ORSHIN

D5.2

Report about essential and beyond-essential s&p guarantees for intra-device communication in restricted environments

Project number	101070008
Project acronym	ORSHIN
Project title	Open-source ReSilient Hardware and software for Internet of thiNgs
Start date of the project	1st October, 2022
Duration	36 months
Call	HORIZON-CL3-2021-CS-01

Deliverable type	Report
Deliverable reference number	CL3-2021-CS-01/ D5.2/ 1.0
Work package contributing to the deliverable	WP5
Due date	Jun 2025 - M33
Actual submission date	30th June 2025

Responsible organisation	SEC
Editor	Security Pattern
Dissemination level	PU
Revision	1.0

Abstract	This deliverable reports on the results of WP5 of the ORSHIN project, T5.3 and T5.4. The WP concluded successfully. This document conducts an examination of current protocols used to secure intra-device communication, including standard, proprietary, and literature-based approaches. We observed a weakness in the use of JSON Web Tokens (JWTs) as a method for authenticating devices on the Cloud. Focusing on the JWT standard, we show an attack that exploits a weakness in it, allowing an attacker to exploit an IoT device during manufacturing, to prepare authentication tokens that can be used by an

Funded by the European Union under grant agreement no. 101070008. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

	attacker to impersonate the device to the cloud service. Our study also motivated us to conceive a new secure channel protocol, called NSCP, which aims to replace the standard SCP03 protocol for secure communications, in order to improve the state of the art, both in efficiency and power consumption.
Keywords	Secure channel protocol, Intra-device communication, Handshake, Record protocol

Editor

SEC

Contributors (ordered according to beneficiary numbers)

Alberto Battistello (SEC)

Federico Gorla (SEC)

Arianna Gringiani (SEC)

Maria Chiara Molteni (SEC)

Lorenzo Nava (SEC)

Reviewers (ordered according to beneficiary numbers)

Benedikt Gierlichs (KUL)

Jan Pleskac (TRPC)

Disclaimer

Executive Summary

The content of this document is the results of the work done for ORSHIN's WP5 in tasks 5.3 and 5.4 about the study of Security and Privacy (s&p) guarantees for intra-device communication in restricted environments. Our work is harmonised with ORSHIN's tasks 5.1 and 5.2 that study the s&p guarantees for inter-device communication, and the other WPs of the ORSHIN's project.

One of the key aspects of the ORSHIN objective, is the analysis of the state of the art security of communications in restricted environments. This directly translates to the analysis of secure communications of a constrained embedded device between the main controller and the peripheral units, like sensors, Secure Elements, or other processors. This is true in particular when the cryptographic material, necessary to establish secure communications with the world outside of the device, is contained in the secure element and needs to be used by the main processor. Thus a secure channel needs to be established between the two components. In this document we provide a detailed study of the different protocols that are used by modern systems to secure such links.

We start our study with the Secure Channel Protocol (SCP) used in most Secure Elements and published by the GlobalPlatform consortium, and the Replay Protected Memory Block (RPMB) protocol used to secure the communication with memories. Our analysis then moves on to other proprietary protocols, like the one used in Optiga products by Infineon, the secure protocol used in Microchip's ATECC chips, or the Chip2Chip security protocol used in Ublox products. We then extend our analysis to other literature publications that tackle the problem of providing a robust and sound record protocol, as it is used in many modern applications worldwide. Finally we illustrate some further constructions that can be interesting to complete the scenario, such constructions, like the JSON Web Token (JWT), show emerging ways to build and deploy security by using standardized basic blocks.

Our work towards the accomplishment of the ORSHIN project then provides an in depth investigation of one particular communication protocol enabled by the use of the JSON Web Tokens (JWT) standard and its use in actual implementations. JWTs are a compact, URL-safe means of representing claims to be transferred between two parties. They are for example widely used as credential tokens, allowing users that are logged in their phone, to access their Google Drive documents without the need to input the credentials again. We present *JWT Back to the future*, the possibility of using the device for preparing JWT that will be later used to connect to the cloud service. A malicious user in the supply chain, for instance someone having access to a testing machinery capable of interacting with all the devices, might have the possibility to collect a large number of JWT for mounting a massive attack in the future. This work has been submitted to the LIGHTWEIGHT CRYPTOGRAPHY FOR SECURITY & PRIVACY (LightSEC 2025) conference 2025.

As a final result of the Beyond essential s&p guarantees for intra-device communication task, we developed an original protocol that takes advantage of open-source hardware. We called this new secure channel protocol NSCP; it aims to replace the standard SCP03 protocol for secure communications. We published it at the DATE2025 conference [Date25].

Table of Content

Chapter 1 Introduction 1
Chapter 2 State of the Art of Intra-Device Comms 3
2.1 State-of-the-Art Protocols 3
2.1.1 The Secure Channel Protocol Family 4
2.1.2 Secure Channel Protocol 03 (SCP03) 5
2.1.3 OPTIGA 9
2.1.4 ATECC 11
2.1.5 Replay Protected Memory Block (RPMB) 13
2.1.6 U-blox 15
2.2 Secure Channel Protocol Frameworks 18
2.2.1 NaCl 18
2.2.2 Noise 20
2.2.3 BLINKER 23
2.2.4 Strobe 25
2.2.5 Remarks 27
2.3 JSON Web Tokens 28
2.3.1 JWT 29
2.3.2 JWS 29
2.3.3 JWE 29
2.3.4 Supported cryptographic algorithms 29
2.3.5 Sign then encrypt 30
2.3.6 Remarks 30
2.3.7 Implementations 31
2.4 Xoodyak 31
2.4.1 Xoodoo 32
2.4.2 Cyclist 33
2.4.3 Implementations 34
2.4.4 Final remarks 34
2.5 Conclusions 34
Chapter 3 JWT Back to the future 35
3.1 JWTs, JWSs and JWEs 36
3.2 JWT 36
3.3 JWT 37
3.4 JWE 37
3.5 Sign then encrypt 37
3.6 Other formats: CWTs and EATs 37

3.7 The IoT generic architecture 37
3.8 Device's lifecycle 39
3.9 Manufacturing 39
3.10 Deployment 40
3.11 Threat Model 40
3.12 Back to the future 40
3.13 Attack 41
3.14 A JWT weakness 43
3.15 Countermeasures 44
3.16 Use the TLS authentication 44
3.17 Pre-provisioned keys 44
3.18 Use the nonce claim 44
3.19 Key use limit 45
3.20 Trusted supply chain 45
3.21 Conclusions 45
Chapter 4 A New Secure Channel Protocol 47
4.1 Overview 47
4.2 Background 47
4.2.1 The Secure Channel Protocol 47
4.2.2 The Xoodyak Primitive 50
4.3 NSCP: a new secure channel protocol for hardening communications in industrial IoT . 51
4.4 Evaluation 55
4.4.1 Experimental setup 55
4.5 Implications for realistic scenarios 58
4.5.1 Comparison with other Secure Channel Protocols 59
4.6 Considerations on security 60
4.7 Conclusions and future work 61
Chapter 5 Summary and Conclusion 62
Chapter 6 List of Abbreviations 63
Chapter 7 Bibliography 65

List of Figures

Figure 1: Handshake in SCP03 7
Figure 2 Encrypt-then-MAC method [SCP CardLogic] 8
Figure 3: Pairing OPTIGA™ Trust M with host [Optiga Trust M] 10
Figure 4: Session key derivation and handshake between OPTIGA™ Trust M and host [Optiga shielded connection] 11
Figure 5: BLINKER domain separators 24
Figure 6: Strobe division of responsibilities [Strobe paper] 25
Figure 7: Strobe operations [Strobe paper] 27
Figure 8: Example of a legitimate JWT 36
Figure 9: Architecture with different modules for the connectivity and security 38
Figure 10: Example of an IoT hardware and software stack architecture 38
Figure 11: Organization of the Arduino MKR WiFi 1010 module 39
Figure 12: Attack schematics (a) and experimental setup (b) 41
Figure 13: Example of a JWT generated by the attacker with “iat” claim set to a date in the future 43
Figure 14: Handshake in SCP03 49
Figure 15: SCP03 encrypt then MAC applied to command PDUs sent from the host 50
Figure 16: High-level comparison of session management in SCP03 and NSCP 52
Figure 17: NSCP internal data on the host 53
Figure 18: NSCP internal data on the secure element 54
Figure 19: Throughput comparison of NSCP and SCP03 level-5 across various payload sizes (host: RPI, SE: ARM Cortex M4, bus: I2C) 56
Figure 20: Throughput comparison of NSCP and SCP03 across various payload sizes (host: RPI, SE: RISC-V@10MHz, bus: I2C) 57
Figure 21: MCU secure boot implemented supported by a secure element 58
Figure 22: Throughput comparison of NSCP and SCP03 level-3 across various payload sizes (host: RPI, SE: ARM Cortex M4, bus: I2C) 59

List of Tables

Table 1: Layout of RPMB 13
Table 2: RPMB data structure 15
Table 3: Benchmarks of NSCP vs SCP03 57
Table 4: Comparison of Secure Channel Protocols 60

Chapter 1 Introduction

This deliverable contains some material from the interim deliverable iD5.2 provided half way through the project and a lot of novel material. More precisely, Chapter 2 in this deliverable corresponds to Chapter 2 in iD5.2 and Chapter 3 in this deliverable corresponds to Chapter 3 in iD5.2 (with some reshaping of the content). Chapter 4 is completely new with respect to iD5.2, Introduction and Conclusion have been updated.

The ORSHIN project investigates open-source resilient hardware and software solutions for Internet of Things (IoT) security. In industrial environments, IoT security is crucial to ensuring data confidentiality, system integrity, and operational continuity. Many IoT devices handle sensitive information, such as pre-shared secrets or public key infrastructure (PKI) certificates used for authentication. Additionally, attackers can exploit vulnerabilities in sensors or embedded components to manipulate system behavior, potentially impacting the functionality of entire industrial plants. Protecting IoT devices from both remote and physical threats is therefore an essential aspect of security.

A widely adopted approach to improving IoT security is the integration of Secure Elements, specialized hardware components designed to resist tampering and to provide a protected environment for cryptographic operations. Secure Elements store cryptographic keys securely and ensure that sensitive data remains protected even in the presence of physical attacks.

One of the critical security challenges in this context is ensuring the confidentiality and integrity of intra-device communication. Specifically, the communication between Secure Elements and microcontrollers. If this communication is not properly protected, an attacker with physical or software access to the device could intercept sensitive information, manipulate authentication processes, or gain unauthorized access to critical resources.

This document reports part of the research conducted within Work Package 5 (WP5) of the ORSHIN project, specifically tasks 5.3 and 5.4. The work focuses on securing intra-device communication, analyzing existing solutions, identifying potential vulnerabilities, and proposing a new protocol designed to enhance security while maintaining efficiency and ease of implementation.

In Chapter 2, we present the contribution to the ORSHIN project about analyzing existing solutions, and then we propose a study of the different existing protocols that are used in commercial products to secure intra-device communication.

We start our analysis by illustrating one of the most common protocols for secure communication between a microcontroller and a secure element, the Secure Channel Protocol 03 (SCP-03), published by the GlobalPlatform consortium. Such a protocol has been the subject of a few revisions, due to security weaknesses found on previous versions. Another standardized protocol is the RPMB. It is used to secure memory accesses from a microcontroller to a specific memory peripheral. It is standardized by the JEDEC organization and implemented by most eMMC.

Not only standard protocols for secure communications do exist, but many manufacturers design and implement their own protocols. We explore them in the second part of the second chapter, for example analyzing Infineon's proprietary protocol to secure the communication between a host MCU and the OPTIGA Trust M security controller. Other manufacturers similarly opted for a proprietary protocol. For instance, Microchip, with the ATECC cryptoauthentication products, that use the IO protection key for securing the communication with the host microcontroller, or the Chip2Chip protocol used by the Ubxlib library in U-blox products. We analyze different aspects of their functionalities, performances and security in the following sections.

Another important avenue of research is provided by the protocols in scientific literature devoted to the so-called "Record Protocol", which aims at defining suitable protocols that target both usability, performances and security, while trying to provide a sufficient level of abstraction. A record protocol is very well suited to replace standard secure channel protocols, as it ensures both privacy and

authenticity of the communication, while preserving a transcript of the messages exchanged during the session. Among those, NaCl (pronounced “salt”) is probably one of the first efforts to design an all-rounded network security library, providing a high level abstraction from the record protocol to complex cryptographic primitives, without sacrificing speed and security. From that initial effort, further propositions have been made, and protocols like BLINKER, Strobe and the Noise have seen a wide adoption due to their flexibility. In particular the Noise protocol is used in the Whatsapp application to secure exchanges of billions of users every day.

