proxy-Scheme: "coap",
0xff,
(true, EDHOC message_1)
} // Encrypted with CTX_C_P
Establish CTX_C_P
Rebuild REQ from COMB_REQ
Decrypt REQ with CTX_C_P
POST
Code: 0.02 (POST)
Token: 0xa5
Uri-Host: "example.com",
Uri-Path: ".well-known"
Uri-Path: "edhoc"
0xff
Payload: (true, EDHOC message_1)
Code: 2.04 (Changed)


2.04
Token: 0xa5
0xff
Payload: EDHOC message_2
Encrypt RESP with CTX_C_P
Code: 2.04 (Changed)
Token: 0x82
OSCORE: -
0xff
Payload: {Code: 2.04 (Changed),
0xff,
EDHOC message_2
} // Encrypted with CTX_C_P
Decrypt RESP with CTX_C_P
Establish CTX_C_S
Encrypt REQ with CTX_C_S
Prepare COMB_REQ for S from REQ
Encrypt REQ with CTX_C_P
POST
Code: 0.02 (POST)
Token: 0x83
OSCORE: [kid:0x20, Partial IV:1]
0xff
Payload: {Code: 0.02 (POST),
Uri-Host: "example.com",
OSCORE: [kid:0x5f, Partial IV:0],
EDHOC: -,
Proxy-Scheme: "coap",
0xff,
EDHOC message_3, // Intended for S


{
Code: 0.01 (GET),
Uri-Path:"alarm_status"
} // Encrypted with CTX_C_S
} // Encrypted with CTX_C_P
Decrypt REQ with CTX_C_P
POST
Code: 0.02 (POST)
Token: 0xa6
Uri-Host: "example.com",
OSCORE: [kid:0x5f, Partial IV:0]
EDHOC: -
0xff
Payload: EDHOC message_3, // Intended for S
{
Code: 0.01 (GET),
Uri-Path: "alarm_status"
} // Encrypted with CTX_C_S
Establish CTX_C_S
Rebuild REQ from COMB_REQ
Decrypt REQ with CTX_C_S
Encrypt RESP with CTX_C_S
2.04
Code: 2.04 (Changed)
Token: 0xa6
OSCORE: -
0xff
Payload: {Code: 2.05 (Content),
0xff,
"0"
} // Encrypted with CTX_C_S
Encrypt RESP with



Square brackets [...] indicate content of compressed COSE object.
Curly brackets { ... } indicate encrypted data.

(A, B) indicates a CBOR sequence [RFC8742]
of two CBOR data items A and B.

Figure 5: Use of OSCORE between Client-Server and Proxy-Server, with OSCORE Security Contexts established through EDHOC using the EDHOC + OSCORE request

B.6. With Reverse-Proxy; OSCORE: C-P, P-S

In the example shown in Figure 6, message exchanges are protected with OSCORE as follows.

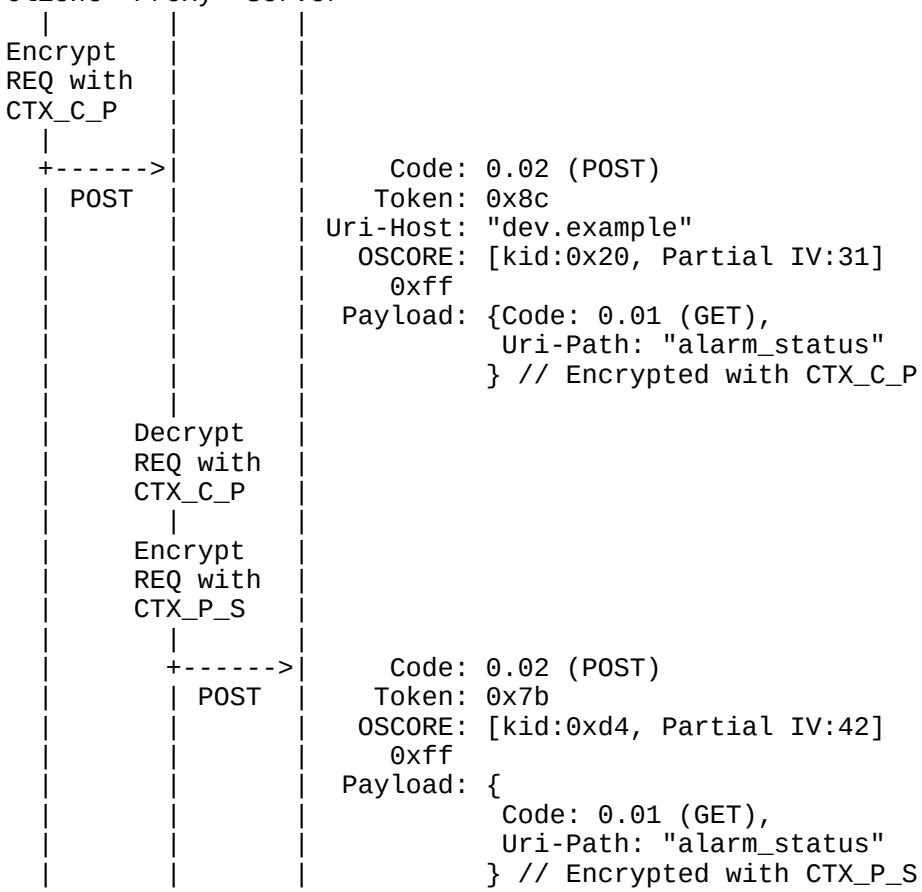
* Between the client and the proxy, using the OSCORE Security Context CTX_C_P. The client uses the OSCORE Sender ID 0x20 when using OSCORE with the proxy.

