Performance Evaluation of Group OSCORE for Secure Group Communication in the Internet of Things

MARTIN GUNNARSSON, RISE Cybersecurity, RISE Research Institutes of Sweden
KRZYSZTOF MATEUSZ MALARSKI, DTU Technical University of Denmark
RIKARD HÖGLUND, RISE Cybersecurity, RISE Research Institutes of Sweden, Sweden and Department of Information Technology, Uppsala University
MARCO TILOCA, RISE Cybersecurity, RISE Research Institutes of Sweden

The Constrained Application Protocol (CoAP) is a major application-layer protocol for the Internet of Things (IoT). The recently standardized security protocol Object Security for Constrained RESTful Environments (OSCORE) efficiently provides end-to-end security of CoAP messages at the application layer, also in the presence of untrusted intermediaries. At the same time, CoAP supports one-to-many communication, targeting use cases such as smart lighting and building automation, firmware update, or emergency broadcast. Securing group communication for CoAP has additional challenges. It can be done using the novel Group Object Security for Constrained RESTful Environments (Group OSCORE) security protocol, which fulfills the same security requirements of OSCORE in group communication environments. While evaluations of OSCORE are available, no studies exist on the performance of Group OSCORE on resource-constrained IoT devices.

This article presents the results of our extensive performance evaluation of Group OSCORE over two popular constrained IoT platforms, namely Zolertia Zoul and TI Simplelink. We have implemented Group OSCORE for the Contiki-NG operating system and made our implementation available as open source software. We compared Group OSCORE against unprotected CoAP as well as OSCORE. To the best of our knowledge, this is the first comprehensive and experimental evaluation of Group OSCORE over real constrained IoT devices.

CCS Concepts: • Security and privacy → Security protocols; • Networks → Network performance analysis; • Computer systems organization → Embedded and cyber-physical systems;

Additional Key Words and Phrases: End-to-end security, Group OSCORE, Internet of Things, group communication, Contiki-NG
1 INTRODUCTION

In recent years, we have been witnessing a massive Internet of Things (IoT) rollover, which is further accelerated under the umbrella of 5G deployment and is expected to result in 55.7 billions of connected devices in 2025 [32]. Thousands of resource-constrained sensors and actuators are being deployed in farms, factories, smart homes and buildings, as well as public spaces. The range of relevant applications encompasses environmental monitoring, automated building or infrastructure control, remote metering, teledicine, and many more. It goes without saying that ensuring efficient performance as well as fulfilling security requirements are of paramount importance.

For several applications, it is additionally convenient to rely on a group communication model, where a single transmitted message targets multiple devices, with all of them as intended recipients. This communication model especially suits some typical and renowned use cases, including but not limited to device and appliance control in smart buildings (e.g., smart lighting and door locks), discovery of resources and services in the network environment, distribution of software and firmware updates (e.g., functionality upgrades and vulnerability patches), as well as emergency broadcast. Evidently, interaction and message exchange among IoT devices have to be lightweight, robust and secure also for applications taking advantage of such a group communication model.

As of today, most notable application-layer transfer protocols used for the IoT are MQ Telemetry Transport (MQTT) [1] and the Constrained Application Protocol (CoAP) [67], both of which are available as open standards.

MQTT relies on a publish-subscribe models, where publisher clients can disseminate messages organized into topics among subscriber clients. This is mediated by a Broker server, as responsible for dispatching messages of a certain topic toward subscribers to that topic. MQTT mainly works over the Transport Layer Protocol (TCP) and it can enjoy secure communication between the clients and the broker by means of the Transport Layer Security (TLS) protocol suite [20].

CoAP builds on the REpresentational State Transfer (REST) model [57] and thus focuses on client-server interactions for manipulating and transferring the state of server resources. As a consequence, CoAP enables seamless cooperation with the ubiquitous and also REST-based HyperText Transfer Protocol (HTTP) [58] used in the Internet, while introducing a small and limited message overhead. CoAP runs mainly over User Datagram Protocol (UDP) and provides optional features such as the event-based resource Observation [39]. Also, CoAP natively supports deployed intermediaries such as proxies, which typically provide additional services such as caching of responses and translation of underlying transport protocol. Hereafter, we focus on CoAP as the specific target of the security protocol evaluated in this article.

As to secure communication, CoAP initially considered only the Datagram Transport Layer Security (DTLS) [21] protocol suite to protect exchanged messages at the transport layer. Recently, the protocol Object Security for Constrained RESTful Environments (OSCORE) has been published as RFC 8613 [30], and it enables the end-to-end protection of CoAP messages at the application layer, independent of the underlying transport protocol and preserving messages mostly opaque to possible (untrusted) intermediaries. OSCORE has attracted attention from the industry [22, 63] and is the application-layer security solution used in the IoT device management standard Open Mobile Alliance Lightweight Machine-to-Machine [6, 55].
Furthermore, CoAP natively supports one-to-many group communication scenarios, where a single instance of a message intended to multiple recipients is dispatched to all of them at once, e.g., using Internet Protocol (IP) over multicast [5, 19]. This makes CoAP particularly suitable to serve the use cases mentioned above. However, with respect to security, group communication introduces additional challenges and especially DTLS cannot be used to provide message protection in group communication scenarios. To fill this gap, the new security protocol Group Object Security for Constrained RESTful Environments (Group OSCORE) [49] has been proposed and is under ongoing standardization in the Internet Engineering Task Force (IETF). In particular, Group OSCORE extends and adapts OSCORE to work in group communication scenarios, while fulfilling the same security requirements. That is, Group OSCORE protects CoAP messages end-to-end at the application layer and provides replay detection, confidentiality, integrity, and source authentication of exchanged messages. In particular, the group mode of Group OSCORE that this article focuses on provides source authentication of messages by means of digital signatures. The communication model considers protected request messages sent to multiple recipients at once, as well as the corresponding protected, cryptographically bound multiple responses. The design and development of Group OSCORE are also actively supported by the industry [22].

Even though Group OSCORE is suitable for applications using CoAP in group communication environments, it clearly requires to be tested on commercially available IoT hardware platforms, on which its performance and impact ought to be assessed. However, while evaluations of OSCORE have been carried out and made available [44], no studies have been made so far on the feasibility and performance of Group OSCORE on resource-constrained IoT devices. Note that, while the Group OSCORE specification at Reference [49] focuses on the theoretical protocol design and practical considerations, it does not refer to any particular implementation nor does it provide an experimental evaluation.

This raises the following questions: How does an implementation of Group OSCORE perform in a setup of real constrained IoT devices relying on CoAP group communication over IP multicast? How does it quantitatively impact their performance, especially due to the use of digital signatures? How is this in turn affected by the use of different signature algorithms? Ultimately, is it a feasible choice to implement and use in a scenario with constrained devices relying on group communication?

In this article, we address these questions and especially fill the lack of performance assessments by providing an experimental performance evaluation of the Group OSCORE security protocol on two popular constrained IoT platforms. Specifically, we implemented Group OSCORE in the Contiki-NG operating system [16], and made our implementation available as open source software.¹ Then, we used our implementation to perform experiments on real low-power IoT devices, considering the two platforms Zolertia Firefly Rev.A board (Zoul) [68] with the CC2538 System on a Chip (SoC) and TI Simplelink with the CC1352R1 SoC [62].

We considered a setup consisting of a constrained CoAP client exchanging messages with three constrained CoAP servers. Both one-to-one and one-to-many unprotected communications, as well as one-to-one communications protected with OSCORE and one-to-many communications protected with Group OSCORE were evaluated. Similarly, when OSCORE or Group OSCORE were evaluated, cryptographic operations were performed either in software or through hardware acceleration. Specifically for Group OSCORE, we considered the two different signature algorithms Edwards-curve Digital Signature Algorithm (EdDSA) P-256 and Elliptic Curve Digital Signature Algorithm (ECDSA) 25519. We experimentally evaluated performance and resource utilization on the server side for both IoT platforms, in terms of Random Access Memory (RAM)/Read-Only Memory (ROM), Central Processing Unit (CPU) and energy consumption.

Furthermore, we investigated the **Round Trip Time (RTT)** as perceived by the client during the message exchanges.

As key takeaways from our evaluation, we can show that the memory utilization in terms of RAM and ROM is manageable for constrained IoT devices. The use of **Elliptic Curve Cryptography (ECC)** signatures adds a non-negligible delay to a full message exchange and significantly contributes to the energy consumption. However, the EdDSA signature algorithm displays considerably better performance than ECDSA P-256. Even though the complexity of ECC cryptography adds such penalties, Group OSCORE remains feasible for low-power applications.

We can summarize the key contributions of this article as follows.

- Our implementation of Group OSCORE in Contiki-NG, which is to the best of our knowledge the first open source and publicly available implementation of Group OSCORE suitable for constrained IoT platforms.
- An extensive performance evaluation of Group OSCORE on real IoT devices of two commercially-available hardware platforms, based on our open source implementation for Contiki-NG. To the best of our knowledge, this is the first publicly available experimental evaluation of Group OSCORE over commercial IoT devices.

The rest of this article is organized as follows. Section 2 discusses the related work. Section 3 overviews relevant use cases relying on group communication and related security requirements. Section 4 introduces relevant background technologies and concepts, while Section 5 provides a high-level description of the Group OSCORE protocol. Section 6 describes the methodology and the setup considered in our performance evaluation, while in Section 7 we present and discuss our experimental results. Finally, Section 8 draws our conclusive remarks and highlights possible future work.