We conclude our study with the analysis of two further constructions that can be of interest in order to provide an overview of the state of the art in terms of intra-device secure communications. These are the JSON Web Tokens (JWT), which provide a compact way to express claims to be transferred between parties. They are defined in RFC7519, and several further RFCs define the way to provide privacy and authenticity for the claims. Furthermore, we also analyze the properties of the Deck functions, where the authors present a way to make use of the sponge construction to create a versatile structure to provide the basic functionalities needed by a record protocol, with a very efficient, simple and elegant framework.

In Chapter 3, we address the identification of vulnerabilities in studied protocols, in line with the scope of the ORSHIN project to investigate software solutions for Internet of Things (IoT) security.

We identified a weakness in the JWT standard, showing an attack that exploits it. In particular an attacker that takes control of one device in the supply chain, may be able to create a series of valid JWTs, that may be used further after the deployment, to impersonate the device when accessing the cloud infrastructure. Among the advantages of the attack is that the network is completely unaware about the JWTs created in the supply chain by the attacker.

We call *JWT Back to the future* the possibility of using the device for preparing JWT that will be later used to connect to the cloud service. A malicious user in the supply chain, for instance someone having access to a testing machinery capable of interacting with all the devices, might have the possibility to collect a large number of JWTs for mounting a massive attack in the future, for example to flood the Cloud server, or bias their collected data.

This work has been submitted to the Lightweight Cryptography For Security & Privacy (LightSEC 2025) conference 2025, which falls in the scope of the ORSHIN project.

Finally, in Chapter 4, we explain how the ORSHIN project contributes with a novel secure communication protocol designed for industrial IoT environments, taking advantage of open source hardware, which is the focus area of the ORSHIN project.

It focuses on enhancing the security of the connection between microcontrollers and Secure Elements while improving efficiency compared to SCP03, the current industry standard. By leveraging the Xoodyak cryptographic primitive, our protocol, called NSCP, achieves strong security with significantly lower computational overhead, making it ideal for resource-constrained devices. This work has been published at the DATE2025 conference [Date25].

Chapter 2 State of the Art of Intra-Device Comms

This chapter analyzes the state of the art of the protocols for securing the communication between the main controller and the peripheral units, like sensors, Secure Elements, or other processors of a system. In particular, we focus on the security protocols that provide authenticity and privacy on the exchanged messages.

We start our analysis with a brief overview of the general steps that compose a secure channel protocol. Then we study the Secure Channel Protocol (SCP) used in most Secure Elements and published by the GlobalPlatform consortium. Our analysis then moves on to other proprietary protocols, like the ones used in Optiga products, by Infineon, and the secure protocol used in Microchip's ATECC chips. Afterwards, we illustrate the Replay Protected Memory Block (RPMB) protocol used to secure the communication with memories, and the Chip2Chip security protocol used in Ublox products. We then extend our analysis to other literature publications that tackle the problem of providing a robust and sound record protocol, as it is used in many modern applications worldwide. Finally we illustrate some further constructions that can be interesting to complete the scenario. Such constructions, like the JWT, show emerging ways to build and deploy security by using standardized basic blocks.

2.1 State-of-the-Art Protocols

In the following we analyze protocols that provide security for intra-device communications. We denote such protocols in general as secure channel protocols. A secure channel protocol plays the role of the so-called *record protocol* of TLS, in the sense that no key exchange is involved, but it is assumed that both ends of the communication share the same (set of) symmetric key(s), and the protocol is fast enough to allow efficient exchange of several messages between the parties. Thanks to this pre-shared secret, secure channel protocols typically begin with a handshake to establish temporary session keys. These session keys, derived from the pre-shared secret, are sufficiently diversified to prevent an attacker who compromises them from recovering the original secret. This ensures that future handshakes can re-establish security.

In general, before the record protocol, a key exchange mechanism is executed. Depending on the architecture, this key exchange can be performed on-line (with for example a Diffie-Hellman key exchange) or off-line (by using a physically secure environment to write the keys in the memory of the different devices involved). This document will not discuss this step, and in the following we assume that both components of the communication share the same secret(s), or pre-shared key(s).

The timeline of a secure channel session can thus be conceptually divided in four phases, occurring in the following order:

1. **Provisioning**, during which the pre-shared material is stored on both the communicating endpoints. The provisioning phase takes place in a secure and trusted environment.
2. **Handshake** or initiation, during which the communicating endpoints authenticate each other, and exchange the necessary data to perform the cryptographic functions required by the following phase. At the end of the initiation phase the session is considered to have been successfully established, and both entities are assumed to possess the same session keys to be used for the following phases.
3. **Data exchange**, during which the entities exchange data enforcing the security protections established during the initialization of the session.
4. **Termination**, when the session is closed by request of either entity.

2.1.1 The Secure Channel Protocol Family

GlobalPlatform [GlobalPlatform] is a collaborative organisation driven by consensus, and dedicated to the standardisation of isolated execution environments in different types of devices, to deliver secure services and trusted storage for diverse industries and stakeholders. The members of this organisation are dedicated to facilitating the efficient initiation and oversight of innovative, securely designed digital services and devices. The objective is to provide users with end-to-end security, privacy, simplicity, interoperability, and convenience through these services and devices.

GlobalPlatform spans various industries, issuing specifications outlining the procedures for post-issuance management of smart cards, including the capability to securely manage the content of cards remotely. At the core of these mechanisms lies the family of Secure Channel Protocols (SCPs) designed to protect bidirectional communication between a smartcard and a host. These protocols are used as mutual authentication and they provide cryptographic protection for card and host subsequent communication, supporting entity authentication, as well as integrity, authenticity, and confidentiality of the payload.

The family of the Secure Channel Protocol (SCP) span a wide set of usages, ranging from a record protocol for the encryption of communications over simple buses (like Uart or I2C), to key exchange protocols, or protocols to secure the communication between a host and a far away secure element, by encapsulating the messages over a high level application protocol.

SCP is built above ISO/IEC 7816, a standard used to represent command / response messages as application program data units (APDUs) (see Section Packet Format).

The SCPs family comprises the protocols SCP01, SCP02, and SCP03 [Card Specification] which describe symmetric keyed ciphering mechanisms. The deprecated SCP01 used DES encryption [DES], and SCP02 uses Triple DES in CBC mode with fixed IV of binary zeros. As a result, the encryption scheme is deterministic and it was revealed to be vulnerable to classical plaintext-recovery attacks [SCP02 Attack]. SCP02 is now deprecated and the use of SCP03 [SCP03] is recommended.

Secure Channel Protocol '03' includes services similar to Secure Channel Protocol '02', however, it is based on the Advanced Encryption Standard (AES) [AES] encryption algorithm with a randomly generated Initialization vector, resulting in a non-deterministic and secure algorithm as opposed to SCP02.

The GlobalPlatform organisation defines two SCP channels relying on asymmetric-key cryptography: SCP10 [Card Specification] and SCP11 [SCP11]. Asymmetric cryptography usually leverages on wrapping symmetric session keys with asymmetric encryption or key negotiation.

The Secure Channel Protocol '10' offers authentication services using an RSA-based Public Key Infrastructure (PKI) and secure messaging protection using symmetric cryptography, while SCP11 uses ECC-based cryptography and secure messaging protection based on SCP03.

Depending on the real world use case, asymmetric-based protocols could be necessary if there is no way to set up pre-shared secrets between the parties involved.

Concerning SCP02 and SCP03, they both use a counter for the generation of new session keys. For instance, if the counter is 2-bytes long, then after 2^{16} sessions the long-term secret key must be changed. In SCP10, no counter is required, as the process of key generation/derivation only uses freshly generated random bytes without maintaining any stateful information.

However, multiple issues were discovered in SCP10 ([SCP10 Flaws]), and SCP11 is not broadly adopted due to its complexity.

Other versions of SCP include the following:

- **SCP21** [Card Specification] focuses on enforcing privacy following the requirements defined in CEN/EN 419 212 [CEN/EN 419 212]. SCP21 defines two steps: Password Authentication Connection Establishment (PACE) and modular Extended Access Control (mEAC).

- **SCP22** [Opacity Secure Channel] is a secure channel and key establishment protocol, collectively known as the Opacity Secure Channel establishment method.
- **SCP80** supports the Over-The-Air security scheme defined in ETSI TS 102 255 [ETSI TS 102 225] and ETSI TS 102 266 [ETSI TS 102 226]. The protocol uses DES, which is not secure, therefore the use of AES is recommended.
- **SCP81** [Remote Application Management over HTTP] supports an Over-The-Air security scheme based on HTTP and Pre-Shared Key TLS protocols.

2.1.2 Secure Channel Protocol 03 (SCP03)

Among the SCP family, the Secure Channel Protocol '03' is the current state-of-the-art protocol for intra-device communications. The protocol provides decryption and MAC verification for incoming commands, and encryption and MAC generation on card response. SCP03 provides security guarantees, resistance to replay, out of order delivery and protection against algorithm substitution attacks.

The SCP03 protocol supports different security levels for the subsequent command APDUs (C-APDUs) and response APDUs (R-APDUs).

The security level corresponds to different combinations of message authentication and encryption enforcement. Encryption is performed using AES in Cipher Block Chaining (AES-CBC) [AES-CBC] mode while authentication is achieved by appending an 8-byte (or, in some versions, 16 bytes) Message Authentication Code (MAC) produced by an AES-based CMAC [AES-CMAC].

The security mechanism combinations are listed below (being "C" command and "R" response).

- C-Encryption, R-Encryption, C-MAC, R-MAC
- C-Encryption, C-MAC, R-MAC
- C-MAC, R-MAC
- C-Encryption, C-MAC
- C-MAC
- Plaintext with no authentication

2.1.2.1 Provisioning

During the provisioning phase, the security domain is initialised on both the endpoints (the card entity and the host). In order to initialise it, two AES-128/AES-192/AES-256 keys, K-enc and K-mac, are loaded into the memory of each endpoint. Later, these master keys are used to derive three session keys required to establish a secure channel session, during the Handshake phase.

As opposed to the session keys, the master keys do not change across multiple sessions, unless the host entity performs a "key rotation" procedure. Key rotation involves using the master keys presently stored (or provisioned) on the endpoints to securely encapsulate the new keys. Following this process, both endpoints possess the same new master keys, while the old ones are discarded.

2.1.2.2 Handshake

The authentication of an off-card entity is accomplished by initiating a secure channel session. During such initialization, dedicated session keys, valid only for the current session, are exchanged between the two parties. In the event of any failure in the authentication process, it requires a restart.

The authentication of the off-card entity is applicable only until the termination of a secure channel session, and holds validity solely for the messages exchanged within that specific secure channel. This secure channel session pertains to both the establishment and termination phases of a secure channel.

This procedure enables the host to communicate to the card the required security level for the ongoing session, specifying whether integrity and/or confidentiality are necessary, and applying this decision to all subsequent exchanged messages.

The handshake assumes that the off-card entity and the card entity share the 2 static keys, previously previsioned. These are used to derive the session keys, denoted as S-enc, S-mac, and S-rmac, respectively.

The process consists of the following steps.

1. The channel is initiated by the off-card entity which sends a specific command (*initialise update*) that carries a host challenge v-h, i.e. a random data unique to this secure channel session.
2. The card, on receipt of this challenge, generates its own card challenge v-s.
3. The card, using the challenges and its internal static pre-shared keys, creates new secret AES secure channel session keys. A CMAC-based key derivation function (KDF) is used [AES-CMAC].
4. Then, the card employs S-mac to produce an initial cryptographic value known as the *card cryptogram,* x-s.
5. This card cryptogram x-s and the card challenge v-s are transmitted back to the host.
6. Thanks to the information received from the card, the host can generate identical secure channel session keys and the corresponding card cryptogram x-s to those of the card. Also, by comparing the cryptogram internally generated against the one received, the host can authenticate the card.
7. The off-card entity employs a comparable process to generate a second cryptographic value known as the *host cryptogram*, x-h, which is then transmitted back to the card with the *external authenticate command.*
8. The card should be able to generate a copy of the host cryptogram and compare it to perform the off-card entity authentication.

After the handshake, both the communicating entities can use the session keys to enforce the security level. The S-enc and S-mac are used to respectively compute the encryption and the mac of transmitted messages, while S-rmac is used to protect the transmission of new static keys.