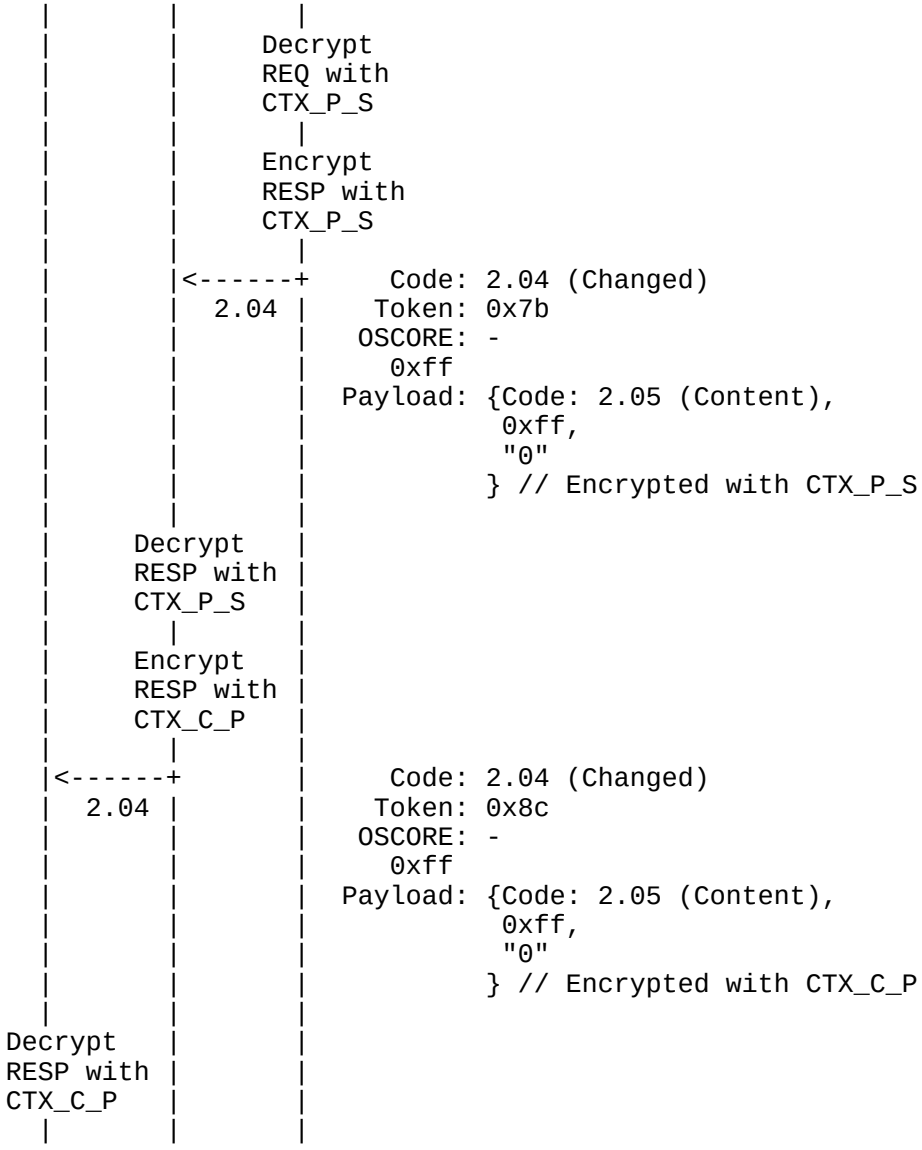
* Between the proxy and the server, using the OSCORE Security Context CTX_P_S. The proxy uses the OSCORE Sender ID 0xd4 when using OSCORE with the server.

In this example, the proxy is specifically a reverse-proxy. Like typically expected in such a case, the client is not aware of that, and believes to communicate with an origin server.

In order to determine where it has to forward an incoming request to, the proxy relies on the hostname that clients specify in the Uri-Host option of their sent requests. In particular, upon receiving a request that includes the Uri-Host option with value "dev.example", the proxy forwards the request to the origin server shown in the example.

Furthermore, this example assumes that, in the URI identifying the target resource at the server, the host component represents the destination IP address of the request as an IP-literal. Therefore, the request from the proxy to the server does not include a Uri-Host option (see Section 6.4 of [RFC7252]).





Square brackets [...] indicate content of compressed COSE object.
Curly brackets { ... } indicate encrypted data.

Figure 6: Use of OSCORE between Client-Proxy and Proxy-Server (the proxy is a reverse-proxy)

B.7. With Reverse-Proxy; OSCORE: C-S, C-P, P-S

In the example shown in Figure 7, message exchanges are protected with OSCORE as follows.

* End-to-end between the client and the server, using the OSCORE Security Context CTX_C_S. The client uses the OSCORE Sender ID 0x5f when using OSCORE with the server.

* Between the client and the proxy, using the OSCORE Security Context CTX_C_P. The client uses the OSCORE Sender ID 0x20 when using OSCORE with the proxy.