## 2 RELATED WORK

Previous works have considered secure group communication, enforced at different layers in the communication stack. In 2017, Reference [60] characterized typical security properties desired in secure group communication, covering message protection and main aspects of group management, and surveyed some known approaches. This section focuses on message protection in group communication and overviews the main current approaches in a bottom-up fashion. The specific layer enforcing security may depend on the device capabilities and the requirements of the application scenario.

At the data link layer, the IEEE 802.15.4 Standard [33] can also be used to provide secure group communication, by using a single group cryptographic key commonly shared among all network nodes. For most of the resource-constrained platforms commercially available, the actual cryptographic operations are typically available in hardware. Securing communications at the data link layer presents a number of limitations. First, link layer frames are protected hop by hop, thus they have to be decrypted and re-encrypted at each network node on the path between the original sender and the final recipient. This considerably affects network performance and message delivery time, especially in multi-hop setups, and yields a non-negligible overhead in terms of energy consumption [56]. Second, like for other link layer protocols, IEEE 802.15.4 does not consider the establishment and management of cryptographic keys and blindly entrusts this to the higher layers in the stack [33]. Third, as using only a common group key to protect group communication, it is not possible to ensure source authentication of messages exchanged in the group.

At the network layer, security can be provided by the protocol suite IPsec [7, 41], with security associations established through the IKEv2 protocol [15]. The IPsec suite has been adapted to work in group communication scenarios [26], and an analogous adaptation of IKEv2 has been
defined [40, 65]. To foster the use of IPsec also for constrained IoT devices, additional work has been done. This includes an extension of 6LowPAN to provide header compression also for IPsec [59], a further adaptation of the IKEv2 protocol, i.e., Minimal-IKEv2 [61] and Diet-ESP [18], i.e., an adaptation of the IPsec protocol Encapsulated Security Payload (ESP). In particular, by leveraging an ESP Header Compression strategy, Diet-ESP achieves an overhead that is limited and smaller than the one of ESP. As noted in Reference [60], such approaches used in group communication scenarios do not provide source authentication of messages but only confidentiality and group authentication. More generally, source authentication of exchanged IPsec packets can be enforced again by embedding digital signatures computed with the RSA algorithm, as defined in Reference [9]. However, this has not been defined for more recent, efficient, and secure signature algorithms (e.g., ECDSA and EdDSA) and the responsible Working Group MSEC\(^2\) of the IETF terminated its activities.

The work in References [51, 53] proposed an approach to secure group communication for the IoT at the transport layer, through an adaptation of the DTLS protocol suite [21], which originally supports only one-to-one communication among two peers. This approach relies on a pre-established Group Security Association, including common key material shared among the group members. Building on that, sender nodes use the common group key material to protect one-to-many multicast request messages addressed to the other group members, which can individually reply back. Similarly to other techniques above, this approach relies only on shared cryptographic material, hence it does not provide source authentication of exchanged messages.

In Reference [8], the authors proposed a similar DTLS-based approach for group communication, with additional focus on key provisioning. This relies on a centralized Group Controller that, upon the joining of new group members, provides those with a common group key for protecting request messages sent over IP multicast. When receiving a multicast request from a certain sender in the group for the first time, a listener asks the Group Controller for a pairwise key to share with that sender and to protect unicast responses sent to that sender. When receiving a response from a certain listener for the first time, a sender asks the Group Controller for the same pairwise key. Also in this case, since requests sent over multicast are protected with the same common group key, their source authentication is not ensured.

In the presence of intermediaries such as proxies, these DTLS-based approaches do not provide end-to-end security between the original data producer and the final data consumer(s) in the group. In fact, to enable proxy operations on CoAP messages, a DTLS session at a sender node would have to terminate at the proxy [28, 30], which has to use a separate DTLS session with the final recipient(s). This both worsens performance due to the double security processing at the proxy [44], and requires to trust the proxy beyond what minimally required in the application scenario.

The Group OSCORE protocol [49] evaluated in this article provides an efficient and lightweight solution for secure group communication. In particular, Group OSCORE operates at the application layer, although not within the actual application. As explained more in detail in Section 5, Group OSCORE transforms an unprotected CoAP message into an equivalent protected CoAP message, which allows to use Group OSCORE over any one-to-many transport where CoAP works. As a result, Group OSCORE fills the gaps and limitations of the approaches above, i.e., it ensures end-to-end security between the original data producer and the final data consumer(s), also in the presence of (untrusted) intermediaries; it ensures source authentication of exchanged CoAP messages, through either digital signatures or pairwise symmetric keys derived from pairwise Diffie-Hellman secrets; it ensures crypto agility and extensibility in the space of encryption and signature algorithms, by using the standard CBOR Object Signing and Encryption

and it can seamlessly rely on the off-the-shelf approach for group joining and key provisioning defined in Reference [50], in turn based on the ACE framework for authentication and authorization in constrained environments [43].

Alternative approaches for group communication leverage multi-party content dissemination, rather than message delivery on actual one-to-many links over multicast. This notably includes the open standard MQTT [1] based on the publish-subscribe model, where publisher clients can send messages for a certain topic to a Broker server, which in turn forwards those messages to the subscriber clients that have registered their interest for that topic. MQTT mainly works over TCP, and communications between the Broker and each client can be secured with the TLS protocol suite [20]. The work in Reference [31] proposed an extension to provide end-to-end protection of content exchanged between the clients. That is, the Broker provides each registered client with a pairwise session key, as well with a per-topic group key encrypted with that session key. As all the clients interested in a topic share the same group key, a publisher (subscriber) client can encrypt (decrypt) published content by using that group key, thus the Broker does not perform a decryption and re-encryption process.

It is possible to use CoAP to set up an application based on the publish-subscribe model, as defined in Reference [45]. In such a case, a CoAP server acts as Broker, while the CoAP clients acting as subscribers use CoAP Observe [39] on the Broker’s resources associated to their topics of interest. By doing so, the subscribers receive unsolicited notifications from the Broker when the topic resources change their content, following a new publication on such topics from the CoAP clients acting as publishers. Secure communications can be provided between the Broker and each client as usually in CoAP, i.e., by using OSCORE and/or DTLS. Besides, Reference [52] defines how to further extend this model, by enabling the Broker to send a single notification to all the subscribers over multicast. These notifications can be protected using Group OSCORE, having all the clients and the Broker configured as members of the same group.

Lizardo et al. proposed Sharelock [2], a security protocol that provides end-to-end security and confidentiality of messages exchanged by groups of communicating nodes. In particular, the group members include different clusters of IoT devices behind untrusted Edge Servers used as communication intermediaries, as well as Cloud Servers as intended recipients and consumers of data sent by the IoT devices. Also, the group members are directly responsible for managing and establishing keys in the group, as well as for managing group memberships and consequent update of group key material.

In Reference [13], the authors considered multi-party dissemination of content based on the information-centric Named-Data Networking (NDN) architecture [64]. In particular, the study used CoAP in an NDN-based setup, where clients retrieve the content of interest from the origin server only as a last resort, while greatly relying on that content being available at a closer intermediary, that has cached previous responses conveying that content. If secure communication is “naively” introduced with OSCORE, then the nature of OSCORE limits the benefits of caching to exact request-response pairs, thus helping only in case of request retransmission. However, Reference [13] has additionally considered the particular use of Group OSCORE proposed in Reference [10] and shown that it effectively enables full-fledged caching of OSCORE-protected responses to a same deterministic request for the same content, thus considerably reducing content retrieval time.

Finally, regardless the underlying message transport, a number of stand-alone cryptographic schemes may also be supportive of group communication, although they fulfill only some of the expected security requirements. One of these schemes is \(\mu\)Tesla [4], i.e., an adaptation for constrained devices of the Tesla scheme [3] originally designed for streaming applications. While it relies on symmetric keys to provide authentication and thus displays very efficient performance, \(\mu\)Tesla provides neither source authentication nor confidentiality. Further schemes include Identity-Based
Signature [35] and Attribute-Based Encryption [66], which, however, provide only source authentication and integrity without confidentiality and only confidentiality without authentication and integrity, respectively.

3 RELEVANT USE CASES

This section provides a high-level overview of most relevant use cases that benefit of one-to-many, group communication. In addition, Section 3.1 lists a number of security requirements that such use cases expect to be fulfilled by the adopted security solutions and protocols. Most relevant use cases can be roughly organized into the following different categories, for which we provide some of the most representative examples.

- **Device control in group settings.** A first simple, yet effective and recurring use case concerns the control of lighting appliances. For example, a group can include lighting devices (acting as servers) and switch devices (acting as clients) deployed in a same room, corridor, floor, or open environment. Switch devices can then be used to control the lighting devices by sending a single on/off/dimming command at once to all the lighting devices in the group, e.g., over IP multicast or other one-to-many technologies. Connectivity between lighting devices may be achieved, for instance, by means of IPv6 and (border) routers supporting 6LoWPAN.