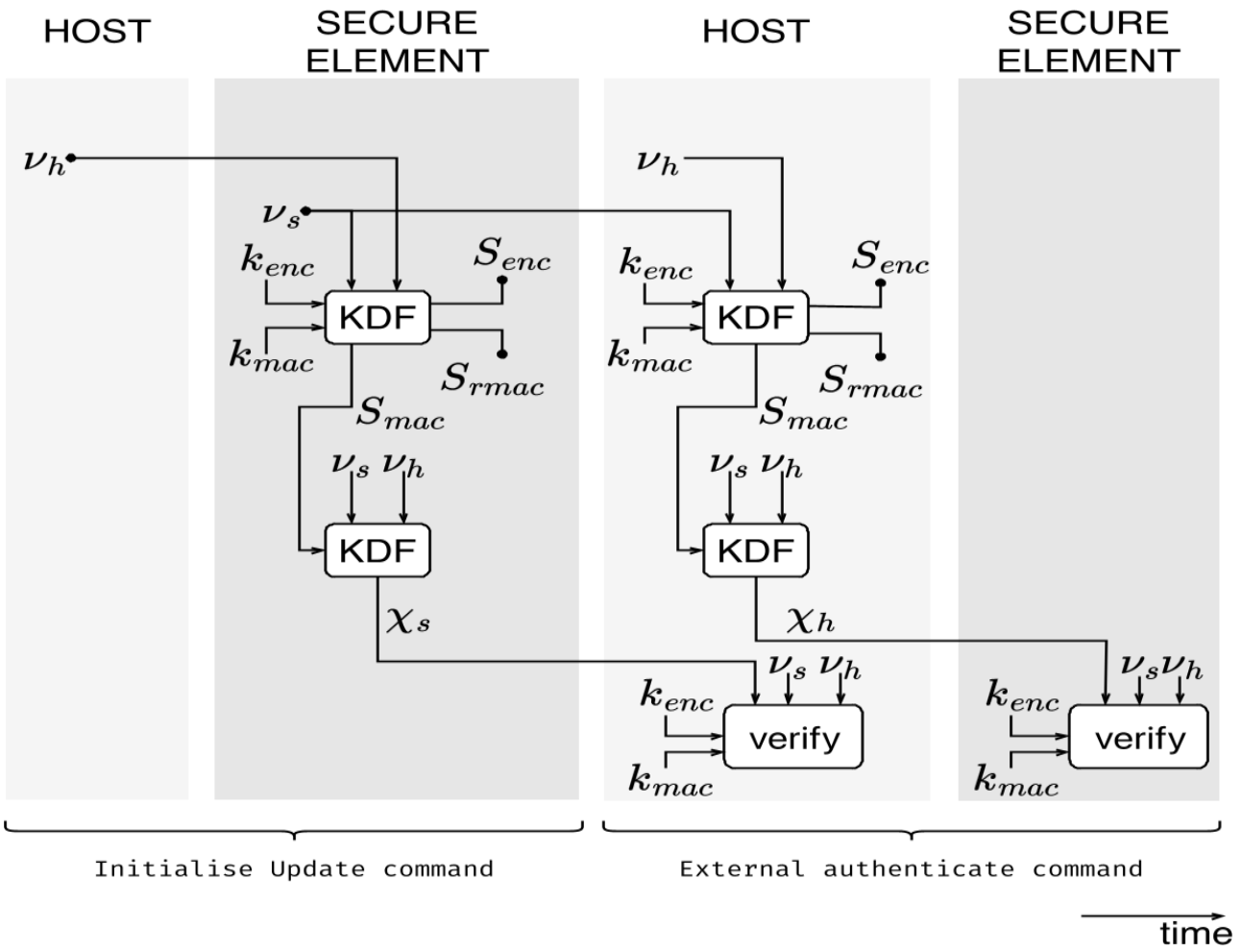


Figure 1: Handshake in SCP03

2.1.2.3 Data exchange

During the data exchange phase, the session keys created by the two parties during the handshake phase are used to secure the exchange of messages. Various security properties are applied to the exchanges, depending on the requirements specified in the handshake by the off-card entity.

The *external authenticate command* is also used by the host to specify one of the five security levels of the following communication:

1. Level 1: authentication of commands.
2. Level 2: encryption and authentication of commands
3. Level 3: authentication of commands and responses.
4. Level 4: encryption and authentication of commands and authentication of responses.
5. Level 5: encryption and authentication of commands and responses.

Message **authentication** is achieved by comparing the C-MAC received from the Host with the C-MAC computed by the card. The C-MAC is generated by applying the NIST CMAC calculation [NIST SP 800-38B] using the S-MAC session key generated in the handshake step.

To provide integrity of the command sequence, the 16-byte C-MAC of a command is used as input for computing the C-MAC of the subsequent command. The final 16-byte C-MAC calculated is preserved as part of the channel state, initially set to '00' for the first computation. This mechanism serves to reassure the card that every command in a sequence has been received.

Additionally, the integrity of the response is chained to the integrity of the command sequence by employing the same 16 bytes as input for computing the R-MAC on responses.

When message data confidentiality is required, the message data field is encrypted using an encryption key derived from the session keys generated during the initiation process. Encryption is applied across the entire data field of the command message to be transmitted to the card. If required, the encryption is also applied across the response transmitted from the card.

SCP03 relies on the Encrypt-then-MAC method, meaning that the plain-text is first encrypted with AES-CBC, then the MAC is computed on the ciphertext, and lastly the MAC is appended to the ciphertext. If message data confidentiality is also required, the C-MAC applies to the message data field after the encryption has been performed.

The maximum number of messages per session adheres to the rules set by AES-CBC and AES-CMAC. In fact, it is crucial to ensure that the initialization vector used for encryption or MAC calculation does not repeat. In fact, the initial chaining value (ICV) used by AES-CBC, iv(n), depends on the current message counter n. The counter increases with each command sent from the host to the secure element, making it dependent on the number of commands sent n. Using iv(n), identical payloads within the same session will be encrypted differently, preventing chosen plain text attacks (CPA).

For message authentication, authentication tags are generated using MAC chaining values. At any moment, the MAC chaining variable in the host (μ in Figure 2) guarantees the integrity of the command sequence produced by the host. In a way, it can be thought of as a summary of the session's history. The authentication tag α associated with the message is the most significant 8 bytes (or, in some versions, 16 bytes) of the current chaining value μ which is computed from the previous one with the current command ciphertext and the S-mac key. Using such a chaining value captures the entire command history up to message n.

This effectively nullifies attempts at replay attacks, as two identical commands or responses will have different authentication tags. Thanks to its design, SCP03 has been proven secure against replay attacks, out-of-order attacks, algorithm substitution attacks, and more [SCP CardLogic] [SCP Cryptanalysis].

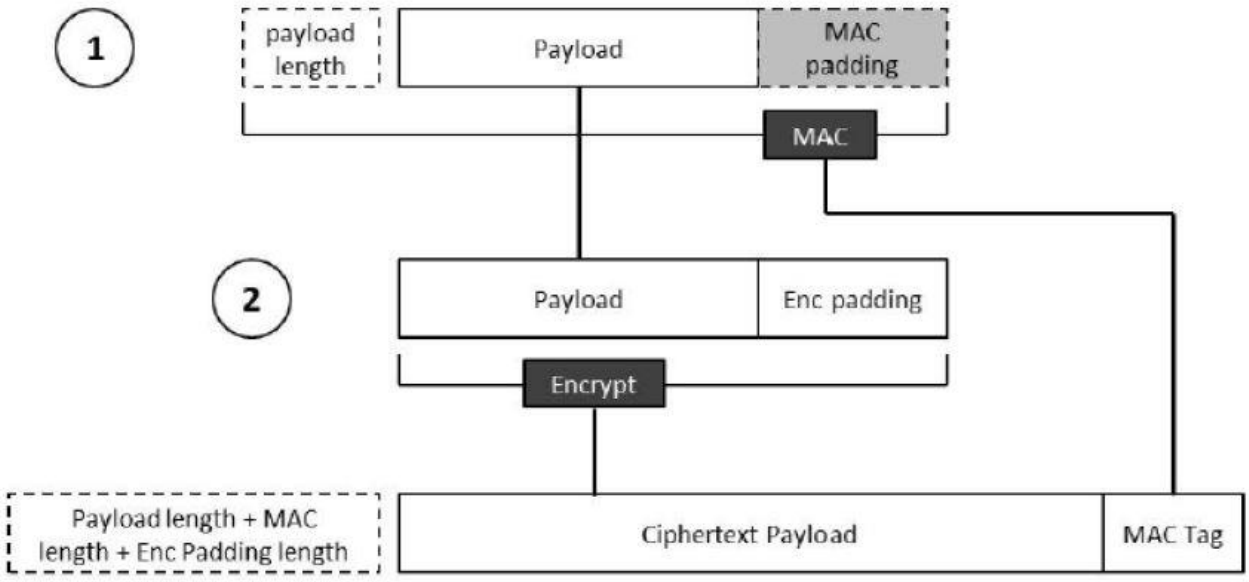


Figure 2 Encrypt-then-MAC method [SCP CardLogic]

2.1.2.4 Packet Format

SCP03 packets adhere to the format defined by the ISO 7816 standard [ISO 7816], specifically designed for communications with smartcards. This format, namely ISO7816 APDU, is organized as a TLV, where each packet consists of a Tag, Length, and Value. The Tag uniquely identifies the type of data, the Length field specifies the size of the accompanying Value, and the Value itself contains

the actual data. This standardized structure ensures consistency and interoperability across communication interfaces. SCP03 supports many different communication channels (including i2c and uart); in general, this protocol supports all the mechanisms supported by ISO 7816 standard.

In particular, during the handshake, the packets contain specific information in the header required to set up a secure channel (e.g.: security level), as described in section Handshake. The body of the packet, on the other hand, contains the cryptographic material required by the protocol handshake to establish a secure connection. In essence, SCP03 handshake and message exchange take advantage of the structured ISO7816 format to encapsulate secure communication, ensuring the integrity, confidentiality, and authenticity of the transmitted data during the entire session between the host and the card.

The packets exhibit four distinct formats, enumerated below, each tailored to specific contexts:

- **Case 1:** `CLA, INS, P1, P2`
- **Case 2:** `CLA, INS, P1, P2, Le`
- **Case 3:** `CLA, INS, P1, P2, Lc, Data`
- **Case 4:** `CLA, INS, P1, P2, Lc, Data, Le`

Where

- `CLA` = Operation class
- `INS` = Instruction
- `P` = Parameter
- `Le` = Expected length to be received from the card
- `Lc` = Length of the Data field
- `Data` = Message sent to the card

The exact content of such fields depends on the protocol and on the instruction-set of the card.

2.1.2.5 Implementations

Unfortunately, GlobalProtform does not offer a reference implementation for their protocol. Instead, during our study, we found several implementations of the SCP03 protocol from individuals. Probably the de-facto standard implementation of the SCP03 (but of all the GP suite in general), is the one from Martin Paljak: https://github.com/martinpaljak/GlobalPlatformPro/. It is not dedicated to SCP03, but in order to load applets into JavaCards, it needs to speak SCP03 to load the applet into the card. It also comes with some test vectors, which comes in hand when implementing SCP03 from scratch.

2.1.3 OPTIGA

OPTIGA is a family of security solutions designed by Infineon, for integration into embedded systems to protect the confidentiality, integrity and authenticity of information and devices. The OPTIGA Trust family includes products for smaller platforms as well as programmable solutions, such as OPTIGA™ Trust M [Optiga Trust M]. OPTIGA™ Trust M implements a connection protocol that provides a layer of security over the I2C channel [Optiga shielded connection].

2.1.3.1 Provisioning

The protocol for ensuring secure connection between a host microcontroller and an OPTIGA Trust M is based on establishing a Pre-Shared Secret (PreSSec), also known as platform binding secret, between them. Then a secret key is derived from the PreSSec and used to protect the subsequent connections. The process is described in Figure 3.

1. To generate the shared secret, first the host generates a random number by using a Pseudo random number generator (PRNG), then uses it to derive the PreSSec. The host can further optionally XOR it with a second random number.

2. The PreSSect is then stored in nonvolatile memory (NVM) by the host.
3. Once the PreSSec has been generated, it is sent to the OPTIGA™ Trust M for storage.
4. The host sets the metadata of the PreSSec stored on the OPTIGA™ Trust M in a secure environment for integrity and confidentiality protection. Once the PreSSec is written in OPTIGA™ Trust M, it can be updated only via a secure protected update mechanism.