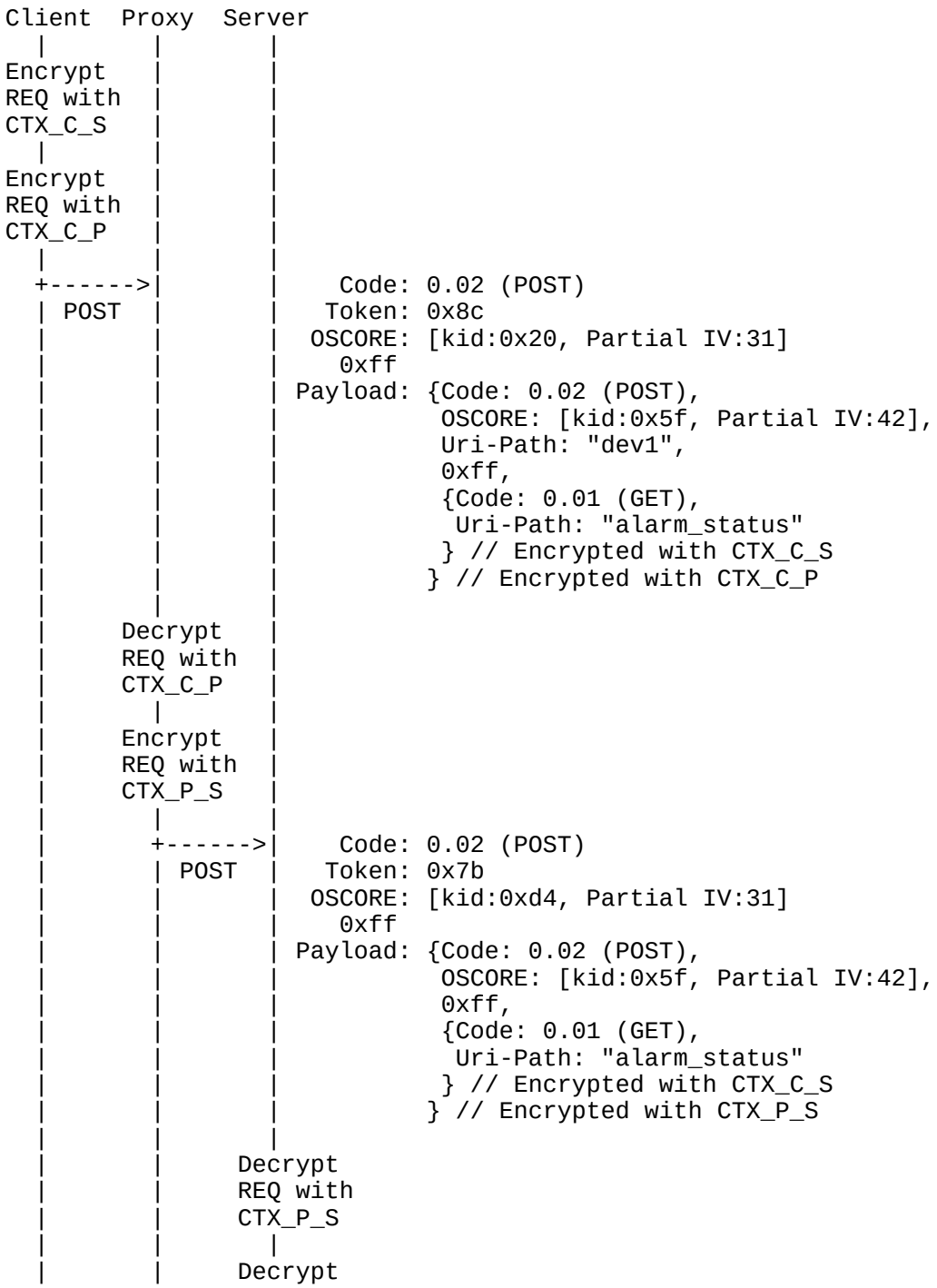
* Between the proxy and the server, using the OSCORE Security Context CTX_P_S. The proxy uses the OSCORE Sender ID 0xd4 when using OSCORE with the server.