  A more advanced use case concerns the integrated control of smart buildings. While leveraging the same principles, this makes it possible to efficiently check and control the operational status of multiple types of appliances, such as physical sensors, heating, ventilation and air-conditioning units, or locks of doors and windows. Furthermore, groups can be configured to reflect not only the devices’ physical positioning but also their types and capabilities. Controlled devices may respond providing the result of the operation as well as their current operational status. As additional support, border routers connected to an IP network backbone (which is also multicast-enabled) can be used to interconnect routers in the building with each other, hence covering larger environments spanning over multiple floors.

  In Reference [23], the Fairhair Alliance described its Security Architecture for IoT-based building automation, which especially relies on the Group OSCORE protocol considered in this article for securing CoAP group communication.

- **Discovery of resources and services.** Group communication allows to conveniently reach out multiple devices at once to discover their precise physical positioning as well as their hosted particular resources or services, possibly based on filter criteria, such as ID, name, protocol, version, and type. To this end, groups can be configured to specifically reflect that their members are devices with similar capabilities and features or a common physical location. Queried devices may reply back to notify about their presence and provide the requested set of information as well as their current operational status.

- **Software update.** Instead of sending software updates separately to each individual device, group communication enables the efficient delivery of common updated data to a large set of devices at once, with benefits in terms of performance. That is, it yields a reduced network load and overall time latency for providing the data to all the intended recipient devices. The software update can consist in an application component, a firmware image or patch, or a set of parameter values for a single common configuration update. Devices receiving software updates typically reply, to provide feedback on the result of the update operation and their current operational status.

- **Emergency multicast.** Particular situations, such as a safety crisis or a natural disaster, require a notifier to quickly broadcast emergency related information to multiple devices as
part of a wide audience. The recipient devices may reply back, providing feedback and local information concerning the ongoing emergency.

3.1 Security Requirements

Typical applications relying on group communication have the following security requirements and expect them to be fulfilled by the used security solutions and protocols. As per Section 4, Group OSCORE fulfills all these requirements.

- **Data confidentiality.** Group-level data confidentiality must be ensured for all messages exchanged in a given group. This is achieved by encrypting messages at a group level, through security material shared by the members of that group. As a result, a message can be decrypted by any member of the group but not by an external adversary or other external entities. This must apply both to one-to-many request messages as well as to the corresponding multiple response messages. Group OSCORE provides this when used in Group Mode.

  Some security protocol may additionally provide pairwise data confidentiality for messages exchanged in a one-to-one fashion within the group, i.e., as sent to a single group member rather than to the whole group. This can be achieved by encrypting such messages using pairwise security material, which is shared only between two group members. As a consequence, only the intended single recipient is able to correctly decrypt the message. Group OSCORE provides this when used in Pairwise Mode.

- **Data replay protection.** It must be possible for a group member to detect whether an incoming message exchanged in the group has been replayed.

- **Source message authentication.** It must be possible for a group member to verify that an incoming message was indeed originated by the specific group member identified as alleged sender. Group OSCORE ensures this by using digital signatures appended to messages, when used in Group Mode; or by encrypting messages with a pairwise key derived from two group members’ asymmetric key material, when used in Pairwise Mode.

- **Message integrity.** It must be possible to ensure that a message sent in the group has not been tampered with while in transit by another group member, an external adversary, or any other external entity that is not a group member. This is practically achieved by the same means used to ensure message authentication.

- **Message ordering.** A group member must be able to determine the ordering of incoming messages from each different sender in the group. In this respect, Group OSCORE especially ensures absolute freshness of response messages that are not notifications [39] and relative freshness of request messages and notification responses. However, it is typically not required to determine the ordering of messages from different senders.

Fulfilling the requirements above concerns the actual secure communication protocol used to protect messages exchanged in the group, such as Group OSCORE. Furthermore, practical applications also rely on additional mechanisms for provisioning cryptographic keys and other security material to the group members, both upon their joining the group or later if needed during their operation. Typically, this can be achieved through a logically centralized Key Distribution Center, which maintains a dedicated secure communication association with each group member, for each of the groups it is responsible for. A Key Distribution Center suitable for groups where the Group OSCORE protocol is used has been proposed in Reference [50] and is under ongoing standardization.

Finally, applications may require that the security material used in the group is revoked and a new one is distributed upon a change in the group membership. That is, renewing the security material upon a new member’s joining will ensure backward secrecy, since the new member would
not be able to access messages exchanged in the group before its joining (even if it recorded them). Also, renewing the security material upon a member’s leaving (e.g., if compromised or suspected so) will ensure forward secrecy, since the leaving member would not be able to access messages exchanged in the group after its leaving.

The Group OSCORE protocol specifically requires that the security material in the group is renewed in case of a member’s leaving. Nevertheless, Group OSCORE is agnostic of the particular approaches and key management scheme used to renew the group security material and fulfill the related requirements discussed above. Further details on the secure, efficient revocation and re-distribution of security material in the group is out of the scope of this article.

4 TECHNICAL BACKGROUND

In this section, we introduce background technical concepts referred throughout the rest of the article.

4.1 CoAP

CoAP is a lightweight application-layer protocol, intended to support applications for constrained networks and with the aim of integrating massive IoT in the existing Internet infrastructure [67]. Since the contemporary Internet services (web-browsing, online trade and banking, to name a few) are based on the HTTP/REST architecture [57, 58], the design of CoAP follows the same principles, being HTTP/REST-compliant and providing analogous operation methods such as GET, PUT, POST, and DELETE.

Furthermore, CoAP was first designed to operate over unreliable transport, especially the UDP protocol [36], and was later extended to possibly operate over reliable transports [12], such as the TCP protocol [34]. CoAP enables synchronous and asynchronous communications and considers low-power IoT deployments with multiple years on battery expected. In particular, it supports messages as small as 4 bytes\(^3\) and can work seamlessly with network proxies or caches, which can be used to offload the constrained devices or schedule the message delivery so that devices can remain in sleep, power-saving mode for longer time.

As shown in Figure 1, a CoAP message consists of some mandatory header fields (coloured black) and optional additional elements (coloured gray). The header specifies the release of CoAP in use (Version); the type of the message (i.e., Confirmable, Non-Confirmable, Acknowledgement or Reset); the length of the Token, which correlates a response to a request; the CoAP Method Code indicating the Request Method in request messages (e.g., GET, PUT, etc.) or the Response Code in response messages; and the Message ID, which correlates an Acknowledgment to a Confirmable request/response and allows perform message deduplication. None or more CoAP options with different sizes may follow to provide additional information regarding the communication as well.

\(^3\)For the sake of comparison: only identifying the protocol version (e.g., “HTTP/1.1”) already takes 8 bytes in a HTTP message.
as the message encoding and expected handling. Finally, should application data be present as message payload, it needs to be prepended by a 1-byte marker with all bits set to one.

4.1.1 Group Communications. As CoAP is suitable for the IP stack, it is possible to transmit the requests over UDP [36] and IP multicast [19], thus decreasing the amount of data to be transmitted to deliver the same payload to multiple recipients (CoAP servers). Instead of addressing the request to a specific endpoint, the CoAP client sends the request message toward the multicast IP address of the group. The server devices that registered for listening to the group address can respond to the client in a unicast fashion. All CoAP request messages sent over IP multicast can only be Non-Confirmable (i.e., with no Acknowledgement expected). The servers must delay their responses to the client by a randomized value, namely leisure time, which must be equal to or longer than:

\[
\text{leisure}_{\text{min}} = S \times G/R,
\]

where \( S \) corresponds to the message size, \( G \) denotes the group size, and \( R \) symbolises the transmission data rate. Such an approach reduces the chances of simultaneous transmissions from the servers in the group, which would result in colliding responses and possible network congestion. The risk of network congestion may also be controlled by setting the servers as silent nodes that never reply with a response and/or by suppressing certain responses (e.g., error responses of a certain error class).

Rather than IP multicast, applications that want to use CoAP for group communication can rely on alternative technologies providing one-to-many message delivery. For example, a CoAP request message addressed to a group of devices can be directly transported as payload of an IEEE 802.15.4 broadcast frame [33]. In the rest of this article, we focus on group communication for CoAP using UDP and IP multicast as per Reference [19].

4.2 OSCORE

OSCORE [30] is a recently standardized application-layer security protocol that, unlike DTLS, provides end-to-end security between the original data producer and the final data consumer. Instead of protecting the whole communication channel between the client and the server, the protocol is “CoAP-aware” and encrypts only the parts of the CoAP message that require confidentiality, while the fields meant to be used by proxies are left unprotected or only integrity protected. In particular, OSCORE provides end-to-end encryption, integrity protection, source authentication, and replay protection of messages, while displaying smaller power consumption and memory burden on constrained devices when compared to DTLS [44].

At a high-level, OSCORE takes a CoAP message as input and produces as output a new protected CoAP message, namely an OSCORE message. The reverse process occurs when an OSCORE message is received, and the original CoAP message is recomputed. Furthermore, OSCORE is independent of the specific transport layer, and it works wherever CoAP works. Also, it is possible to combine OSCORE with communication security on other layers, e.g., to further protect an OSCORE-protected message using DTLS.

The lightweight design of OSCORE leverages the efficient and small-size encoding scheme Concise Binary Object Representation (CBOR) [11]. That is, the data to be protected compose a CBOR structure, which is then encrypted and authenticated by using COSE [37, 38], thus yielding a COSE object that is transported in the protected CoAP message. Therefore, OSCORE follows the object security paradigm, where each data chunk is secured separately.