The host and OPTIGA™ Trust M now have the same PreSSec. The recommended secret size is 32-bytes, or at least 16-bytes.

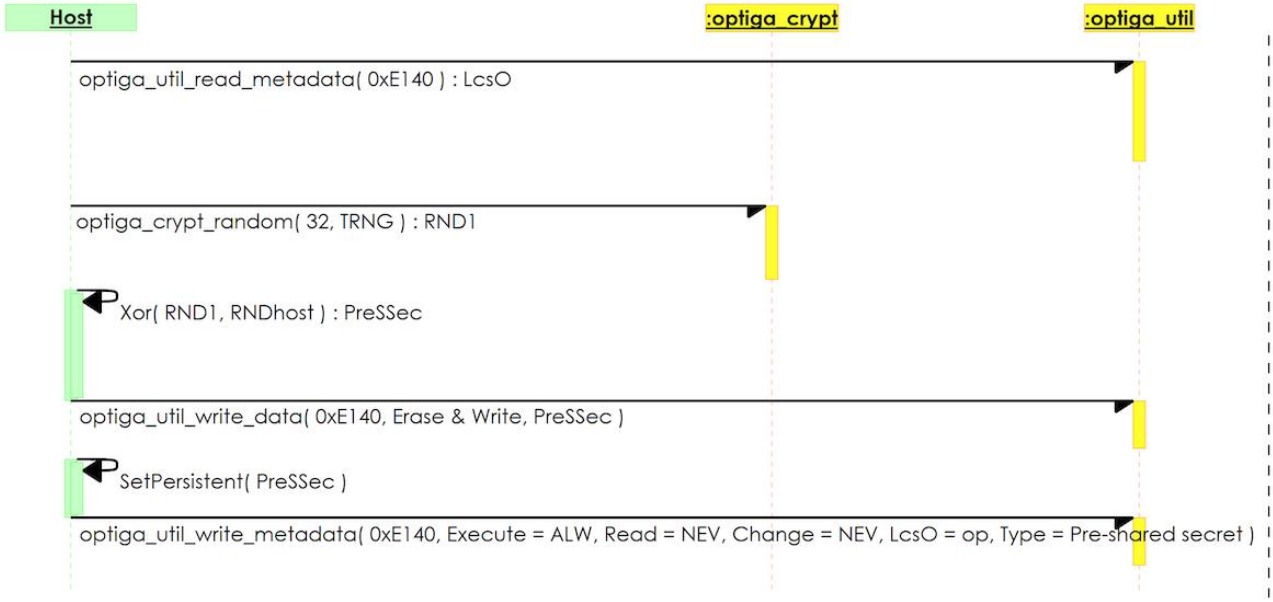


Figure 3: Pairing OPTIGA™ Trust M with host [Optiga Trust M]

2.1.3.2 Handshake

Once the PreSSec is established, it is used to derive the symmetric key for encrypting the traffic between the host and OPTIGA™ Trust M. The PreSSec is transformed into a symmetric key based on the TLS pseudo random function (TLS PRF) SHA256 [TLS 1.2, FIPS 180-4].

The session key is derived every time a shielded communication is established between the host and OPTIGA™ Trust M and therefore, unique on each startup or each session. The following steps, also illustrated in Figure 4, are involved in the session key derivation using the PreSSec previously established.

1. The host sends the supported protocol versions to the Trust M with a particular Security Control packet to the presentation layer. This indicates the starting of the handshake process.
2. The Trust M generates a random value (RND) and the Slave sequence number (SSEQ), used to avoid replay attacks.
3. The slave returns both values along with the chosen protocol version to the host.
4. The host computes the session secrets (MasterSsec) and so does the Trust M (SlaveSsec) by using the TLS PRF. This function takes as input a nonce and the PreSSec, and derives the Session Key.
5. The host uses the SSEQ to generate the ciphertext by using the Generation-Encryption Process as per AES128-CCM8. The host sends the SSEQ and the ciphertext to the Trust M, indicating that the host-side handshake is finished.
6. The Trust M decrypts the ciphertext using the Decryption-Verification Process of AES128-CCM8 and verifies if the expected values were received. If it is the case, the Trust M saves the SSEQ for further use.

7. Repeating the steps of the host, the Trust M generates the Master Sequence Number (MSEQ) and the ciphertext with the Generation-Encryption Process and sends both to the host.
8. The host decrypts the ciphertext and verifies in the payload if the expected values were received and upon success, saves the received MSEQ for further use.
9. If both ciphertext validations are successful, a cryptographic link is established between the two entities, allowing a protected record exchange.

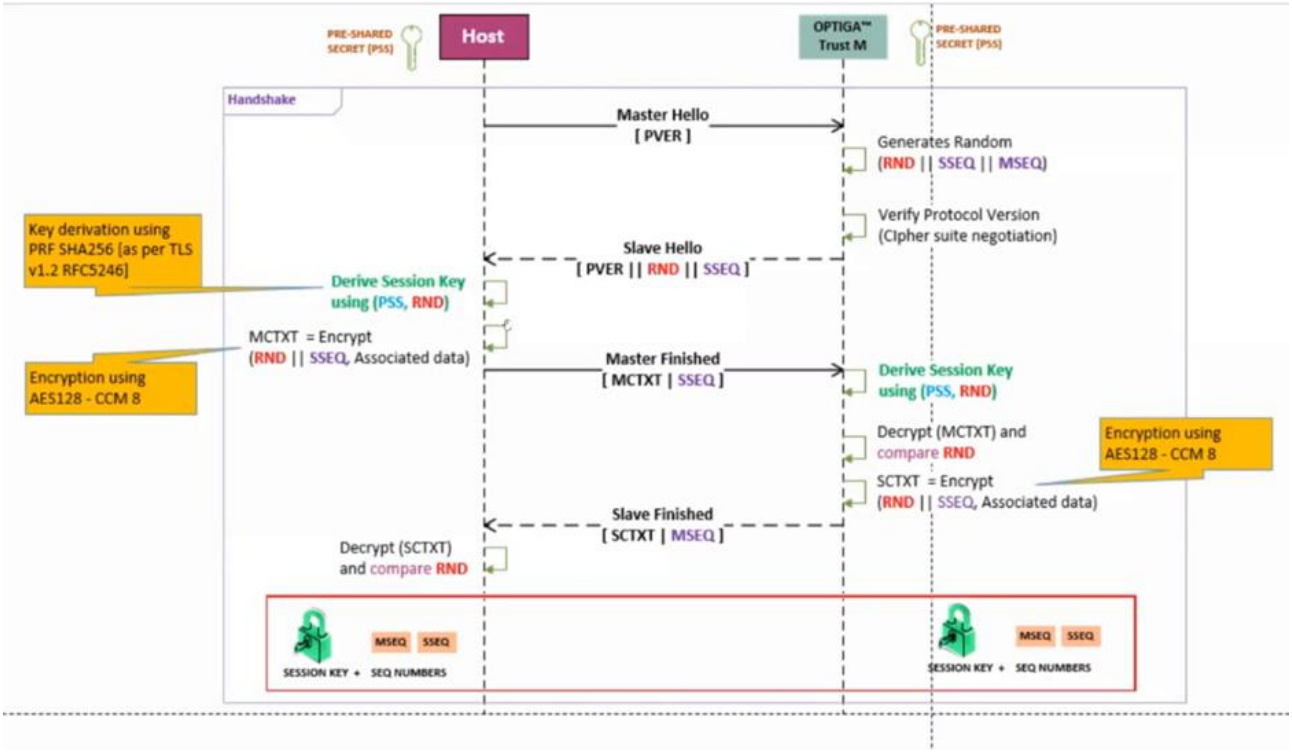


Figure 4: Session key derivation and handshake between OPTIGA™ Trust M and host [Optiga shielded connection]

2.1.3.3 Data exchange

After deriving the session keys from the PreSSec, they are used for encryption and decryption. The used scheme is AES128-CCM8 [NIST SP 800-38C].

The same key is also used for authentication using MAC. This kind of scheme where both encryption, decryption and authentication are done using the same key is called Authenticated encryption.

2.1.3.4 Implementations

Infineon does not provide the source code for the client side of the communication, instead they provide some example code for interacting with Optiga Trust M, with the “shielded connection” over I2C enabled here: https://github.com/Infineon/mtb-example-optiga-crypto/tree/6f712e2de35d4aa7203d963bf6cd9401a1ca0223.

2.1.4 ATECC

The Microchip CryptoAuthentication™ [CryptoAuthentication™] is a family of high-security cryptographic devices that combine hardware-based key storage with hardware cryptographic accelerators to implement various authentication and encryption protocols.

The family of Microchip **ATECCx08A/B** CryptoAuthentication Devices are crypto engine authentication devices with a flexible command set that include an EEPROM array for storage.

Typical applications include node identity authentication and session key creation and management. They support the entire ephemeral session key generation flow for protocols including TLS 1.2 and TLS 1.3.

The main Secure Elements belonging to the CryptoAuthentication family are the ATECC608A [ATECC608A] and ATECC608B [ATECC608B]. These devices provide confidentiality, data integrity, and authentication to systems with MCU or MPUs running encryption/decryption algorithms.

The Microchip **ATECC608A/B** introduced a method to protect the I/O transmission between the device and the host MCU for some of the commands. The improvement was necessary as the first version of the protection mechanism, as implemented in the **ATECC508A** [ATECC508A], was discovered not to be secure. The verification of the ECDSA signature returns an encrypted boolean result: a value of zero if the signature of the message can be verified using the public key, and a value of one if the signature does not match, or another error code if there is some form of parsing or execution error. Therefore, no security in the verification command is employed.

The new method to protect the communication channel is the following: an IO Protection key is used to protect I/O transmissions between the device and the host MCU. The IO Protection key feature can be used in the ECDSA signature verification, ECDH, secure boot, and the KDF commands to encrypt parameters and validate responses by using a MAC. This allows to protect the communication on the physical I2C bus against man-in-the-middle attacks.

2.1.4.1 Provisioning

To activate the I/O security feature, a pairing procedure between the MCU and ATECC608A/B must be carried out on first boot. The pairing happens by establishing a unique randomly generated secret key and shared between the host MCU and the device, called the IO protection key.

First, the MCU randomly generates a 32-byte IO protection key using a random command and saves it in its internal Flash. Then, the MCU writes the IO protection key to the IO protection key slot, which is locked to make the IO protection key permanent. As a pairing check, the MCU could use the MAC command to issue a challenge to the IO protection key to check if the key stored in Flash matches the IO protection key stored in the ATECC608A/B.

Once established, the Protection key can be used to encrypt the Premaster key generated from ECDH and the generated KDF key, before being sent to the host. In the secure boot process and the signature verification process, a MAC is generated using the IO Protection key to provide additional authentication to the host. After provisioning the two entities, no real handshake is performed. Both the entities own the same IO Protection key with which the material to be shared is encrypted.

2.1.4.2 Security history of the ATECC

The first version of the ATECC608A family was found insecure to laser fault attacks [ATECC608A-Laser]. In particular, the attackers exploited the *self-test* procedure of the chip to fine tune their laser injection. Subsequently, by using a double-fault attack, it was shown possible to bypass the security countermeasures of the chip.

In 2020 Microchip developed a security-enhanced version of the ATECC608A, known as the **ATECC608B**. The ATECC608B has been designed to allow an easy migration from the ATECC608A, while improving the overall security of the devices, continuing the line of products developed as part of the Microchip CryptoAuthentication family. The ATECC608B uses the IO protection key to encrypt the output of the KDF command and to encrypt newly derived keys back to the host. It can also be used as the encrypted read key for all session keys.

All applications and use cases previously supported by the ATECC608A are also supported by the ATECC608B.

2.1.5 Replay Protected Memory Block (RPMB)

Not only devices dedicated to cryptographic computations need secure connections. Also memories may be enhanced by such a mechanism to improve the security of the stored data.

A Replay Protected Memory Block (RPMB) [RPMB] is a small partition within a device that allows storing data to the specific memory area while guaranteeing authentication and protection against replay attacks. RPMB was introduced in eMMC version 4.4, and it is available on other flash-based storage devices among which UFS (Universal Flash Storage) [UFS] and NVMe [NVMe]. The RPMB interface is defined by the JEDEC organisation [JEDEC] and can be accessed with a specific and standardized security protocol that has its own commands and data structures.

Table 1 shows the layout of the RPMB partition within the device storage.