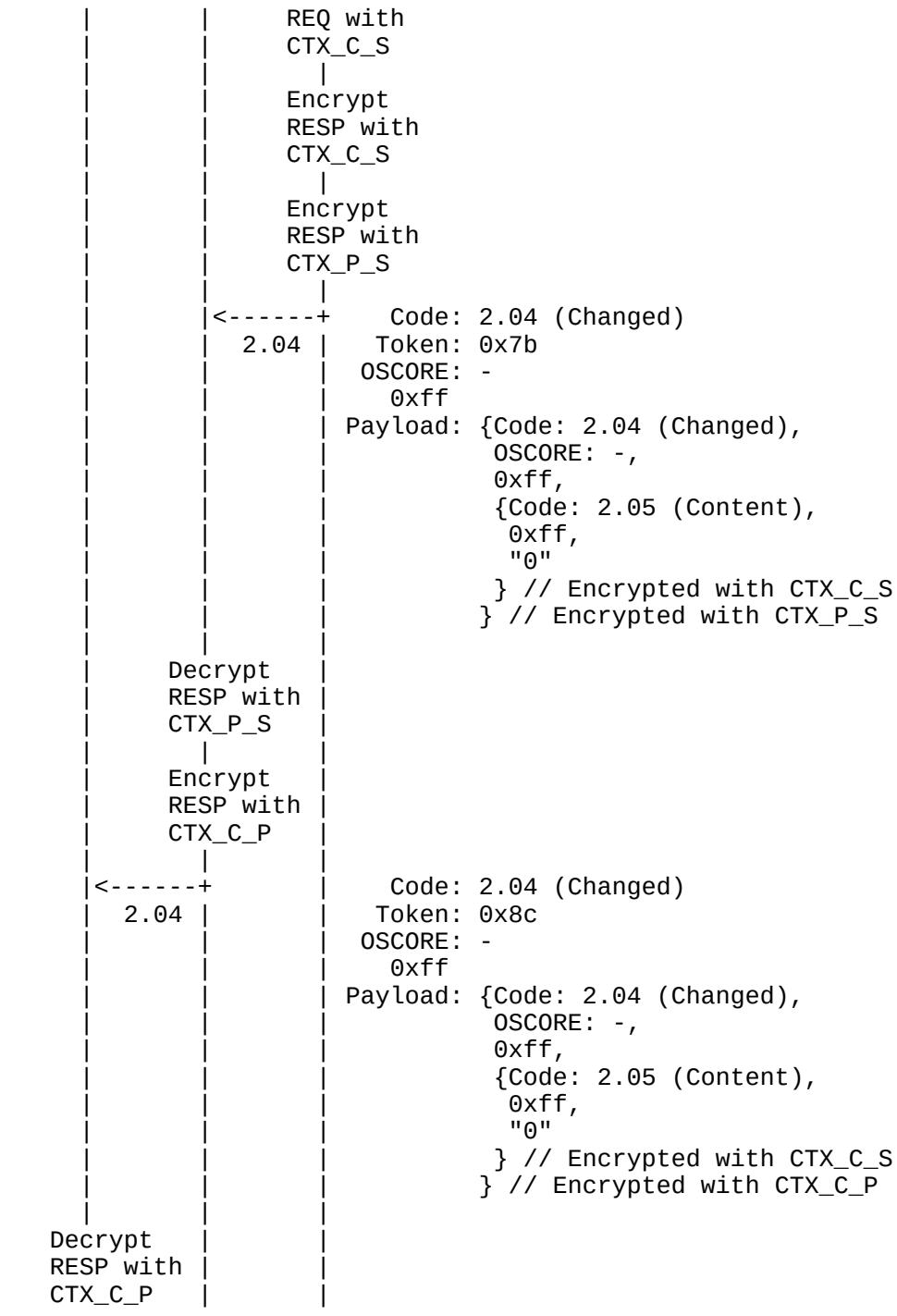
In this example, the proxy is specifically a reverse-proxy. However, unlike typically expected, the client is aware to communicate with a reverse-proxy. This is the case, e.g., in the LwM2M scenario considered in Appendix A.4, where the LwM2M Server acts as CoAP client, and it uses a LwM2M Gateway acting as a CoAP-to-CoAP reverse-proxy in order to reach an end IoT device.

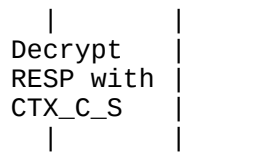
In order to determine where it has to forward an incoming request to, the proxy relies on the URI path components that are specified as value of the Uri-Path options included in the request. In particular, the proxy relies on the first URI path segment to identify the specific IoT device where to forward the request to, while the remaining URI path segments specify the target resource at the IoT device.

However, as shown in the example, the URI path segments that specify the target resource are hidden from the proxy, since they are protected by the additional use of OSCORE end-to-end between the client and the server.

Furthermore, this example assumes that, in the URIs identifying the target resource at the proxy as well as in the URI identifying the target resource at the server, the host component represents the destination IP address of the request as an IP-literal. Therefore, both the request from the client to the proxy and the request from the proxy to the server do not include a Uri-Host option (see Section 6.4 of [RFC7252]).


Client Proxy Server
Encrypt
REQ with
CTX_C_S
Encrypt
REQ with
CTX_C_P
POST
Code: 0.02 (POST)
Token: 0x8c
OSCORE: [kid:0x20, Partial IV:31]
0xff
Payload: {Code: 0.02 (POST),
OSCORE: [kid:0x5f, Partial IV:42],
Uri-Path: "dev1",
0xff,
{Code: 0.01 (GET),
Uri-Path: "alarm_status"
} // Encrypted with CTX_C_S
} // Encrypted with CTX_C_P
Decrypt
REQ with
CTX_C_P
Encrypt
REQ with
CTX_P_S
POST
Code: 0.02 (POST)
Token: 0x7b
OSCORE: [kid:0xd4, Partial IV:31]
0xff
Payload: {Code: 0.02 (POST),
OSCORE: [kid:0x5f, Partial IV:42],
0xff,
{Code: 0.01 (GET),
Uri-Path: "alarm_status"
} // Encrypted with CTX_C_S
} // Encrypted with CTX_P_S
Decrypt
REQ with
CTX_P_S
Decrypt