Before they can exchange secure data, two CoAP nodes need to establish an OSCORE Security Context. However, as focused only on message protection, OSCORE itself does not provide a

\[^{4}\text{Note that the server’s IP address is the source address of the response.}\]
mechanism to do so, i.e., one equivalent to the Handshake protocol of the DTLS suite. However, a number of approaches have been developed to let two CoAP nodes establish an OSCORE Security Context. These include, for instance, the lightweight key establishment protocol Ephemeral Diffie-Hellman Over COSE [29], which is currently an IETF standardization proposal.

An OSCORE Security Context contains three parts, i.e., a Common Context, a Sender Context, and a Recipient Context. The Common Context is identical for both nodes and specifies the HMAC-based key Derivation Function and **Authenticated Encryption with Associated Data (AEAD)** algorithms, as well as information used to identify the Security Context and to derive the cryptographic keys and nonces (i.e., Context ID, Common IV, Master Secret, and Master Salt). Each of the two nodes uses its Sender Context to protect outgoing messages addressed to the other peer, and its Recipient Context to decrypt incoming messages from the other peer. Given a certain node, its Sender (Recipient) Context contains the Sender ID of that node (the Recipient ID that the other node uses as its own Sender ID), the symmetric key used to encrypt (decrypt) outgoing (incoming) messages to (from) the other peer, and a sender sequence number (a replay window). Evidently, the Sender Context of one node mirrors the Recipient Context of the other node. Furthermore, a pair of nodes must ensure that their Context ID, Sender ID, Master Secret, and Master Salt (thus also the derived symmetric keys and AEAD nonces) are unique; otherwise, the security features of OSCORE cannot be guaranteed. Note that each of the two nodes can act as only CoAP client, only CoAP server, or both.

Figure 2 shows the format of an OSCORE-protected CoAP message. The procedure that produces such a message as output can be summarized as follows. A more detailed description is available in the OSCORE specification [30].

1. All the CoAP options that are not needed for proxying, the original CoAP Method Code and the original CoAP payload (if any) are considered as plaintext to be encrypted.
2. The CoAP options to be only integrity protected, the “Version” field of the CoAP header and additional OSCORE-related information compose the Additional Authenticated Data (Additional Authenticated Data (AAD)), to be integrity protected only.
3. The plaintext and the AAD are included in a new COSE object.
4. The agreed cryptographic algorithm is used to encrypt and authenticate the COSE object by taking as input the plaintext and AAD, the Sender Key, and a nonce derived from the Sender Sequence Number and other information from the Security Context. Note that a response may be protected by reusing the same nonce of the request; this spares the server from using its own fresh Sender Sequence Number and from including it in the response, thus yielding a smaller message as result.
5. The resulting ciphertext of the COSE object and its AEAD-tag become the payload of the OSCORE-protected message.
6. The OSCORE-protected messages is finalized by (i) adding an OSCORE option, with information that enables the recipient to decrypt the message, and (ii) replacing the original CoAP

![Fig. 2. Format of an OSCORE-protected CoAP message.](image)
Method Code in the CoAP header to indicate the POST method (or FETCH if CoAP Observe is used [39]) in request messages and the Response Code 2.04 “Changed” (or 2.05 “Content,” if CoAP Observe is used) in response messages.

The recipient of the OSCORE-protected message decrypts and authenticates the ciphertext using the Recipient Key from its Recipient Context. Also, for request messages, it checks the Sequence Number conveyed in the OSCORE option against its Replay Window to enforce replay protection.

5 GROUP OSCORE

In our previous work [44], we showed that the OSCORE protocol has performance advantages over DTLS in terms of energy consumption and communication latency on top of its pivotal end-to-end communication security in the presence of intermediaries. However, the applicability and security guarantees of OSCORE are limited only to unicast communications, where messages are exchanged strictly between two CoAP nodes.

As a step forward to provide security in use cases relying on group communication (see Section 3), new work started for adapting OSCORE to group communication environments (see Section 4.1.1). This resulted in the ongoing standardization activities around the Group OSCORE protocol [49]. This provides the same security properties of OSCORE, while protecting a one-to-many CoAP requests addressed to multiple servers at once, as well as the multiple corresponding CoAP responses. At the time of writing, Group OSCORE is the only available method to protect group communication based on CoAP and is the mandatory security solution for CoAP over UDP and IP multicast [19].

The rest of this section overviews Group OSCORE, presenting its main features as much as possible in comparison with OSCORE. A more detailed description is available in the Group OSCORE specification draft [49].

5.1 General Properties

Group OSCORE extends and adapts OSCORE to work also in group communication scenarios. In particular, Group OSCORE provides end-to-end security of CoAP messages exchanged between members of a group, e.g., using IP multicast.

Group OSCORE ensures cryptographic binding between a CoAP group request, sent by a client to multiple servers, and the corresponding CoAP responses individually sent by the servers in the group. Since message protection builds on commonly shared, group keying material, source authentication of messages exchanged in the group is achieved by using asymmetric keying material. As explained below, this relies on either digital signatures when using the group mode or on pairwise keys derived from asymmetric, individual keying material when using the pairwise mode.

Like OSCORE, Group OSCORE is independent of the specific transport layer, and it works wherever CoAP works. Also, like with OSCORE, it is possible to combine Group OSCORE with communication security on other layers. One example is the additional use of DTLS between one client and one proxy (and vice versa) or between one proxy and one server (and vice versa) to secure and hide from external observers also information left unprotected by OSCORE, such as CoAP options intended for intermediary proxies to perform message forwarding. Note that DTLS cannot be used to secure messages sent over IP multicast or other one-to-many message delivery technologies.

Group OSCORE has two different modes of operation, as different ways to protect CoAP messages. It is up to the application to decide in which particular mode a message has to be protected, possibly on a per-message basis.

- In the group mode, a message is encrypted with symmetric keying material available to all group members and includes an additional signature computed by using the private key of
In the pairwise mode, two group members can exchange unicast messages, as protected only with pairwise symmetric keys and not including a signature. These symmetric keys are derived from Diffie–Hellman shared secrets, calculated with the asymmetric keys of the two group members. Since a signature is not included, this results in a smaller message overhead. This method is applicable to one-to-one messages sent in the group, i.e., responses as individually sent by servers and requests addressed to one single server.

The pairwise mode displays minimal differences from the original OSCORE processing, i.e., the used symmetric keys are derived from the asymmetric keys of the two group members; it uses the same extended AAD defined for the group mode of Group OSCORE (see Section 5.3.2); the inclusion of the "kid" and "kid_context" parameters in the OSCORE option works as in the group mode of Group OSCORE (see Section 5.3.2). The pairwise mode allows a large number of endpoints in the same group to perform OSCORE-protected pairwise communication with one another, while keeping the related key provisioning effort and overhead small, as it is limited to interactions with the responsible Group Manager and it mostly occurs when joining the OSCORE group (see Section 5.2).

Like OSCORE, Group OSCORE provides message binding of responses to requests, which in turn provides relative freshness of responses and replay protection of requests. In particular, Group OSCORE fulfills the following security objectives: data replay protection, source authentication, message integrity, group-level data confidentiality (in group mode) or pairwise data confidentiality (in pairwise mode), and proof of group membership with respect to a message sender.

5.2 The Group Manager

Group OSCORE relies on the presence of an additional trusted entity acting as Group Manager. This is responsible for one or more OSCORE groups, for the respective Group Identifier (Gid) used as OSCORE ID Context, and for the Sender ID and Recipient ID of the respective group members.

The Group Manager has exclusive control over the Gid values uniquely assigned to the different groups under its control and over the Sender ID and Recipient ID values uniquely assigned to the members of each of those groups. A CoAP endpoint receives the Gid and other OSCORE input parameters, including its Sender ID, from the Group Manager upon joining the OSCORE group. That Sender ID is valid only within that group, and is unique within the group.

Furthermore, the Group Manager stores and maintains the public keys of endpoints joining a group and provides information about the group and its members to other current group members. At any time, a group member can retrieve from the Group Manager the public key and other information associated to other group members.

It is recommended that the Group Manager takes care of the group joining by using the approach defined in Reference [50], as based on the ACE framework for authentication and authorization in constrained environments [43]. Further details on the method used for joining an OSCORE group are out of the scope of this article.

5.2.1 Renewal of Group Keying Material. Due to a number of reasons, the Group Manager may force the members of an OSCORE group to establish a new Security Context by revoking the current group key material and distributing new one (rekeying). To this end, a new Gid for the OSCORE group and a new value for the Master Secret parameter is distributed to the group members. When doing so, the Group Manager may additionally distribute also a new value for the Master
Salt parameter, while it should preserve the same current value of the Sender ID of each group member.

Then, each group member re-derives the key material in its own Sender Context and Recipient Contexts (see Section 5.3), using the newly distributed Gid and Master Secret parameters. The Master Salt used for the re-derivations is the newly distributed Master Salt if provided by the Group Manager or an empty byte string otherwise. Thereafter, each group member uses its latest installed Sender Context to protect its own outgoing messages.

When a current endpoint leaves the group, the Group Manager renews the group key material and informs the remaining members about the leaving endpoint. This keeps the group members able to correctly assert the group membership of a message sender and additionally preserves forward secrecy in the group. Furthermore, in accordance with the specific application requirements, it is recommended to rekey the group also when a new joining endpoint is added to the group, thus preserving backward secrecy as well.