Table 1: Layout of RPMB

Section	Access	Size	Initial value
Authentication Key	Write once. Not erasable or readable	32 bytes	Authentication key register which is used to authenticate accesses when MAC is calculated
Write Counter	Read-only	4 bytes	0x0000_0000
RPBM Data Area	Read/write	Multiple of 128Kb, 16MB max	This data may be overwritten by the host but can never be erased.

RPMB makes use of symmetric key authentication, where an authentication key is used by both the host and the device to exchange authenticated data. The process is the following:

- An authentication key is first programmed by the host to the storage device. This process must take place in a secure environment.
- Data to be written to the device is hashed and signed with the authentication key, and the storage device will only accept the write operation after checking the signature, which is computed through MAC.
- When reading from the device, the data is returned together with the MAC, so that the host can also calculate the MAC and compare it with the one received to verify the authenticity of the message

2.1.5.1 Provisioning

The Authentication key is a 256-bit key programmed into the One-Time Programmable (OTP) area of the storage device. This key can only be programmed once in the device lifetime, and it is invisible to any software after it is programmed. This Authentication key must be created in a secure environment like in an OEM production, written to the RPMB device, and securely stored in the target platform. Therefore, a secure storage is recommended for the proper usage of the RPMB partition.

The message to be sent to the device for initialization of the Authentication key is composed as follows:

1. The Authentication key programming is initiated by sending a Security Protocol Out command with Request Message Type = 0001h and the Authentication Key.
2. The device receives the command and returns a confirmation message.
3. The Authentication Key programming verification is issued by a Security Protocol Out command with Request Message Type = 0005h.
4. The device receives it and returns a confirmation message.
5. The host retrieves the verification result by issuing a Security Protocol In command.
6. The device returns the RPBM data frame with Response Message Type = 0100h and the Result code. If the programming of the Authentication Key fails then the returned result is 0005h, meaning write failure. If some other error occurs during Authentication Key programming then the returned result is 0001h, meaning general failure.

2.1.5.2 Data exchange

After the initialization of the key, when reading data from the partition, the replay protection protocol verifies the counter to confirm that the retrieved data is not a replay. If the counter is valid, the protocol generates a MAC using the same encryption keys algorithm employed in the write operation. Subsequently, the protocol compares this MAC with the one generated during the write operation to guarantee the integrity and authenticity of the data throughout the reading process.

The MAC is calculated using HMAC SHA-256 [HMAC-SHA] and it is used to authenticate all the read and write operations accessing the secured area. The HMAC SHA-256 calculation takes as input the secret key, the counter, which counts the total number of writes to the RPMB, and a message. The resulting MAC is 256 bits long, and it is embedded in the data frame as part of the request or response.

The key used for the MAC calculation is the 256 bit Authentication Key. Input to the MAC calculation is the concatenation of the fields in the RPBM message data frame excluding stuff bytes and the MAC itself.

Without the RPMB key, read access is still possible but without the guarantee of data integrity and authenticity. That also means anyone can read the RPMB, so the RPMB does not provide data confidentiality, as encryption should be done by software if necessary.

2.1.5.3 Features

RPMB features the following commands for the request from the memory:

1. Authentication key programming request
2. Reading of the Write Counter value request
3. Authenticated data write request
4. Authenticated data read request
5. Result read request

The message types for the response are the following. The response type corresponds to the previous Replay Protected Memory Block request.

1. Authentication key programming response
2. Reading of the Write Counter value response
3. Authenticated data write response
4. Authenticated data read response

Table 2 shows the data frame structure of RPMB.

Table 2: RPMB data structure

Bytes count	Content
196	Stuff bytes
32	Key/MAC
255	Data
15	Nonce
4	Counter
2	Address
2	Block
2	Result
2	Request/Response

Using the commands reported above and enforcing the sequential MAC calculation, RPMB is capable of denying replay attacks and grants integrity of the read and write operations transparently.

2.1.5.4 Implementations

Microchip, similarly to Infineon, does not provide the source code for the client side of the secure communication with the ATECCx08 family of devices, instead they provide the CryptoAuthLib code: the APIs required to communicate with Microchip Security device, available at https://github.com/MicrochipTech/cryptoauthlib.

2.1.6 U-blox

The U-blox host library, called ubxlib [ubxlib], provides C libraries to build embedded applications with the aim of providing a unified and thoroughly tested solution, complete with examples. The library is delivered as an add-on to chosen popular microcontroller to facilitate integration of connectivity, security, and localization into embedded applications, enabling functionalities such as network connection, TCP socket opening, location establishment, etc.

Embedded systems often incorporate an AT interface that facilitates the serial communication between a microcontroller and modems or GMS modules. The U-blox host library defines a chip to chip (C2C) protection security feature [ubxlib_C2C], which provides confidentiality, integrity and authenticity to the communication channel between the AT interface and the MCU. The ubxlib library also provides some sample code to demonstrate the use of the C2C security feature.

2.1.6.1 Provisioning

AT commands, parameters, command outputs, and all associated data undergo encryption using a pre-generated encryption key produced in the manufacturing process and transmitted to the MCU,

establishing the Root of Trust (RoT). The initial pairing occurs once, during device production, and the keys derived from the RoT can be utilised in each subsequent session.

Before initiating the pairing process, it is crucial to verify that the process will take place in a secure environment, preferably conducted in the factory. To ensure this, the module restricts chip-to-chip security pairing until before it has undergone security bootstrapping. This bootstrapping occurs precisely when the module makes its initial contact with the network. More precisely, the following steps must be meticulously followed in the specified order:

1. Execute the chip-to-chip pairing process between the MCU and the module. It is important that the MCU securely stores the pairing keys for future activation or deactivation of chip-to-chip security.
2. Allow the module to establish its initial network connection, during which it will undergo bootstrapping with U-blox security servers.
3. Conclude the security sealing process.

Once completed, the MCU gains the flexibility to activate the chip-to-chip security at any subsequent time. If re-pairing is necessary, authorization can be granted through the REST API.

2.1.6.2 Handshake

The C2C mechanism defines 3 values used to establish, enable and disable a C2C secure connection: pTESecret, a secret, an AES Key, and a SHA256 HMAC Key. Data is padded before encryption with the PKCS#7 padding [RFC 5652]. As a consequence, the maximum supported data length is 256 bytes. The protocol uses an overhead of 6 bytes as Start and Stop flags, 2-byte length and 2-byte CRC, and the length of the IV is 16 bytes.

A past version of the C2C security feature used AES128 in CBC mode and relies on the MAC-then-encrypt method. U-blox, later deemed such an implementation as "prone to security attacks" [SARA-R5 "00B"]. Thus, in further releases, a different cipher suite which provides stronger security was implemented by still employing the AES128 in CBC mode, but with the encrypt-then-MAC approach.

The pairing between the microcontroller and the AT cell is carried out by the *uCellSecC2cPair* function. The function takes 4 arguments on input:

1. CellHandle: Handler to communicate with the AT module
2. pTESecret
3. pkey: the resulting AES key
4. pHMAc: the resulting SHA256 HMAC key

The handshake performs the following steps:

1. Connect to the AT module
2. Write string pTESecret to AT handle
3. Read back the AES Key and HMAC Key
4. Read the 16 bytes c2c confirmation TAG
5. Encrypt the TAG. The function for the encryption of the TAG takes as input the confirmation TAG, the pTeSecret, the pKey, the pHMACKey, and the outputBuffer and does the following computations:
    - Get a random 16 bytes IV and write it to the encryption buffer. Copy the 16 bytes TAG, padded with 16 bytes padding, to the encryption buffer.
    - Encrypt the encryption buffer with AES-128 CBC with key pKey to the output Buffer, obtaining the following structure:

      16 bytes of IV || 32 bytes of encrypted padded TAG

    - Apply HMAC SHA2 with key HMACKey and truncated MAC to 16 bytes, obtaining:

16 bytes of IV || 21 bytes of encrypted padded TAG || 16 bytes of pTeSecret

- Obtain the final result, composed by:

  16 bytes of IV || 16 bytes of enc. pad. binary TAG || 16 bytes of truncated MAC

6. Send the pTESecret back and the encrypted TAG to the module.

2.1.6.3 Data exchange

The previous pairing process is run only once: C2C sessions are simply opened and closed using the stored keys. The function *uCellSecC2cOpen* defines the session opening operations and it takes 4 arguments on input:

1. CellHandle: Handler to communicate with the AT module.
2. pTESecret.
3. key.
4. pHMAc: the resulting HMAC key.

It performs the following steps:

1. Connection to the AT module.
2. Writing of the string pTESecret to the AT handle.
3. Setting the c2c context with the pKey and HMAC Key.
4. Setting hooks for Send and Receive to be encrypted/decrypted.

2.1.6.4 Security concerns

During our review, we noticed a few points that need particular attention to be paid when integrating such a security protocol.

- The code provides a poor random implementation, to be overridden by the application. However, the sample code for using the C2C does not provide a better implementation, nor warns about it. Therefore, an unaware implementer may accidentally use such a random implementation, provided below for reference:

```
1. int rand()
2. {
3.   uint32_t answer;
4.
5.   while ((answer = sys_rand32_get()) > RAND_MAX) {}
6.
7.   return (int) answer;
8. }
```

For example, we note that in the above implementation, a possible problem or malfunction of the hardware generator is not handled, nor are there error codes returned. So for example if the hardware random number generator is stuck and always produces the same value (thus it is not random), the user would not be aware of it.

- During pairing, the AES Key and HMAC Key are sent in clear over the insecure channel. This is suggested to be done in a secure environment. However, it is not clear that if a malicious user tries to rePair after a Cell is already paired, the latter do not again reply with the AES Key and HMAC Key.
- There seems to be no session keys, only master keys used for each encryption/mac each time the same keys are reused, so they are more exposed to side-channel or faults, or other cryptanalytic attacks in case of IV collisions.
- There is no assurance about IV collisions, it is only specified that IVs are randomly generated.
- There seems to be no tracking of the session counter or an incrementing counter for each message after opening a session. This means in particular that an attacker should be able to replay messages (despite their content remaining encrypted).
- It is not clear what happens if the Pairing process is re-tried.

Such problems have been notified to U-blox as github issues, on the official ublox's github page, as issues #219 (https://github.com/u-blox/ubxlib/issues/219) and #220 (https://github.com/u-blox/ubxlib/issues/220). Issue 219 has been closed as it applies to the Zephyr project, and latest versions solve the issue. Issue 220 has been closed as the chip to chip security feature was removed back in commit 2639236, mid 2023.

2.1.6.5 Implementations

Ublox, provides its ubxlib that provides examples and code to connect to its devices. The library is available at https://github.com/u-blox/ubxlib. However, since the beginning of this project, Ublox sold its cellular IoT branch, and consequently, the public will be archived on 2024-11-08.

2.2 Secure Channel Protocol Frameworks

Designing or implementing secure protocols from scratch is generally not recommended and can be difficult for software developers, as many existing protocols offer little guidance for secure protocol implementation. The most common approach is to use TLS, but it may not suit all applications, for instance those implemented by devices with limited resources. Developers can either build a custom protocol or use one from the academic literature, however both options present multiple challenges. To simplify this process, frameworks like NaCl [NaCl], Noise [Noise], BLINKER [BLINKER] and Strobe [Strobe] have been developed.

2.2.1 NaCl

NaCl [NaCl website, NaCl paper], pronounced "salt", is a cryptographic software library which stands for "Networking and Cryptography Library", first released in 2009. It aims to provide a simple, easy-to-use set of cryptographic primitives for developers, such as encryption and signatures needed to build higher-level cryptographic tools, and to improve security, usability, and speed.

2.2.1.1 Cryptographic algorithms

NaCl restricts the cryptographic algorithms, in light of the cryptanalytic literature, in the attempt to improve the confidence of their implementations. Specifically, NaCl uses:

1. EdDSA [EdDSA] with Elliptic curve Curve25519 [Curve25519].
2. Salsa20 [Salsa20] for encryption and Poly1305 [Poly1305] for hashing. However, it does include an AES implementation on the side.

Among the main advantages pushed by its authors are the following:

Curve25519 [Curve25519] is an elliptic curve offering 256 bits of security. It follows the standard IEEE P1363 security criteria. Secure implementations of Curve25519 offer good performances due

to the use of Montgomery representation, which allows fast single-scalar multiplication using a Montgomery ladder [Montgomery ladder].