REQ with
CTX_C_S
Encrypt
RESP with
CTX_C_S
Encrypt
RESP with
CTX_P_S
Code: 2.04 (Changed)
2.04
Token: 0x7b
OSCORE: -
0xff
Payload: {Code: 2.04 (Changed),
OSCORE: -,
0xff,
{Code: 2.05 (Content),
0xff,
"0"
} // Encrypted with CTX_C_S
} // Encrypted with CTX_P_S
Decrypt
RESP with
CTX_P_S
Encrypt
RESP with
CTX_C_P
Code: 2.04 (Changed)
2.04
Token: 0x8c
OSCORE: -
0xff
Payload: {Code: 2.04 (Changed),
OSCORE: -,
0xff,
{Code: 2.05 (Content),
0xff,
"0"
} // Encrypted with CTX_C_S
} // Encrypted with CTX_C_P
Decrypt
RESP with
CTX_C_P

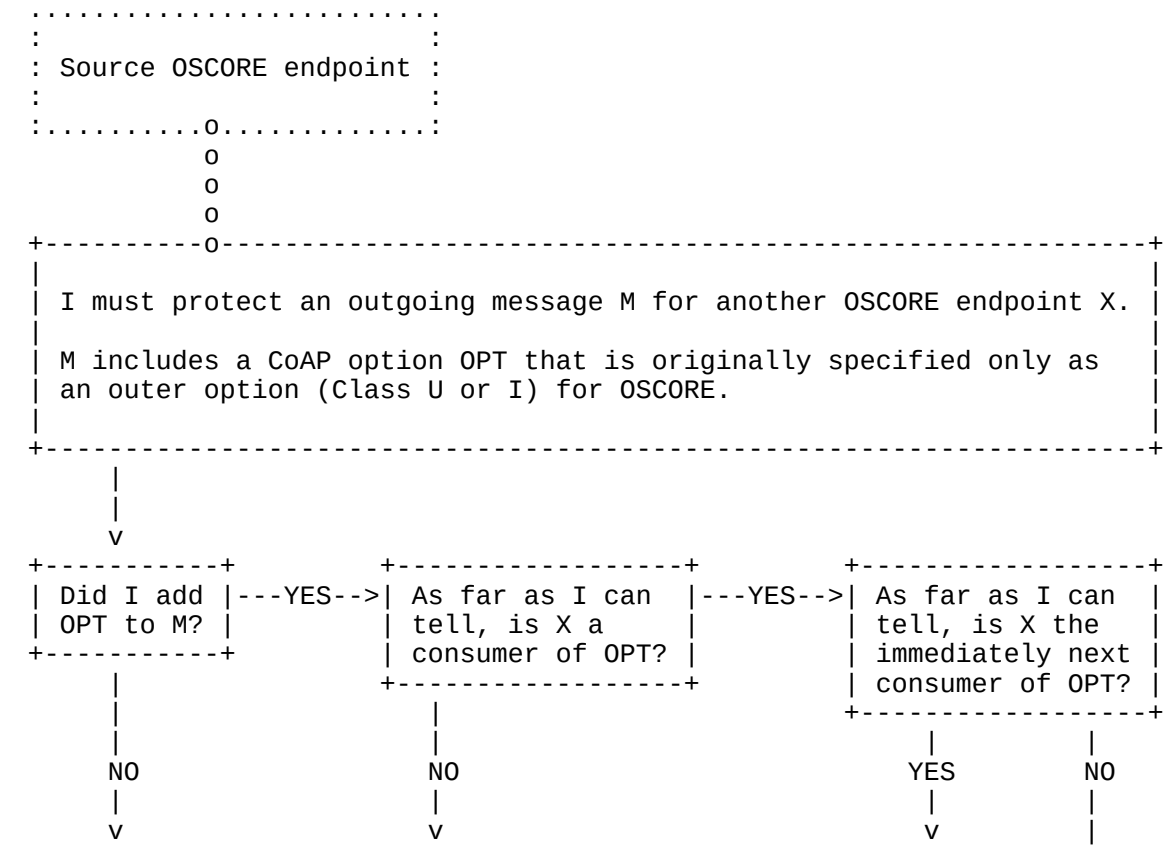


Square brackets [...] indicate content of compressed COSE object.
Curly brackets { ... } indicate encrypted data.

Figure 7: Use of OSCORE between Client-Proxy and Proxy-Server (the proxy is a reverse-proxy)

Appendix C. State Diagram: Protection of CoAP Options

Figure 8 overviews the rules defined in Section 2.2, to determine whether a CoAP option that is originally specified only as an outer option (Class U or I) for OSCORE has to be processed as Class E, when protecting an outgoing message.