Group OSCORE is not devoted to a particular key management scheme for rekeying the OSCORE group. However, the Group Manager should support the distribution of the new Gid and Master Secret parameter to the OSCORE group according to the Group Rekeying Process defined in Reference [50]. Alternatively, different more advanced and efficient methods for group rekeying can be used from the many available in the literature, such as References [14, 17, 24, 25, 47, 48].

5.3 Main Differences from OSCORE

This section introduces in which respects Group OSCORE differs from OSCORE, with a focus on the data structure and key material stored by group members, as well as the COSE object and compressed encoding of OSCORE messages.

As a particular case, a group member can assume the special role of silent server. This kind of endpoint is interested in receiving request messages but never replies to them. An endpoint can implement both a silent server and a client, as the two roles are independent. However, an endpoint implementing only a silent server processes only incoming requests, maintains less keying material, and especially does not have a Sender Context for the OSCORE group.

5.3.1 The Security Context.

Each member of an OSCORE group stores a Security Context (see Section 4.2), which is extended as follows with the respect to the original format considered in OSCORE.

- The Common Context, shared by all the group members, specifies also (i) the Signature Encryption Algorithm used to encrypt messages when using the group mode, (ii) the Signature Algorithm used to compute the message signature when using the group mode, and (iii) the Pairwise Key Agreement Algorithm used to derive pairwise symmetric keys, when using the pairwise mode. Possible parameters associated to these algorithms are embedded in the stored public keys of group members. The AEAD Algorithm parameter inherited from the original format of the Security Context specifies the encryption algorithm used to protect messages with the pairwise mode.

Furthermore, the ID Context parameter contains the Gid of the OSCORE group, which is thus used as Context ID for that group. The choice of the Gid is specific to the application running at the Group Manager. It is up to the application running at the group members how to handle possible collisions between Gids, as used for OSCORE groups managed by different, non-synchronized Group Managers.

- The Sender Context includes also the endpoint’s private key, unless the endpoint is configured exclusively as silent server. When using the group mode, the private key is used to compute the message signature. When using the pairwise mode, the private key is used to...
derive a pairwise key between the endpoint and another member of the OSCORE group. It is out of scope for Group OSCORE how the private key has been established.

- Multiple Recipient Contexts are included, i.e., one for each endpoint from which messages are received. No Recipient Contexts are maintained as associated to endpoints from which messages are not (expected to be) received. The Recipient Context is extended with the public key of the associated endpoint. When using the group mode, the public key is used to verify the message signature. When using the pairwise mode, the public key is used to derive a pairwise key shared with the associated endpoint.

The public key of the associated endpoint and the input parameters for deriving the Recipient Context may be provided to the recipient endpoint upon joining the OSCORE group. Alternatively, these parameters can be acquired at a later time, for example the first time a message is received from this particular associated endpoint in the OSCORE group. The received message, together with the Common Context, includes everything necessary to derive a Security Context for verifying a message, except for the public key of the associated endpoint.

For particularly constrained devices, it can be not feasible to simultaneously handle the ongoing processing of a recently received message and the retrieval of the associated endpoint’s public key. Such devices may instead be configured to drop a received message for which there is currently no Recipient Context, and retrieve the public key of the sender endpoint to have it available to verify subsequent messages from that endpoint.

Group OSCORE uses the same derivation process of OSCORE to derive Sender Context and Recipient Context—and specifically symmetric Sender/Recipient Keys—from a set of input parameters.

### 5.3.2 The COSE Object

Compared to OSCORE (see Section 4.2), the following differences apply to the COSE Object.

- When using the group mode, the COSE Object includes an additional digital signature. Its value is set to the countersignature of the encrypted COSE Object, computed by the sender CoAP endpoint by using its own private key and according to the Signature Algorithm specified in the Security Context.

  The literature traditionally considers a countersignature as applied over another signature (i.e., not over any other security structure) and by a principal different from the one that produced what is being countersigned. However, COSE defines a countersignature as applicable also to other security structures. Furthermore, in a group communication scenario and especially building on the design choices of Group OSCORE, it is also appropriate and correct that the same principal as a group member both encrypts a message and then countersigns the result, thus proving to be the actual message sender and ensuring message source authentication.

- The “kid” parameter is present in all messages, i.e., both requests and responses, specifying the Sender ID of the endpoint transmitting the message. An exception is possible only for response messages, if sent as a reply to a request protected with the pairwise mode.

- The “kid context” parameter is present in every request message, specifying the Gid value of the group’s Security Context. This parameter remains optional to include in response messages.

- The AAD is extended to include additional information, i.e., the algorithms specified in the Common Context; the “request_kid_context” parameter specifying the Gid used when
protection of a request message; and the binary serialization of the OSCORE Option. Like in OSCORE, the AAD is not transmitted but only takes part in the secure message processing.

Compared to OSCORE (see Section 4.2), the following differences apply to the encoding of a Group OSCORE message.

- When using the group mode, the ciphertext of the COSE Object included as payload of the protected message is further concatenated with the value of the countersignature of the COSE Object.
- In the first byte of the OSCORE option, the sixth least significant bit, namely the Group Flag bit, is used to signal the usage of the group mode. In particular, the Group Flag bit is set to 1 if the message is protected using the group mode. In any other case, including when using the pairwise mode, this bit is set to 0.

5.4 The Group Mode

This section describes how Group OSCORE protects messages in group mode, as main differences from OSCORE (see Section 4.2). In particular, source authentication of messages is achieved by appending a signature to the message payload, computed by using the private key of the message sender. Message confidentiality is achieved at a group level, i.e., every member of the OSCORE group is able to decrypt a message protected in group mode.

5.4.1 Protection of Requests. A CoAP client protects a request in group mode as in OSCORE, with the following differences.

- The extended AAD discussed in Section 5.3.2 is used for encrypting and signing the request, while the encryption and encoding of the COSE object are as defined in Section 5.3.2.
- A countersignature of the encrypted COSE object is computed and added at the end of the payload of the protected request message.

5.4.2 Verification of Requests. A server verifies a request protected in group mode as in OSCORE, with the following differences.

- The decoding of the compressed COSE object follows the updates discussed in Section 5.3.2, while the extended AAD discussed in Section 5.3.2 is used for decrypting the request and verifying the countersignature of the encrypted COSE object.
- If the received Recipient ID (“kid”) does not match with any Recipient Context for the retrieved Gid (“kid context”), then the server may create a new Recipient Context and initialize it at that point in time, also retrieving the client’s public key. Such a configuration is application specific. If the application does not specify dynamic derivation of new Recipient Contexts, then the server stops processing the request.
- Before decrypting the request, the server verifies the message signature using the public key of the client from the associated Recipient Context. If the verification fails, then the server may reply with a 4.00 (Bad Request) response.
- If the used Recipient Context was created upon receiving this group request and the message is not decrypted and verified successfully, then the server may delete that Recipient Context. Although application specific, this configuration prevents attacks aimed at overloading the server’s storage and creating processing overhead.

5.4.3 Protection of Responses. A server protects a response in group mode as in OSCORE, with the following differences.
• The encoding of the compressed COSE object follows the updates in Section 5.3.2, while the extended AAD discussed in Section 5.3.2 is used for encrypting and signing the response.

• A countersignature of the encrypted COSE object is computed and added at the end of the payload of the protected response message.

Since a group rekeying can occur, with consequent re-establishment of the Security Context, the server must always protect a response by using its Sender Context from the latest owned Security Context. As a consequence, right after a group rekeying has been completed, the server may end up protecting a response by using a Security Context different from the one used to protect the group request. In such a case, the server must (i) use its current Sender Sequence Number value to build the nonce for the encryption process, (ii) include in the response the "Partial IV" field of the OSCORE Option and set it to the Sender Sequence Number above, and (iii) increment its Sender Sequence Number by one.

5.4.4 Verification of Responses. A client decrypts and verifies a response in group mode as in OSCORE, with the following differences. Note that, as discussed in Section 5.4.3, a client may receive a response protected with a Security Context different from the one used to protect the corresponding group request.

• The decoding of the compressed COSE object follows the updates in Section 5.3.2, while the extended AAD discussed in Section 5.3.2 is used for decrypting the response and verifying its signature.

• If the received Recipient ID ("kid") does not match with any Recipient Context for the retrieved Gid ("kid context"), then the client may create a new Recipient Context and initialize it at that point in time, also retrieving the server’s public key. Such a configuration is application specific. If the application does not specify dynamic derivation of new Recipient Contexts, then the client stops processing the response.

• Before decrypting the response, the client verifies the signature using the public key of the server from the associated Recipient Context.

• If the used Recipient Context was created upon receiving this response and the message is not decrypted and verified successfully, then the client may delete that Recipient Context. Although application specific, this configuration prevents attacks aimed at overloading the client’s storage and creating processing overhead.

6 EXPERIMENTAL EVALUATION

In this section, we present our chosen approach and the experimental setup used to evaluate the performance of the Group OSCORE protocol and to compare it with unicast OSCORE, Group CoAP, and unicast CoAP.