Salsa20 [Salsa20] is a stream cipher that was selected in eSTREAM, the ECRYPT Stream Cipher Project [eSTREAM] in 2005. Salsa20 is based on a pseudorandom function that relies on add-rotate-XOR (ARX) operations, consisting of a 32-bit addition, bitwise addition (XOR), and rotation operations. The core function involves mapping a 256-bit key, a 64-bit nonce, and a 64-bit counter to generate a 512-bit block of the key stream. The recommended number of rounds to achieve a comfortable margin for security is 12.

An interesting feature of Salsa20 is the capability to allow users to efficiently navigate to any position in the key stream in constant time, avoiding use of lookup tables. However, the variant ChaCha20 [ChaCha20], developed in 2008, is now preferred due to the increased diffusion and performances.

Poly1305 [Poly1305] is a hash family that can be used as message-authentication code (MAC). Poly1305 was originally coupled with AES in order to make Poly1305-AES [Poly1305-AES]. NaCl uses Poly1305 together with Salsa20. Overall, Poly1305 is a widely recognized and secure choice for message authentication in various cryptographic applications.

EdDSA [EdDSA] is a digital signature scheme using a variant of Schnorr signature [Schnorr] based on twisted Edwards curves [Twisted Edward Curves]. EdDSA is much newer than other primitives employed in NaCl. Generally, EdDSA is considered to be more secure than ECDSA, as the use of Edward curves offers protection against timing attacks and some side-channel attacks.

2.2.1.2 The crypto_box API

NaCl presents a simple high-level crypto_box function that implements the authenticated encryption. The function puts a packet into a box that is protected in its integrity and confidentiality.

- **crypto_box(m,n,pk,sk)** takes as input the sender's secret key sk of 32 bytes, the recipient's public key pk of 32 bytes, a packet m, and a nonce n of 24 bytes. It outputs an authenticated ciphertext c which is 16 bytes longer than the packet m.
- **m = crypto_box_open(c,n,pk,sk)**. The receiver uses this function to open the packet closed with crypto_box.
- **pk = crypto_box_keypair(&sk)**. It generates a secret key and a public key
- **pk = crypto_sign_keypair(&sk)**. It generates a key pair. The public key is 32 bytes long and the private key is 64 bytes long.
- **sm = crypto_sign(m,sk)**. It creates a signed message (64 bytes longer than the original message)
- **m = crypto_sign_open(sm,pk)**. It recovers the original message.

As for failures, they are indicated by exceptions in C++ NaCl and a -1 return value in C.

NaCl's crypto_sign implementation utilises lookup tables without relying on secret indices. In this approach, each table lookup loads all entries and employs arithmetic operations to obtain the required value, eliminating the need for secret indices. The signature verification process in NaCl employs signed-sliding-window scalar multiplication, where the processing time varies based on the scalars. Importantly, this does not pose security concerns and remains compliant with NaCl's limit on the use of branches depending on sensitive values, since the scalars involved are not kept secret.

2.2.1.3 Code security measures

As indicated by the authors, NaCl follows various code security measures. The aim is to provide an API that helps implementers to improve the security posture of their integration with the least effort possible.

1. **Avoiding unnecessary data flow**. First, NaCl consistently prevents any data loads from addresses tied to confidential information, providing inherent protection against cache-timing attacks in all implementations. This imposes limitations on NaCl's implementation strategies and significantly influences the selection of cryptographic algorithms within the NaCl framework. Analogously, NaCl avoids data flow from secrets to branch conditions by systematically avoiding all branch conditions that depend on secret data.
2. **Avoiding padding oracles.** NaCl uses authenticated-encryption mechanism functions. Decryption only occurs if the data successfully passes authentication, and the authenticator's primary purpose is to prevent any attempts by attackers to forge messages that might otherwise pass authentication. Forged messages follow the authenticator verification path, with runtime depending solely on the publicly known message length. If the message is found to be forged, the system rejects it, providing no output other than confirming the illegitimacy of the message. In the case the attacker manages to forge a message, the forgery will be visible only through the receiver accepting the message. Standard nonce-handling mechanisms in higher-level protocols will instantly reject any further messages under the same nonce.
3. **Managing randomness**. NaCl avoids centralising randomness by simply reading bytes from the operating system kernel's cryptographic random-number generator. Also, when possible, NaCl chooses deterministic cryptographic operations, to reduce the load on the random number generator, improve repeatability, and possibly speed. The keypair operations use new randomness, but all of the other operations listed above produce outputs determined entirely by their inputs. Of course, this imposes a constraint upon the underlying cryptographic primitives: primitives that use randomness, such as ECDSA, are rejected in favour of primitives that use pseudorandomness.

While the NaCl API may appear very simple, NaCl has been seamlessly integrated into real-world, high-security applications currently operational on the Internet. One such example is DNSCurve [DNSCurve], offering authenticated encryption for Domain Name System (DNS) queries between a DNS resolver and server.

As the first framework of its kind, NaCl provided a foundation for constructing secure channel protocols. However, in subsequent years, it has been succeeded by other frameworks, namely Noise [Noise], Blinker [Blinker] and Strobe [Strobe]. These newer schemes offer advantages in terms of extensibility and flexibility. They have built upon the foundation laid by NaCl, enhancing the capabilities and adaptability of secure channel protocols for modern networking needs.

2.2.1.4 Implementations

The [NaCl website] provides instructions on how to download and build the reference implementation detailed in the reference paper [NaCl paper].

2.2.2 Noise

Noise [Noise] is a protocol framework based on Diffie-Hellman key agreement that can be used to construct secure channel protocols. As NaCl, Noise is not a protocol itself, it takes a basic set of cryptographic operations and allows them to be combined in ways that provide exactly the properties needed, as well as analyse whether those properties are present.

Noise is designed to be versatile and can be used to construct various secure channel protocols, allowing for the creation of custom protocols tailored to specific needs. It provides a set of building blocks, called patterns, that can be combined to create secure communication protocols.

One of the main innovations put forward by their authors is that Noise authenticates the protocol transcript by continuously hashing messages being sent and received, as well as continuously deriving new keys based on the output of key exchanges and previously derived keys. This interesting property stops at the end of the handshake.

The benefits of Noise are a short code size, very few dependencies, simplicity of the security guarantees and analysis. The Noise framework starts with the handshake phase, which includes negotiation (not defined in the framework) and the Authenticated Key Exchange (AKE). The second phase is the transport, where transport messages are encrypted with the agreed key and sent.

The Noise framework allows protocol designers to choose from a small set of Diffie-Hellman key exchange functions, symmetric ciphers, and hash functions. The framework supports the following cryptographic primitives:

- The Diffie-Hellman key exchange can be employed with Curve25519 or Curve448. The 25519 DH functions are advisable for standard applications, while the 448 DH functions could provide additional security in the event of an attack targeting elliptic curve cryptography.
- Encryption supports ChaChaPoly cipher functions (AEAD_CHACHA20_POLY1305) [RFC 7539] and AES256 with GCM [NIST SP 800-38B]
- HMAC can be computed using the hash function families of SHA2 [FIPS 180-4] and BLAKE2 [BLAKE2]: SHA2 is widely available and is often used alongside AES, while BLAKE2 is fast and similar to ChaCha20. In particular, SHA256, SHA512, BLAKE2s, and BLAKE2b are supported.

2.2.2.1 Handshake

The handshake phase follows a particular pattern defined by the Noise framework. During the handshake the parties exchange Diffie-Hellman public keys and perform a sequence of Diffie-Hellman operations, hashing the results into a shared secret key. The Noise handshake establishes an AEAD-encrypted channel that provides various forms of confidentiality, integrity, and authenticity guarantees, depending on the chosen handshake pattern.

The Noise framework supports handshakes where each party has a long-term static key pair and/or an ephemeral key pair to provide forward secrecy, so that a later compromise of long-term static keys would not reveal the plaintext contents of previous communications. A Noise handshake is described by a simple language consisting of tokens which are arranged into message patterns, that in turns are arranged into handshake patterns. Noise provides a pre-shared symmetric key or PSK mode to support protocols where both parties have a 32-byte pre-shared secret key.

More in detail, Noise handshake is described by the following language:

- **Tokens**: There are a total of 6 tokens: “e”, “s”, “ee”, “es”, “se”, “ss”, “psk” which are arranged into message patterns.
- **Message patterns:** sequence of tokens. They specify the actual content of a handshake message.
- **Pre-message pattern**: is one of the following sequences of tokens: “e”, “s”, “e, s”, empty. A pre-message pattern represents an exchange of public keys that was performed prior to the handshake.
- **Handshake pattern**: sequential exchange of messages that comprise a handshake, consists of: two pre-message patterns, one with information about the initiator’s public keys for the responder, and the other with information about the responder’s public keys for the initiator; a sequence of message patterns containing the actual handshake messages.

During the handshake, the two parties instantiate a set of variables representing the different keys used in the protocol. A set of rules define how to send and receive handshake messages by sequentially processing the tokens from a message pattern. The variables are updated as the token gets processed.

- **s, e**: the local party's static and ephemeral key pairs.
- **rs, re**: the remote party's static and ephemeral public keys.
- **h**: a handshake hash value that hashes all the handshake data that's been sent and received.
- **ck**: a chaining key that hashes all previous outputs that will be used to derive the encryption keys for the transport phase.
- **k, n**: an encryption key k and a nonce n. Whenever a new output causes a new chaining key to be calculated, a new encryption key is also calculated. The key and the nonce are used to encrypt static public keys and handshake payloads, which provides some confidentiality and key confirmation during the handshake phase.

The possible tokens are:

- "e": This token is used by the sender to store a newly generated ephemeral key.
- "s": The sender writes its static public key from the s variable into the message buffer.
- "ee", "se", "es", "ss": A Diffie-Hellman key exchange is performed between the key pairs of the initiator and the sender. The key pairs could be either ephemeral or static.
- "psk": This token does not cause any transmission of data between the parties, as a PSK is pre-shared by definition.

All Noise messages and handshake messages have a maximum length of 65535 bytes. A transport message is an AEAD ciphertext that consists of an encrypted payload plus 16 bytes of authentication data, and a handshake message consists of a sequence of one or more Diffie-Hellman public keys and a single payload which can be used to convey certificates or other handshake data. Static public keys and payloads will be in cleartext if they are sent in a handshake prior to a Diffie-Hellman operation, and encrypted in the case they occur after.

In a handshake pattern, a letter indicates whether the parties already know each other's static public keys before the handshake or not.

- N, if the long-term public key is not defined.
- X, if it is transmitted during the handshake.
- K, if it is known by the receiver in advance.

For unidirectional patterns, a unique letter is appended and refers to the initiator with respect to the initiator. This is because the receiver's long-term public key needs to be known by the initiator in advance since otherwise no payload can be encrypted to the receiver. For interactive patterns, two letters are used. The first letter refers to the initiator's long-term public key, and the second letter indicates the same for the responder towards the initiator.

We now show an example of pattern, the authenticated Diffie-Hellman handshake (XX):

→ e

← e, ee, s, es

→ s, se

The first message consists of a cleartext public key ("e") followed by a cleartext payload. In fact, a payload is implicit at the end of each message pattern. The second message represents a cleartext public key ("e"), an encrypted public key ("s"), and an encrypted payload. This second message represents the responder authenticating itself. The last message consists of an encrypted public key ("s") followed by an encrypted payload, representing the initiator authenticating itself.

2.2.2.2 Internals

Noise depends on the following Diffie-Hellman functions:

- **GENERATE_KEYPAIR()**: Generates a new Diffie-Hellman key pair.
- **DH(key_pair, public_key)**: Performs a Diffie-Hellman calculation between the private key in key_pair and the public_key and returns an output sequence of bytes of fixed length.

Noise depends on the following cipher functions:

- **ENCRYPT(k, n, ad, plaintext)**: Encrypts the plaintext with an “AEAD” encryption mode, using the cipher key k of 32 bytes and an 8-byte unsigned integer nonce n which must be unique for the key k.
- **DECRYPT(k, n, ad, ciphertext)**: Decrypts the ciphertext.
- **REKEY(k)**: Returns a new 32-byte cipher key, computed as a pseudorandom function of k. By default is ENCRYPT(k, maxnonce, zerolen, zeros)).

In the following we report the hash-related functions and constants.