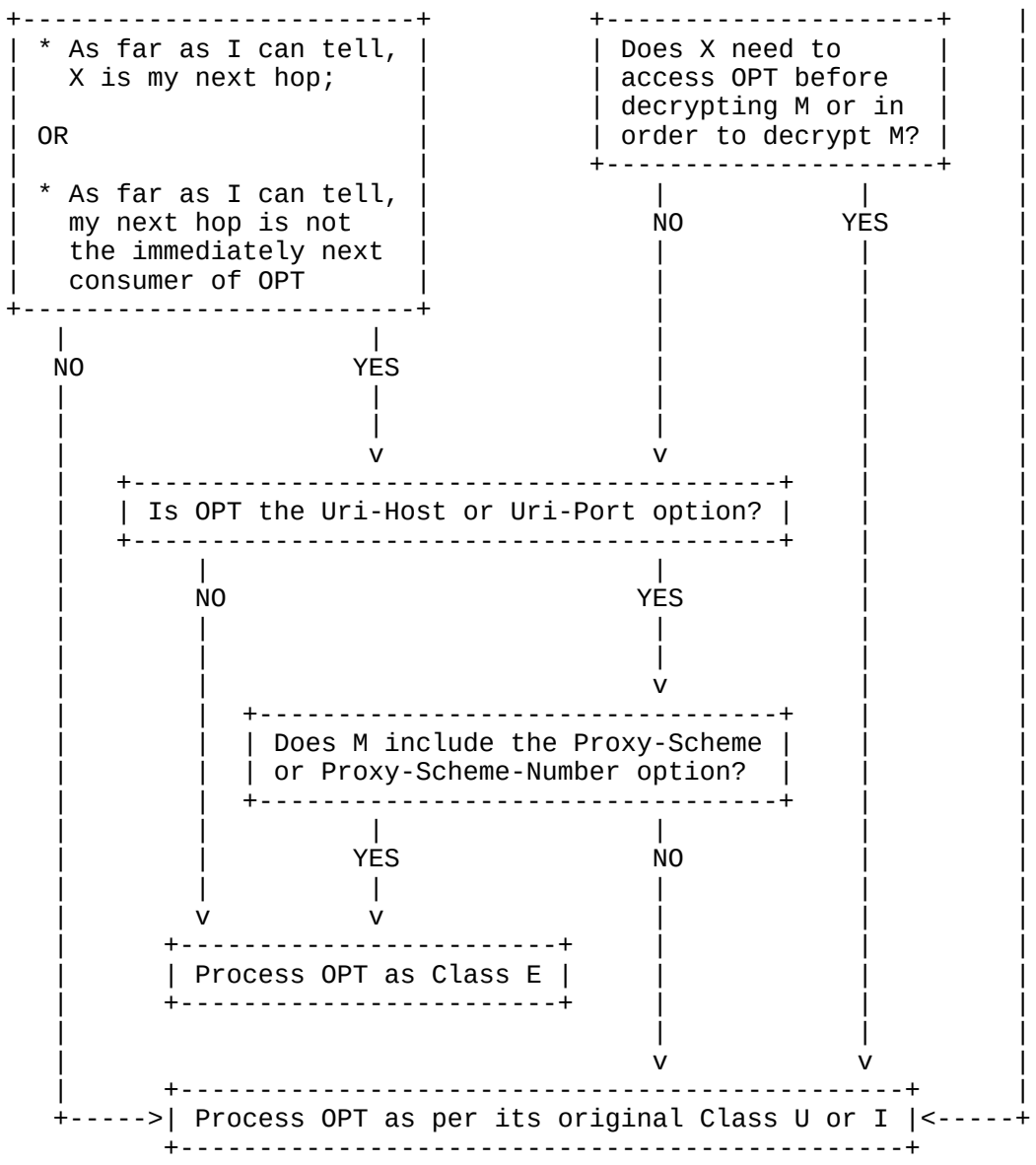
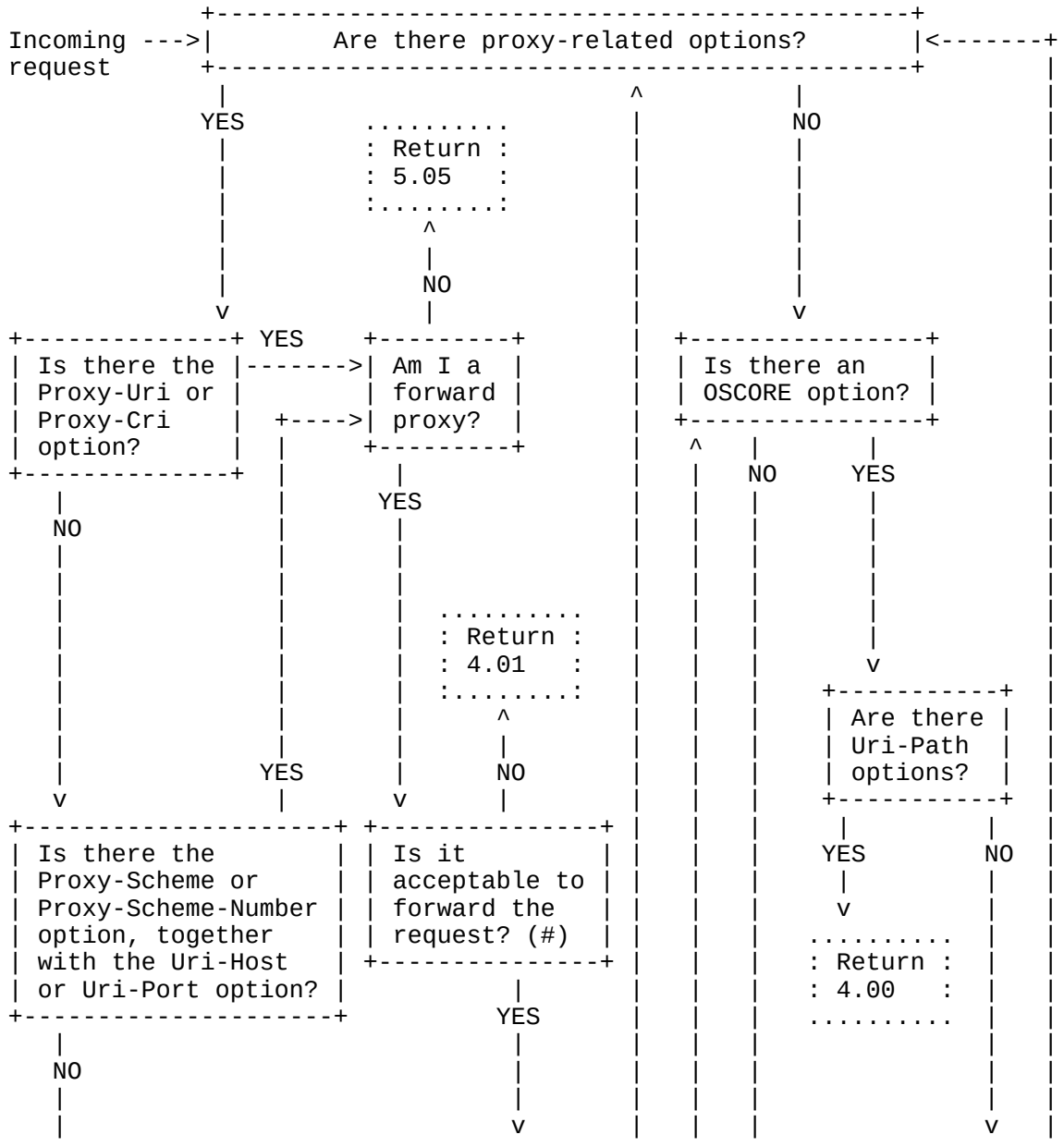
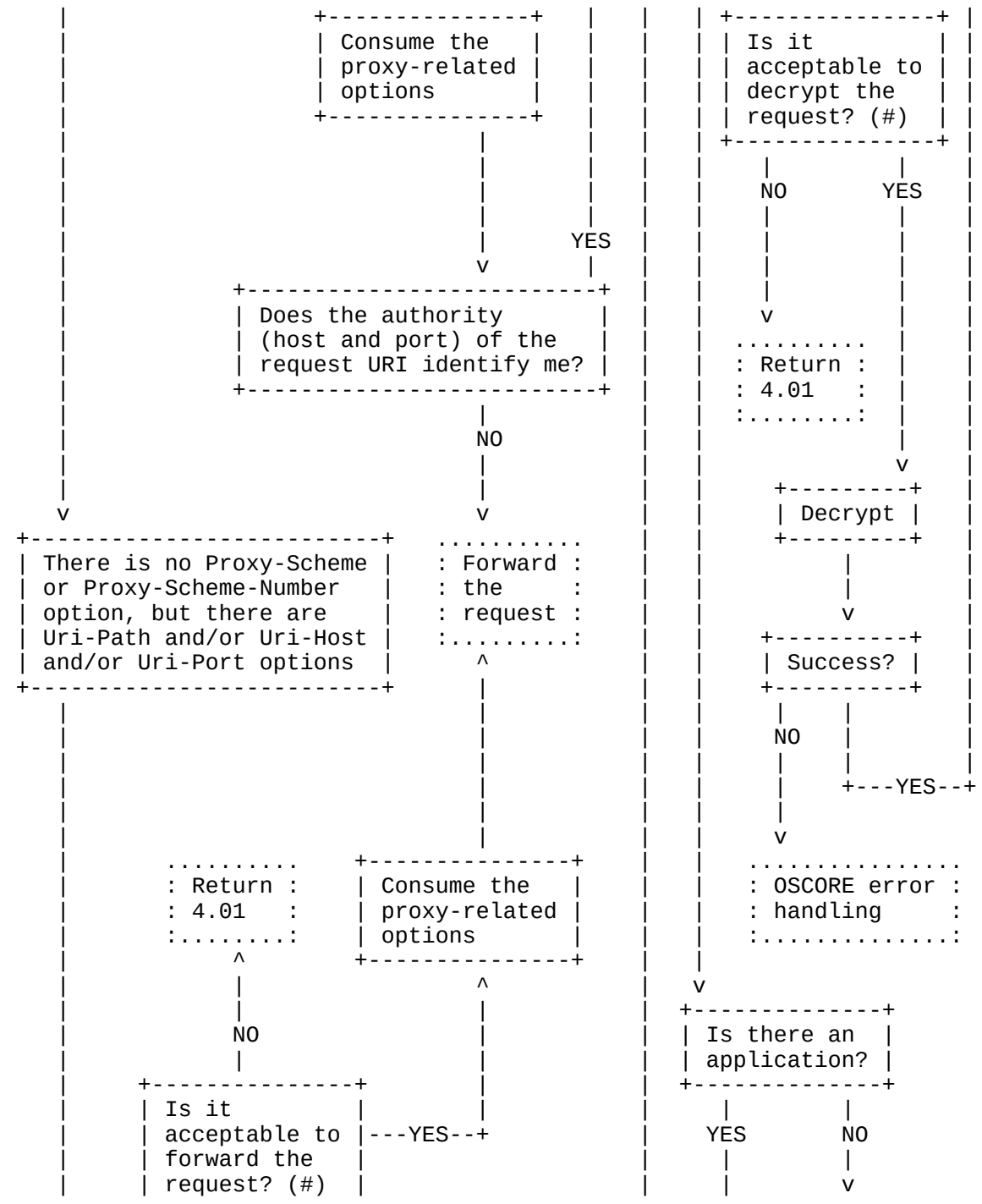