6.1 Methodology

In this work, we aim to investigate the performance of Group OSCORE in scenarios where (most) devices are Central Processing Unit (CPU) constrained and most likely battery-powered devices. For example, in a secure firmware/configuration update scenario or a building or lighting control use-case (see Section 3), a single client communicates with multiple servers (sensors, controllers), yielding changes in their state or requesting sensitive data.

Figure 3 depicts the network topology and network protocol stack considered for all our experiments, one constrained client communicate with three constrained servers. No proxies or any other kinds of intermediary nodes were considered in our experimental scenarios.

For our tests, we have tested the four different scenarios defined below.
(1) Unprotected CoAP, with no security mechanism applied and all messages sent unicast.
(2) Unprotected CoAP with group communication functionality, with no security mechanism applied and with the client sending request messages over multicast and expecting response messages as unicast.
(3) Protected CoAP, i.e., using OSCORE to protect all messages, each of which is sent unicast.
(4) Protected CoAP with group communication functionality, using Group OSCORE to protect all messages, with the client sending request messages over multicast and expecting response messages as unicast.

In particular, in scenarios (2) and (4), the servers listened to a multicast address for group requests from the client. Furthermore, we evaluated scenarios (3) twice, once using security mechanisms implemented entirely in software and once with cryptographic operations executed in a dedicated cryptoprocessor.

Scenario (4) was evaluated thrice with different configurations. All messages are protected using the group mode of Group OSCORE, using as Signature Algorithm either ECDSA with curve P-256 or EdDSA with curve Ed25519. We chose to test both algorithms also, since the draft of Group OSCORE specifies that either ECDSA P-256 or EdDSA25519 is mandatory to support for constrained devices. Both the platforms that we have used for our tests have an hardware implementation of ECDSA P-256. Hence, we have tested ECDSA P-256 using both a software implementation and the hardware implementation. Neither platform supports EdDSA with Ed25519 in hardware, which thus was tested only using a software implementation.

### 6.2 Experimental Setup

All experiments were run with a constrained server of two different hardware platforms, namely Zolertia Firefly Rev. A [68] and TI CC1352R1 Launchpad [62]. We show an overview of the most relevant hardware platform parameters in Table 1. Note that we used the 2.4-GHz frequency band in the experiments.

They are compatible with the Contiki-NG operating system [16], our publicly available implementation of Group OSCORE can be found here.\(^5\)

We have used version 4.0 of Contiki-NG.\(^6\) The network stack used in these experiments are 2.4-GHz IEEE 802.15.4 LR-WPAN for physical and Medium Access Control layer with 6LoWPAN and IPv6 as the network layer. UDP is used at the transport layer. The implementations and settings

---


\(^6\)https://github.com/contiki-ng/contiki-ng/releases/tag/release%2Fv4.0.
Table 1. Summary of the Hardware Platforms Used in the Experiments

<table>
<thead>
<tr>
<th>Platform</th>
<th>MCU</th>
<th>Clock Freq.</th>
<th>RAM</th>
<th>ROM/Flash</th>
<th>Hardware Features</th>
<th>RF Bands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zolertia Firefly Rev. A (Zoul)</td>
<td>ARM Cortex M3</td>
<td>32 MHz</td>
<td>32 kB (16 kB in low-power mode)</td>
<td>512 kB</td>
<td>AES128/256, SHA2, ECC128/256</td>
<td>sub-GHz, 2.4 GHz</td>
</tr>
<tr>
<td>TI CC1352R1 Simplelink (Simplelink)</td>
<td>ARM Cortex M4F</td>
<td>48 MHz</td>
<td>80 kB + 8 kB cache</td>
<td>352 kB flash +256 kB ROM</td>
<td>AES128/256, SHA2, ECC128/256</td>
<td>sub-GHz, 2.4 GHz</td>
</tr>
</tbody>
</table>

used for these protocols are the default from the Contiki-NG source tree. Stateless Multicast RPL Forwarding was used for IPv6 multicast communication [27].

The cryptographic functions have been taken from a variety of sources. The software implementation of AES-CCM-128 used in OSCORE is taken from the Contiki-NG source-tree. The SHA256 software implementation used in both OSCORE and Group OSCORE is taken from Tinydtls.7 The hardware implementation of AES-CCM-128, SHA256, and ECDSA P-256 for the Zoul platform is also taken from the Contiki-NG source-tree. The same functions for the Simplelink platform was taken from the Simplelink SDK provided by Texas Instruments. The software implementations of asymmetric cryptography used is uECC (micro-ecc) for ECDSA P-2568 and Monocypher for EdDSA25519.9 These were chosen because they are written in portable C and have permissible licenses. Both libraries have shown good performance in previous evaluations [42].

The tested software configurations are as follows:

(1) In the CoAP case, no security schemes were used.
(2) In the Group-CoAP case, no security schemes were used.
(3) In the OSCORE-SW case, relevant cryptographic operations were performed on the primary CPU.
(4) In the OSCORE-HW case, the cryptographic operations were performed on the dedicated crypto accelerators of the IoT platforms.
(5) In the Group-OSCORE-SW case, the cryptographic operations were performed on the primary CPU. The asymmetric cryptography algorithm used is ECDSA, with curve P-256.
(6) In the Group-OSCORE-HW case, the cryptographic operations were performed on the dedicated crypto accelerators of the IoT platforms.
(7) In the Group-OSCORE-EDDSA case, the cryptographic operations were performed on the primary CPU. The asymmetric cryptography algorithm used is EdDSA, with curve Ed25519.

6.3 Experimental Scenarios

For each of the hardware and software configurations, 14 in total, the following experiments were conducted:

(1) Memory occupancy (RAM and ROM) on the server.
(2) Time elapsed on the client from the start of request processing to the end of response processing, or RTT.
(3) Time spent by the CPU on the server, to process incoming/outgoing messages.
(4) Energy consumption on the server, specific to message processing.

9https://monocypher.org/.
The measurement of memory occupancy (1) relied on the fact that Contiki-NG only dynamically allocates stack memory, while the heap is a static memory block. We determined heap usage with the GNU command `size`, while we measured the stack utilization with `stack painting`. We painted the stack by writing a known value to all the bytes of the stack at device boot, then we ran the experiments. After we finished the experiments, we counted the bytes with values different from the initial values.

We measured the RTT experienced by the client, as the time interval from the start of the processing of an outgoing request until the completion of the processing of the received response in (2).

As for the CPU time for message processing (3), the measured time interval included all the cryptographic operations considered by the used security protocol, i.e., encryption/decryption, integrity verification, and message signing/verifying.

To obtain the energy consumption measurements of message processing (5), we used a DC energy analyzer to measure the voltage and current supplied to the tested device (see Section 7.4).

During the RTT, CPU and energy consumption tests, a constrained Contiki-NG client generated and sent requests of varying payload sizes (1 to 128 bytes). This number (1 to 128) indicates the CoAP payload size, before any protocol overhead. The data transmitted were dummy bytes that were changed on the server to indicate that the responses had actually passed the server. These data were transmitted by the client and received by the servers, which in turn responded with messages transmitted to the client.

### 7 RESULTS AND DISCUSSION

In this section, we present the results of our experiments. We start with memory utilization on the servers, comparing the RAM and ROM requirements in the different scenarios. Then, we present the RTT on the observed client, followed by the overall processing time and the relative impact due to the performed cryptographic operations on the server side. Finally, we provide the measured energy consumption on the server side in the different scenarios as a vital element to consider for constrained devices.

#### 7.1 Memory Utilization

We calculated the memory utilization by combining data from compiled ELF files with stack usage numbers from runtime. We used the GNU tool `objdump`\(^\text{10}\) to extract the size of the `.Text`, `.BSS`, and `.Data` segments from the ELF files. Furthermore, we wrote a known bit pattern to the stack at system boot and ran the experiments, and afterwards we counted the number of bytes on the stack that the tested program had overwritten. We then calculated the RAM utilization as the size of `.Data` + `.BSS` + stack, while the ROM utilization was calculated as the size of `.BSS` + `.Text`.

Figures 4 and 5 present RAM and ROM utilization, additionally highlighting the stack usage and the memory overhead corresponding to the cryptographic operations. Noteworthy, the use of the stack depends on the message size, while implementation issues influences the code size for cryptography. Regarding the scenarios defined in Section 6.1 and summarized in Table 3, the bars shown in the figures use the following labels: `COAP` for scenario (1); `Group-COAP` for scenario (2); `OSCORE-SW` for scenario (3) and `OSCORE-HW` for scenario (4), with cryptographic operations performed in software or with hardware acceleration, respectively; and `Group-OSCORE-SW` for scenario (5), `Group-OSCORE-HW` for scenario (6), with cryptographic operations performed in software or with hardware acceleration, respectively. Finally, `Group-OSCORE-EDDSA` for scenario (7) covers also the use of the EdDSA signature algorithm in software.