- **HASH(data)**: Hashes some arbitrary-length data and returns an output of HASHLEN bytes.
- **BLOCKLEN**: A constant specifying the size in bytes that the hash function uses internally to divide its input for iterative processing. This is needed to use the hash function with HMAC.
- **HMAC-HASH(key, data)**: Applies HMAC from using the HASH() function. This function is only called as part of HKDF(), below.
- **HKDF(chaining_key, input_key_material, num_outputs)**: Takes a chaining_key byte sequence of length HASHLEN, and an input_key_material byte sequence with length either zero bytes, 32 bytes, or DHLEN bytes. Returns a pair or triple of byte sequences each of length HASHLEN.

Noise is used today in several high-profile projects. For instance, WhatsApp uses the "Noise Pipes" construction from the specification to perform encryption of client-server communications. WireGuard [WireGuard], a modern VPN, uses the Noise IK pattern to establish encrypted channels between clients. The Slack's Nebula project [Nebula], an overlay networking tool, uses Noise, together with the Lightning Network [Lighting network] and I2P [I2P].

2.2.2.3 Implementations

The noise webpage [Noise] provides reference implementations in different coding languages, from C and C#, to Go, Haskell, Java, Javascript, Python and Rust.

2.2.3 BLINKER

BLINKER [Blinker] is a light-weight cryptographic suite based on the Sponge construction used by the SHA-3 algorithm [SHA-3] KECCAK [KECCAK]. The basic idea is to use the Duplexing of the sponge [Sponge Duplex], where the ciphertext is produced by absorbing the plaintexts on the rate. Then at each time a MAC of the session can be generated by padding with 0's and squeezing.

BLINKER utilises a sequential state authentication mode, prioritising security and efficiency while allowing for straightforward security proofs. To address synchronisation challenges and minimise the implementation footprint, BLINKER employs a half-duplex mode, enabling full state sharing between the involved parties. In a half-duplex mode, communication alternates between the two parties on a single channel. The continuously updated shared state authenticates the current message and also validates all prior messages and secrets exchanged during the session by both parties, preserving their relative order.

The protocol defines an encoding transform that takes as inputs a state variable, a plaintext, and a padding, and outputs a new state and ciphertext message. The decoding function produces the same new state and plaintext from the ciphertext and equivalent state variable and padding,

synchronising the state between sender and receiver, or resulting in a failure in case of an authentication error.

Through judicious use of domain-separating padding, security proofs enable the utilisation of sponge states for an unlimited sequence of authenticated messages without the necessity for sequence numbers and re-keying: The new state can be then used for transmitting another message. This is one of the main observations which led to BLINKER and it is called **MAC-and-Continue mode**. This mode greatly reduces the latency of implementation as "initialization rounds" are not required for each message.

The new padding rule is called **Multiplex**. All different kinds of data, such as input and output blocks, encrypted and authenticated data, keys, and nonces must be encoded unambiguously as Sponge inputs. Sponge constructions generally consist of a state and a keyless cryptographic permutation. At each iteration, one word of size equal to the capacity is retained for domain separation. The iteration is then defined in terms of an arbitrary absorption, squeezing, encryption and decryption. In the squeezing phases the output blocks are virtually 0-padded. If less than rate bits of the block are being squeezed out, a single "1" bit is XORed to the state after the location of the last output bit.

A set of domain separators is indicated to distinguish the different phases of the protocol. In Figure 5 are shown the proposed bits used in the Multiplex Padding Word which is XORed with the state. Depending on protocol state and the intended usage of the message block, multiple bits are set simultaneously.

Bit	Mask	When set
0	0x0001	This is a full input or output block (r bits).
1	0x0002	This is the final block of this data element.
4	0x0004	Block is an input to sponge ("absorption").
3	0x0008	Block is output from sponge ("squeezing").
4	0x0010	Associated Authenticated Data input.
5	0x0020	Secret key block.
6	0x0040	Nonce input block.
7	0x0080	Encryption / Decryption block.
8	0x0100	Hash block.
9	0x0200	Keyed Message Authentication Code (MAC) output block.
10	0x0400	Block for state storage or reloading.
11	0x0800	Pseudo Random Number Generator (PRNG) block.
12	0x1000	Originating from Alice (client / slave).
13	0x2000	Originating from Bob (server / master).
14	0x4000	Tree chaining Node.
15	0x8000	Tree final Node.

Figure 5: BLINKER domain separators

BLINKER has inspired the development of Strobe, another cryptographic protocol framework based on a sponge construction. We delve into the details of Strobe in the next section.

2.2.3.1 Implementations

Unfortunately, we were not able to find any open source implementation of the Blinker framework.

2.2.4 Strobe

Strobe [Strobe paper, Strobe website] is a lightweight framework, designed for embedded systems, for building both cryptographic primitives and network protocols. Strobe is a sponge construction in the same family as the BLINKER framework, used for building cryptographic two-party protocols. It can also be used for symmetric cryptosystems such as hashing, AEAD, MACs, PRFs and PRNGs.

Strobe aims to enhance the development, deployment, and analysis of cryptographic protocols, tailored to seamlessly integrate with memory-constrained IoT devices. It achieves this by employing a singular block function, Keccak-f, for both message encryption and authentication. Strobe supports a diverse array of protocol building blocks, such as authenticated Diffie-Hellmann, signatures, and password-authenticated key exchange. Performance is a secondary goal for Strobe as it is based on SHA-3, which does not have acceleration on most CPUs.

Strobe benefits from a similar property as Noise, effectively absorbing every operation to influence the next ones. The Strobe specification is comparable in aspect to Noise, but focuses only on the symmetric parts of a protocol.

Strobe offers a notable advantage over BLINKER by linking the cryptography of an operation primarily to its data flow, resulting in divided metadata. At a low level, padding in Strobe serves as an indication of data flow, adhering to a strict format closely tied to the operation's execution. The second layer of metadata defines the operation's significance to the protocol, presenting a flexible, free-form structure with arbitrary length. This metadata, sharing mechanisms with other messages, can be implicit or serve as framing information for the transport, sent either encrypted or in the clear. Strobe's approach enhances application flexibility, generally outperforming BLINKER with comparable complexity. Additionally, Strobe refines BLINKER's ordering of metadata and data: BLINKER’s pad word is input along with the data, so it only affects later operations and later blocks of the same operation. Strobe’s padding is entered before the data, so that a MAC is always different from an encryption.

Strobe library offers a link between application and network, shown in Figure 6. The application domain holds data such as keys, plaintext messages, nonces and associated data, which must be protected in their confidentiality and integrity.

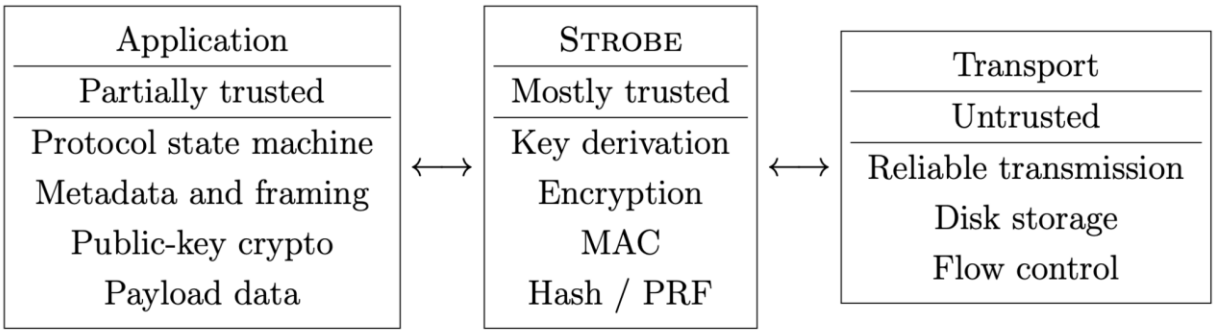


Figure 6: Strobe division of responsibilities [Strobe paper]

2.2.4.1 Implementation

The parameters are the following.

- Let b be either 400, 800 or 1600.
- Let F be the function KECCAK-f.
- Let N = b/8. Strobe treats F as a function which takes as input an array of N bytes and returns another array of N bytes.
- Let sec be a target security level, either 128 or 256 bits.

- Let R = N - (2*sec)/8 - 2. This will be the number of bytes in a Strobe block.

A Strobe object has the following state variables:

- A duplex state st: an array of N bytes.
- A position pos: the position in the duplex state where the next byte will be processed.
- A position posbegin: the position in the duplex state which is 1 after the beginning of the current operation, or 0 if no operation began in this block.
- A variable I0: This variable describes the role of this party in the protocol to keep protocol transcripts consistent.
    - I0 = None. Initially, the role is undecided
    - I0 = 0. When a party sends a message, becoming initiator
    - I0 = 1. When a party receives a message, becoming responder

A Strobe protocol is composed of a sequence of low-level operations. For example, an AEAD system might consist of a key, an associated datum, an encrypted message and a message authentication code.

Each operation has a corresponding "meta" variant, and the meta operation functions in the exact same manner as the standard operation. The only point of distinction between the two lies in the presence of an "M" bit, which is hashed into the protocol transcript specifically for the meta operations.

The Horton principle states that it is important to authenticate the meaning rather than the message itself. Strobe implements the principle with Composite operations. This is achieved by incorporating metadata into the operation to articulate its significance. The way to encode this meaning is not defined, allowing the metadata to manifest in numerous meta operations of varying lengths. The processing sequence involves handling the metadata before the data, meaning that PRF outputs, encrypted data and MACs will depend on the metadata as well as on previous operations.

The operations are:

- **AD**: xor bytes of the associated data into the state. This data must be known to both parties, and will not be transmitted.
- **KEY**: sets a symmetric key and, if there is already a key, the new key will be cryptographically combined with it. The KEY operation shares a data flow similar to AD, as both are intended to function as inputs to a random oracle. However, the KEY operation involves overwriting a section of the state with the new key, so that when the key has sufficient length, it serves as a preventive measure against rollback.
- **CLR**: send or receive plaintext data, and xor it into the state.
- **ENC**: send or receive encrypted data. To send, run F to begin a new block, xor the plaintext with the state, which produces both a new state and a ciphertext, and send the ciphertext to the other party. To receive, run F to begin a new block, receive a ciphertext, and xor it with the state to obtain a plaintext.
- **MAC**: send or receive message authentication code. To send: run F to begin a new block, send bytes of the state to the other party. To receive, run F to begin a new block, receive bytes and check whether they coincide to the bytes of your state. If not, then the session has been corrupted.
- **PRF**: extract hash or pseudorandom data. Run F to begin a new block, read bytes of the state out to the application. This data can be treated as a hash of all preceding operations, messages and keys.
- **RATCHET**: This operation does not have input or output, as its primary purpose is to mitigate rollback attacks. In a scenario where an attacker manages to retrieve the sponge's state at the protocol's conclusion, potentially through an application exploit, the attacker could reverse the Strobe steps to discern earlier states and decrypt preceding messages. This process is analogous to recovering the key in a symmetric cipher. The RATCHET operation

addresses this vulnerability by selectively erasing a portion of the state, rendering the protocol non-invertible. Consequently, an attacker would not be able to decrypt messages from earlier stages. The function runs F to begin a new block and then overwrites L bytes of the state with zeros to prevent rollback. As nullifying the whole state would destroy the key, only up to R bytes are nullified at a time, calling F in between.

Operations are categorised by four important pieces of information, called flags:

- **I**: If the operation first performs transport, then cipher, then application, as opposed to the other direction.
- **A**: If the operation sends or receives data belonging to the application.
- **C**: If the operation sets a key or uses the cipher's output. Operations with the C flag either output the cipher's data, or if they have no output, rekey the cipher.
- **T**: If the operation sends or receives data via the transport.

By using this flag system, the behaviour of each operation follows in a straightforward manner. Strobe operations and their data flow are illustrated in Figure 7.