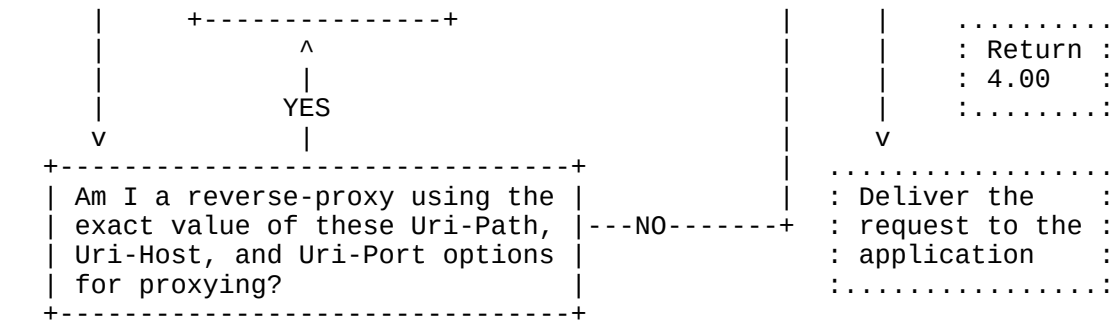
Figure 8: Protection of CoAP Options Originally Specified only as Outer Options (Class U or I) for OSCORE.