Figure 4 shows the memory utilization for the Zoul platform. The bars represent memory utilization for the whole system, i.e., the Contiki-NG operating system, network stack, default libraries, drivers and a CoAP server possibly including OSCORE or Group OSCORE. The slight increase in both RAM and ROM for Group-COAP compared to COAP is caused by the inclusion of IPv6 Multicast routing functionality. Additional routines for processing OSCORE messages add 2 KB of extra ROM and another 1 KB for the AES-CCM-128 and SHA256 algorithms. Although hardware-accelerated cryptography used in Group-OSCORE requires less ROM than the cryptographic software libraries, it requires more RAM. The code used to interface with the hardware accelerator on the Zoul platform implements ECDSA P-256 without any optimizations. This negatively impacts performance in terms of memory utilization. The software library used for ECDSA P-256, micro-ecc, is optimized to conserve memory. This explains how the hardware implementation of ECDSA P-256 surprisingly utilize more memory. Finally, in the case of Group-OSCORE-EDDSA we observe a larger RAM and ROM utilization that we attribute to the used cryptographic library, which has been shown to require significant memory resources [42].
Figure 5 shows the memory utilization for the Simplelink platform. Note that the comparison between software and hardware-accelerated cryptography solutions looks slightly different than in the case of the Zoul platform (see Figure 4), which results from hardware discrepancies and the nuances of particular device driver implementations.

Considering that the Zoul platform has 32 KB of RAM and 512 KB of ROM, the increased amount of memory needed for Group-OSCORE processing seems reasonable. We can conclude that fitting Contiki-NG with a network stack and our Group-OSCORE implementation in addition to application code is feasible on the Zoul platform. Moreover, low-power operation, resulting in only half of Zoul RAM memory available, could be achieved with Group-OSCORE on board. The Simplelink platform has even more memory, i.e., 88 KB of RAM and 608 KB of ROM. Thus, the increased memory utilization due to Group-OSCORE is not significant in terms of the total memory.

### 7.2 Message Round Trip Time

We measured the RTT as the time elapsed from the start of processing the outgoing request in the constrained client until the moment when the received response has been processed by the client. In particular, we considered different sizes for the application payload, excluding any message overhead from headers, AEAD-encryption tags, or signatures.

Because of the response delay randomization required by CoAP group communication (see Section 4.1.1), we considered both the time elapsed until the first response arrived and the time elapsed until the last response arrived. As used together with group communication, Group OSCORE also relies on the response delay randomization mechanism. When evaluating the test cases relying on group communication, the servers chose the delay of each response as a random value between 0 and 8 s, to conservatively prevent collisions and ensure a feasible collection of results. It follows that, on average, a large share of RTT values when group communication is used consists of the added randomized delay.

Figure 6 shows the RTT measured when considering the Zoul platform, where we can see two sets of response times. The first set comprises response times for the three test cases, COAP, OSCORE-SW, and OSCORE-HW. Consistently with a quick response delivery, they are all low for both the first and the last response, with no notable influence from the considered message payload sizes.
These second set comprises response times for the four test cases relying on group communication, i.e., Group-COAP, Group-OSCORE-SW, Group-OSCORE-HW, and Group-OSCORE-EDDSA. In these cases, the response times are considerably greater and fall within a large range of possible values. This is mainly due to the random delay necessitated when using group communication for CoAP, as introduced by each responding server (see Section 4.1.1). The extent of such an impact may largely vary on a per-response basis, depending on the exact delay randomly selected and introduced by each server. Also, its contribution to the overall response time might be negligible, or rather comparable to or bigger than that from possible cryptographic operations, again on a per-message basis. Thus, the contribution of security operations is later analyzed in Section 7.3, when discussing CPU utilization. At the same time, the variability introduced by the random delays overshadows the possible role played by the message payload sizes. More in general, the range of values where the random delay is selected from can be determined according to the characteristics of the system and network deployment, hence its width could be limited in particular settings, in turn limiting the variability of response times.

Figure 7 shows the RTT measured for the Simplelink platform. We can make the considerations discussed above for the Zoul platform and the related results shown in Figure 6. In particular, the three purely-unicast test cases display small, consistent and similar response times, while the four test cases relying on group communication display a larger response time with a large range of values due to the random delays introduced by the servers.

7.3 CPU Utilization
To evaluate the impact of the cryptographic operations on the overall message processing, we have also measured the time required to perform such functions and compared it against the total message processing time. For OSCORE, we have measured the time needed to encrypt (decrypt) a message and presented the results as the percentage of the total time required to serialize an outgoing (parse an incoming) message. For Group-OSCORE, we have measured the time needed to encrypt and sign (or decrypt and verify the signature of) an outgoing (incoming) message and presented the results as the percentage of the total time needed to serialize an outgoing (parse an incoming) message.
Table 2. CPU Usage of the Zoul Platform in Milliseconds for Serializing (S) and Parsing (P) of Messages with Cryptographic Operations as a Percentage of Total Payload (Bytes)

<table>
<thead>
<tr>
<th>Payload (Bytes)</th>
<th>1</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S</td>
<td>P</td>
<td>S</td>
<td>P</td>
</tr>
<tr>
<td>COAP</td>
<td>0.05</td>
<td>0.12</td>
<td>0.07</td>
<td>0.12</td>
</tr>
<tr>
<td>Group-COAP</td>
<td>0.05</td>
<td>0.12</td>
<td>0.05</td>
<td>0.12</td>
</tr>
<tr>
<td>OSCORE-SW</td>
<td>0.51</td>
<td>0.56</td>
<td>0.71</td>
<td>0.76</td>
</tr>
<tr>
<td>Encrypt/Decrypt</td>
<td>54.2%</td>
<td>53.0%</td>
<td>65.0%</td>
<td>64.3%</td>
</tr>
<tr>
<td>OSCORE-HW</td>
<td>0.32</td>
<td>0.38</td>
<td>0.35</td>
<td>0.39</td>
</tr>
<tr>
<td>Encrypt/Decrypt</td>
<td>26.7%</td>
<td>27.6%</td>
<td>29.3%</td>
<td>28.1%</td>
</tr>
<tr>
<td>G-OSCORE-SW</td>
<td>576.97</td>
<td>624.99</td>
<td>577.68</td>
<td>626.98</td>
</tr>
<tr>
<td>Encrypt/Decrypt</td>
<td>99.9%</td>
<td>99.9%</td>
<td>99.8%</td>
<td>99.9%</td>
</tr>
<tr>
<td>G-OSCORE-HW</td>
<td>348.84</td>
<td>706.55</td>
<td>349.08</td>
<td>711.96</td>
</tr>
<tr>
<td>Encrypt/Decrypt</td>
<td>99.9%</td>
<td>99.9%</td>
<td>99.8%</td>
<td>99.9%</td>
</tr>
<tr>
<td>G-OSCORE-EDDSA</td>
<td>107.50</td>
<td>269.03</td>
<td>109.13</td>
<td>268.09</td>
</tr>
<tr>
<td>Encrypt/Decrypt</td>
<td>99.9%</td>
<td>99.8%</td>
<td>99.4%</td>
<td>99.8%</td>
</tr>
</tbody>
</table>

Table 2 shows the processing times for the Zoul platform in milliseconds. The table reveals that hardware acceleration for cryptographic operations improves OSCORE performance, as hardware accelerated cryptography takes up a smaller fraction of the total processing time, i.e., less than 50%. For Group-OSCORE, cryptographic functions are responsible for the absolute majority of time taken to serialize and parse messages. Particularly, over 99% of the processing time is spent for producing or verifying the signature of an outgoing or incoming message, respectively. Note that using hardware acceleration for cryptographic operations reduces the overall time needed to serialize messages. Instead, signature verification and the total message parsing time do not show the same performance improvement when hardware acceleration for cryptographic operations is applied. The reason is the relative efficiency displayed by the software implementation of ECDSA P-256. The ECDSA software implementation used in the Group-OSCORE-SW code was μECC,\(^\text{11}\) which has outstanding performance compared to both other ECDSA software implementations [46]. This performance has been achieved through careful optimizations of Elliptic-Curve operations in Assembly and the use of Shamir’s Trick [54], which reduces the number of Elliptic-Curve scalar multiplications needed to verify a signature from two to one. In particular, relying on Shamir’s Trick reduces the time to verify a signature by approximately 40%.

EdDSA25519 shows even better performance and beats both the software and hardware implementation of ECDSA P-256. This implementation shows consistently superior performance for both signing and verifying messages.

We present the results for the Simplelink platform in Table 3. The reported values follow the same trends as in Table 2, but they are generally lower compared to the values for the Zoul platform. One notable exception is OSCORE-SW, where the times are longer in the Simplelink case. This is due to the less efficient AES-CCM Simplelink implementation, rendered by the increased delay share of the OSCORE-SW crypto functions in Table 3.

The processing time results for Zoul and Simplelink platforms show similar trends, albeit shorter total times for the Simplelink platform can be observed. OSCORE cryptography takes between 25% and 90% when implemented in software and between 10% and 55% when implemented in hardware. This is a significant part of the processing time, but it is noticeably smaller compared to the 99% of the total time that cryptography takes up when Group-OSCORE is used. As learned from the OSCORE experiments, 1–2 ms were needed to encrypt and decrypt a message, while the further

---

Table 3. CPU Usage of the SimpleLink Platform in Milliseconds for Serializing (S) and Parsing (P) of Messages with Cryptographic Operations as a Percentage of Total Payload (Bytes)

<table>
<thead>
<tr>
<th>Payload (Bytes)</th>
<th>1</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S</td>
<td>P</td>
<td>S</td>
<td>P</td>
</tr>
<tr>
<td>COAP</td>
<td>0.00</td>
<td>0.18</td>
<td>0.06</td>
<td>0.16</td>
</tr>
<tr>
<td>Group-COAP</td>
<td>0.03</td>
<td>0.20</td>
<td>0.05</td>
<td>0.20</td>
</tr>
<tr>
<td>OSCORE-SW</td>
<td>1.85</td>
<td>1.98</td>
<td>2.93</td>
<td>3.06</td>
</tr>
<tr>
<td>Encrypt/Decrypt</td>
<td>90.8%</td>
<td>85.2%</td>
<td>93.8%</td>
<td>90.0%</td>
</tr>
<tr>
<td>OSCORE-HW</td>
<td>0.31</td>
<td>0.46</td>
<td>0.37</td>
<td>0.49</td>
</tr>
<tr>
<td>Encrypt/Decrypt</td>
<td>41.2%</td>
<td>34.2%</td>
<td>50.0%</td>
<td>37.5%</td>
</tr>
<tr>
<td>G-OSCORE-SW</td>
<td>296.91</td>
<td>317.10</td>
<td>298.40</td>
<td>319.06</td>
</tr>
<tr>
<td>Encypt/Decrypt</td>
<td>99.8%</td>
<td>99.8%</td>
<td>99.8%</td>
<td>99.8%</td>
</tr>
<tr>
<td>G-OSCORE-HW</td>
<td>233.89</td>
<td>467.10</td>
<td>233.95</td>
<td>466.39</td>
</tr>
<tr>
<td>Encrypt/Decrypt</td>
<td>99.8%</td>
<td>99.9%</td>
<td>99.8%</td>
<td>99.9%</td>
</tr>
<tr>
<td>G-OSCORE-EDDSA</td>
<td>50.58</td>
<td>112.17</td>
<td>52.63</td>
<td>112.92</td>
</tr>
<tr>
<td>Encrypt/Decrypt</td>
<td>99.1%</td>
<td>99.5%</td>
<td>99.0%</td>
<td>99.4%</td>
</tr>
</tbody>
</table>

increase in the processing delay is due to verifying a signature or generating one. Hardware acceleration shortens the total processing time for Group-OSCORE. Ed25519 using the Monocypher library shows excellent performance in terms of speed on both platforms, outperforming ECDSA P-256 in both hardware and software on both platforms. This speed comes at the penalty of a larger memory footprint in ROM and the largest utilization of RAM of all the tested asymmetric cryptography implementations on both platforms. This leads to a remark that asymmetric cryptography is more time-consuming compared to symmetric cryptography, when measured on these types of constrained devices.

7.4 Per-message Energy Consumption

For resource-constrained devices, energy consumption is an important metric to assess when evaluating protocol performance. In this article, we have measured the energy consumption by using the following method.

We have used a DC energy analyzer, an instrument that measures voltage and current with a high precision at a very high sampling frequency. In particular, we used a Joulescope as the DC energy analyzer to measure the power consumption of the tested software configurations on both hardware platforms. Power was supplied to the device by a fixed voltage from a small power supply. The Joulescope measures voltage and current supplied to the device, as evaluated at 2 million samples per second with nanoampere precision.\footnote{\url{https://www.joulescope.com/}; retrieved January 21, 2022.}

We opted to measure the actual energy consumed during message processing. As in the case of CPU-time measurements, we considered the period between the start of application-layer processing of the incoming message until the message is delivered to the application. We also measured the total time spent by the application-layer to fully prepare and process an outgoing message, including possible security operations (see Section 7.3).

We did our experiments with 10 message exchanges per payload size, since the times measured in Section 7.3 had small deviations from the mean.

Figure 8 shows the measured energy consumption for the Zoul platform. Note that the Y-scale is logarithmic, since it was impossible to adequately show the large difference in energy consumption for both the Group-OSCORE and COAP cases in the same plot using a linear scale.
Looking at the bars for OSCORE-SW, OSCORE-HW, Group-OSCORE-SW, and Group-OSCORE-HW, it becomes clear that the per-message energy consumption increases with growing message sizes for OSCORE-SW and OSCORE-HW, but remains virtually constant for COAP, Group-COAP and the three Group-OSCORE variants. This can be explained by the time needed to generate and verify ECC signatures, which is almost constant, as discussed in Section 7.2. Furthermore, the energy consumption for Group-OSCORE-EDDSA is lower than that for the Group-OSCORE-SW and Group-OSCORE-HW results. This is due to the efficient (i.e., fast) cryptographic library for EdDSA signing used. When it comes to COAP and Group-COAP, the roughly constant energy consumption can be explained by the very short CPU time observed in Section 7.3. For OSCORE-SW and OSCORE-HW, a linear growth trend can be observed, because of the increased processing time needed to encrypt the payload of the messages.

In Figure 9, we show the results for the Simplelink platform. As for the previous case, one can witness the situation, where COAP, Group-COAP, Group-OSCORE-SW, Group-OSCORE-HW, and Group-OSCORE-EDDSA show a roughly constant energy consumption across message payload sizes. Instead, OSCORE-SW and OSCORE-HW show a linear relation between message payload size and energy consumption. This is due to the encryption and decryption of the message payload taking more CPU-time, and thus energy, to process a larger payload.

Among the three test cases relying on Group OSCORE, Group-OSCORE-SW displays the highest energy consumption for both parsing and serializing messages, while Group-OSCORE-HW and Group-OSCORE-EDDSA display the lowest energy consumption and a value in between, respectively.

For the Zoul platform, the per-exchange energy consumption is 0.3 mJ for OSCORE-SW, 125 mJ for Group-OSCORE-SW, 107 mJ for Group-OSCORE-HW, and 39 mJ for Group-OSCORE-EDDSA. Using OSCORE-SW as a baseline, Group-OSCORE-SW consumes 416 times more energy per message exchange. For Group-OSCORE-HW, the corresponding number is 356 times the energy of OSCORE-SW. Lastly Group-OSCORE-EDDSA consumes 130 times more energy per message exchange.

For the Simplelink platform, the processing energy consumption per exchange is 2 mJ for OSCORE-SW, 96 mJ for Group-OSCORE-SW, 9 mJ for Group-OSCORE-HW, and 27 mJ for Group-OSCORE-EDDSA.
Group-OSCORE-EDDSA. This results in a 48 times increase in energy consumption for Group-OSCORE-SW, a 4.5 times increase for Group-OSCORE-HW, and a 13.5 times increase for Group-OSCORE-EDDSA.

Considering a 5000-mAh battery, operating at 3.3 V (equivalent to 16.5 Wh or 59400 J of energy), one can realise that it becomes possible to perform the following amount of message exchanges:

- Simplelink platform: 30 million OSCORE-SW transactions, 619 thousand Group-OSCORE-SW transactions, 6.6 million Group-OSCORE-HW transactions, and 2.2 million Group-OSCORE-EDDSA transactions.

As a matter of fact, the above calculations are rough and do not take into account physical phenomena (e.g., leakage currents or battery degradation) as well as application overhead, which occur when a device is deployed in a real-life setting. Nevertheless, the amount of possible client-server transactions in Group-OSCORE variants is in our opinion sufficiently high to claim that this protocol can be applied in low-power, low-data IoT applications.

8 CONCLUSIONS

In this article, we have presented and experimentally evaluated the Group-OSCORE security protocol. This can be used to protect CoAP messages end-to-end at the application level in group communication scenarios, where a CoAP client sends a request intended to multiple CoAP servers, e.g., over IP multicast. In particular, source authentication of messages is achieved by means of digital signatures embedded in the protected payload.

To best of our knowledge, this work is the first publicly available Group OSCORE performance evaluation using an implementation on real, constrained IoT hardware. For our study, we developed a Contiki-NG implementation (published as open source) and tested the protocol on two platforms: Zolertia Firefly Rev. A CC2538 and TI Simplelink CC1352R1 Launchpad. Furthermore we considered multiple signing algorithms, implementing and evaluating the Group OSCORE solution both for EdDSA and ECDSA P-256.
Our experiments encompassed memory consumption, RTT and energy consumption overhead of the novel protocol. The results showed that incorporating Group OSCORE does not introduce significant RAM/ROM penalty on the two tested platforms. However, Group OSCORE operations (most notably message signing and signature verification) contribute to a considerably larger RTT, when compared to CoAP, Group CoAP and OSCORE. Group OSCORE also consumes more energy per message sent if compared to OSCORE, which might be a limiting factor for constrained, battery powered IoT nodes communicating frequently. Nevertheless, we have showed that the number of possible message exchanges is high, also for battery-powered devices. Finally we observed that using the EdDSA signature algorithm resulted in better performance than ECDSA P-256, especially when considering the ECDSA P-256 software implementation.

We believe that, to fully understand the potential and limitations of Group OSCORE, more trials are appropriate to be conducted in the future. Specifically, a follow-up evaluation can take into account additional hardware platforms. Furthermore, it can also consider the pairwise mode, which uses only symmetric encryption to protect the unicast responses. This is expected to substantially reduce the computing time and the corresponding energy consumption, as well as the Round Trip Time experienced by a client.

ACKNOWLEDGMENTS
The authors sincerely thank the anonymous reviewers and the Editor-in-Chief for their insightful comments and suggestions, that have helped to improve the technical and editorial quality of the manuscript. The authors thank Matthew Bradbury for his valuable help with the implementation and Pegah Nikbakht Bideh for her valuable help with the energy measurement equipment.

REFERENCES


Received May 2021; revised January 2022; accepted February 2022