Appendix D. State Diagram: Processing of Incoming Requests

Figure 9 overviews the processing of an incoming request, as specified in Section 2.4. The dotted boxes indicate ending states where the processing terminates.




Consume the proxy-related options
YES
Does the authority (host and port) of the request URI identify me?
NO
There is no Proxy-Scheme or Proxy-Scheme-Number option, but there are Uri-Path and/or Uri-Host and/or Uri-Port options
Forward the request
Return 4.01
Consume the proxy-related options
Is it acceptable to forward the request? (#)
YES
NO
Is it acceptable to decrypt the request? (#)
NO
YES
Return 4.01
Decrypt
Success?
NO
YES
OSCORE error handling
Is there an application?
YES
NO



(#) This is determined according to the endpoint's configuration and a possible authorization enforcement.

Figure 9: Processing of an Incoming Request.

Appendix E. Document Updates

This section is to be removed before publishing as an RFC.

E.1. Version -03 to -04

* Removed definition and use of "OSCORE-in-OSCORE".
* Moved use cases to an appendix.
* Explain deviations from RFC 8613 as an actual subsection.
* More precise indication of outer or inner CoAP options.
* Added security consideration on membership of OSCORE groups.
* Updated references.
* Editorial improvements.

E.2. Version -02 to -03

* Clarified motivation for updating RFC 8768 in the introduction.
* Explained that OSCORE-capable proxies have to recognize CoAP options included in outgoing messages to protect.
* Fixed typo about the intended class of Hop-Limit option for OSCORE.

* Fixed protection of the Uri-Host option in examples.

* Added security considerations about the Hop-Limit option.

* Clarifications and editorial improvements.

E.3. Version -01 to -02

* Revised escalation of CoAP option protection.

* Specified general ordering for protecting outgoing requests.

* Explicit definition of OSCORE processing for the Hop-Limit option (update to RFC 8768).

* Added examples of message exchange with a reverse-proxy.

* Clarifications and editorial improvements.

E.4. Version -00 to -01

* Escalation of option protection as explicit update point to RFC 8613.

* Clarified examples of Class U/I CoAP options that become encrypted.

* Considered also the CoAP Options Proxy-Cri and Proxy-Scheme-Number.

* Added reference to Onion CoAP as use case.

* Required to set a limit on OSCORE layers that can be added/removed.

* Revised general rules on protecting CoAP options.

* A forward-proxy consumes a request when the request URI identifies the proxy itself.

* Consistency fix: a reverse-proxy can forward based on Uri-Host, Uri-Port or Uri-Path.

* Generalized authorization checks as acceptability checks.

* Added acceptability check before decrypting a request.

* Fixes in the examples of message exchange.

* Updated state diagram of the incoming request processing.

* Added state diagram on the protection of CoAP options of Class U/I.

* Updated references.

* Editorial fixes and improvements.

Acknowledgments

The authors sincerely thank Christian Amsss, Peter Blomqvist, Carsten Bormann, David Navarro, and Gran Selander for their comments and feedback.

The work on this document has been partly supported by the Sweden's Innovation Agency VINNOVA and the Celtic-Next projects CRITISEC and CYPRESS; and by the H2020 project SIFIS-Home (Grant agreement 952652).

Authors' Addresses

Marco Tiloca
RISE AB
Isafjordsgatan 22
SE-16440 Kista
Sweden
Email: marco.tiloca@ri.se

Rikard Hglund
RISE AB
Isafjordsgatan 22
SE-16440 Kista
Sweden
Email: rikard.hoglund@ri.se