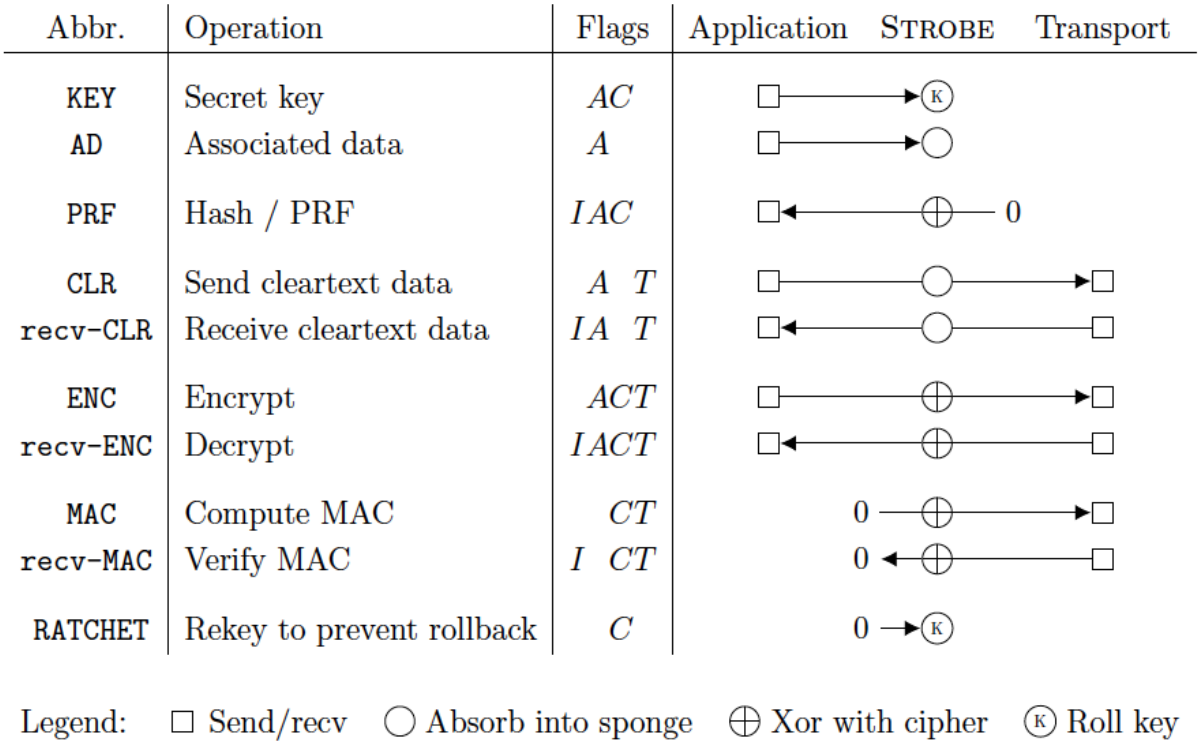


Figure 7: Strobe operations [Strobe paper]

2.2.4.2 Implementations

The Strobe project page [Strobe website] provides information on an open source implementation of its framework. Alternatively, we found another open source implementation at https://github.com/mimoo/strobe-mirror.

2.2.5 Remarks

The frameworks discussed in this section - NaCl, Noise, Blinker, and Strobe - function as sets of simple APIs for constructing cryptographic protocols, providing a range of primitives for implementing secure communication.

NaCl, the pioneering framework, laid the initial foundation for secure channel protocols frameworks, but over time, Noise, Blinker, and Strobe have emerged as successors.

Noise is characterised by keeping a running hash of the messages: in this way, the output from any step depends not only on the keys, but also on all the preceding inputs, such as nonces and associated data. Noise has achieved significant real-world adoption with its flexible and extensible design, due to the use of predefined patterns in the handshake phase. Blinker, designed with simplicity, relies on a single cryptographic permutation, Keccak. It operates in a half-duplex mode, meaning that it alternates communication between parties on a single channel. Strobe evolved from Blinker with a focus on versatility, and also shares certain characteristics with Noise, continuously hashing messages and deriving keys. However, it exclusively focuses on the symmetric part.

In designing our new protocol, we will take inspiration from these established frameworks, incorporating their best properties, but adapting these properties and algorithms to target resource constrained environments, like that of IoTs.

2.3 JSON Web Tokens

The JSON Web Token (JWT) standard described in RFC 7519 [JWT] specifies how to securely represent claims between two parties. It is one of the building blocks of the Oauth protocol [RFC 9396], which allows authorization grants to be exchanged between services. For example, applications on computers, phones, TVs, printers etc. can use Google OAuth 2.0 to authorize access to Google's APIs [Goo23]. This means that once you are logged in your phone, you can access Google Drive documents without the need to input your credentials again.

JWT's have been developed in order to reduce both development complexity and friction in user adoption. Developers can use a single API to create and validate JWTs for different services. At the same time the user experience is simplified by requiring a Single-Sign-On (SSO) to use different services.

While the user experience has effectively been improved, the same is not completely true for developers. The main problem lies in the versatility of JWTs which can be declined in several different forms, use many different cryptographic algorithms and solve many different problems.

For example, security has been introduced in JWTs in the form of signature and encryption, providing respectively authenticity and privacy. JWTs are described in RFC 7519 [JBS15b], however, their security aspects are discussed in separate RFCs, namely RFC 7515 [JBS15a] and RFC 7516 [JH15]. The former describes the JSON Web Signature (JWS), a signed JWT, to ensure authenticity. The latter, instead, describes a JSON Web Encryption (or JWE), which is an encrypted JWT and ensures privacy. These three RFCs specify how JWT, JWS and JWE can be combined together, as not all combinations are allowed.

However, the cryptographic algorithms to be used are specified in RFC 7518 [Jon15a], which describes the JSON Web Algorithms (or JWA). Finally, a JSON Web Key (JWK), described in RFC 7517 [Jon15b] is the data structure used to represent a cryptographic key for one of the JWT/JWE/JWS. Apparently, this fragmentation of references does not ease the work of developers, which may lead to security issues. To cope with such a situation, a further RFC was provided: RFC 7520, which describes "Examples of Protecting Content Using JSON Object Signing and Encryption (JOSE)".

On top of all these references, IETF added its contribution, by drafting RFC 8725 [SHJ20], with best practices for implementations (specific to the cryptographic usage) in order to guide developers implementing the specifications. Unfortunately, the issues described in this work are not covered by the best practices listed in that RFC.

Recently, Shingala [Shi19] published an interesting comparison between the use of JWTs versus the use of the mutual authentication from the TLS [Res18] protocol for authentications of things on the Internet. Interested by such work and from the internet discussions on-going, we decided to contribute with our findings concerning the use of JWTs for IoT.

In the following sections we present in more detail the different aspects of the JWT, JWS and JWE standards, described respectively in RFC 7519, RFC 7515, and RFC 7516 [JBS15b, JBS15a, JH15].

2.3.1 JWT

A JSON Web Tokens (JWT), described in RFC 7519 [JBS15b], is a compact, URL-safe means of representing claims to be transferred between two parties. The payload of a JWT consists of a series of key-value pairs known as claims. Some claims have specific value that triggers validation logic, for instance the *exp* claim which defines when the token should expire and no longer be considered valid.

The claims in a JWT are encoded as a JSON object that can be used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a MAC and/or encrypted.

2.3.2 JWS

A JSON Web Signature (JWS), described in RFC 7515 [JBS15a], allows the creation of a digital signature or Message Authentication Code (MAC) over a JSON-based data structure to ensure data integrity and authenticity.

Cryptographic algorithms and identifiers for use with this specification are described in the separate JSON Web Algorithms (JWA) specification and an IANA registry defined by that specification.

The JWS does NOT protect the confidentiality of the data. Anyone who can get ahold of the JWS can decode and read the bytes that were signed. Related encryption capabilities are described in the separate JSON Web Encryption (JWE) specification.

2.3.3 JWE

A JSON Web Encryption (JWE), described in RFC 7516 [JH15], is a JSON-based data structure which protects the confidentiality of the payload by encrypting it and can sometimes ensure integrity of the payload data (depending on the algorithm chosen). A JWE can be used to store/transport sensitive data.

2.3.4 Supported cryptographic algorithms

One issue within the JWTs is how public keys used to sign them are distributed among the services. For example, a common tool used to generate key pairs and certificates is Keytool [Keytool].The tool first generates a key pair, which also creates a Java Keystore. The public key is then wrapped into an X.509 self-signed certificate, which can be distributed directly by the owner or by a Certificate Authority (CA).

The cryptographic algorithms supported by JWT and the identifiers to be used are collected in the JSON Web Algorithms (JWA) [JWA] specification. The JWT module supports the following JSON Web Algorithms:

- Signing algorithms:
    1. HS256, HS384, HS512
    2. RS256, RS384, RS512
    3. ES256, ES384, ES512, ES256K
    4. PS256, PS384, PS512
    5. EdDSA
- Encryption key agreement algorithms:
    1. RSA1_5, RSA-OAEP, RSA-OAEP-256, ECDH-ES

1. A128KW, A192KW, A256KW
2. A128GCMKW, A192GCMKW, A256GCMKW

- Content encryption algorithms:
  1. A128CBC-HS256, A192CBC-HS384, A256CBC-HS512
  2. A128GCM, A192GCM, A256GCM

The first combination of algorithms that can be used is RS256, RSA-OAEP, and A256GCM:

1. Sign with RSASSA-PKCS1_v1_5 [RFC 3447] with SHA256. The digital signature is validated as follows: submit the JWS Signing Input, the JWS Signature, and the public key corresponding to the private key used by the signer to the RSASSA-PKCS1-v1_5-VERIFY algorithm using SHA-256 as the hash function.
2. Key encryption is performed with RSA with OAEP.
3. Perform content encryption with AES256 GCM [RFC 5288].

In RFC 3447 [RFC 3447] it is explicitly recommended to not adopt RSASSA-PKCS1-v1_5 for new applications, and instead requests the transition to RSASSA-PSS. However, the specification does consider RSASSA-PKCS1-v1_5 for interoperability reasons as it is commonly implemented.

The second option is to use ES256, ECDH-ES, and A128GCM. The signature is computed with ECDSA using P-256 and SHA-256 [JWA]. Then, ECDH-ES (Ephemeral-Static) is used for the key agreement using Concat KDF [NIST SP 800-56A], and a temporary AES128 key is computed with AES128GCM [RFC 5288] to encrypt the content.

Regarding the final size of the JSON Web Token, it can be noticed that due to the necessary base64 encoding the size of the final composition of JWE and JWS is increased by about 60% (30 + 30). However, it is possible to use compression algorithms to compress back the JTW to a reasonable size.

2.3.5 Sign then encrypt

JSON Web Tokens can be signed then encrypted to provide confidentiality of the claims. While it's technically possible to perform the operations in any order to create a nested JWT, senders should first sign the JWT, then encrypt the resulting message. Sign-then-encrypt is the preferred order for three reasons:

1. It prevents attacks in which the signature is stripped, leaving just an encrypted message.
2. It provides privacy for the signer.
3. Signatures over encrypted text are not considered valid in some jurisdictions.

Certain papers advocate applying a second signature after the encryption [Dav01]. This isn't required with standard JWE algorithms due to their use of authenticated encryption [Bla05].

2.3.6 Remarks

JWTs are used in many places on the internet, for example OpenID Connect (OIDC) ID Tokens [ID Token] and Access Tokens, or in the authentication protocol to connect devices to the Google IoT core. During our review of the protocol we identified a security issue in the use of JWT as authentication mechanism, for example in the Google IoT code. Such an issue is described in detail in chapter [JWT Back to the future].

2.3.7 Implementations

Several implementations for JWTs exist. The project page provides an extensive list of libraries for JWTs in different languages: https://jwt.io/libraries.

2.4 Xoodyak

In recent years, newer AEAD algorithms have been proposed that might promise better performance and productivity trade-offs with respect to the current SCP standard. One of them is based on the sponge construction and is called Xoodyak [Xoodyak].

Xoodyak operates in a duplex sponge-based mode, which allows for flexible and efficient processing of various cryptographic operations. It does not use traditional block cipher modes like CBC or GCM but instead relies on a single evolving state that is continuously updated through the Xoodoo permutation. Xoodyak's main modes of operation include authenticated encryption with associated data (**AEAD**), **hashing**, and **key derivation**. In **AEAD** mode, the algorithm initializes with a secret key and nonce, processes associated data for authentication, encrypts the plaintext while simultaneously generating an authentication tag, and ensures decryption integrity by verifying the tag. In **hashing** mode, Xoodyak absorbs input data in blocks, applies the Xoodoo permutation to mix the state, and then squeezes out a hash digest. For key derivation (**KDF**), it absorbs an initial key and optional context data, processes them through Xoodoo, and extracts cryptographic keys. This flexible duplex mode makes Xoodyak highly efficient for lightweight cryptography, as it eliminates the need for separate encryption and authentication steps while maintaining strong security guarantees

Authenticated Encryption with Associated Data (AEAD)

Xoodyak provides AEAD, which ensures confidentiality, integrity, and authenticity of encrypted data. It follows the MonkeyDuplex construction, where encryption and authentication are combined into a single operation.

Steps:

1. **Initialization**
   - A key and **nonce** are absorbed into the state.
   - Xoodoo permutation is applied to mix the input securely.
2. **Processing Associated Data (AD)**
   - Any additional data (e.g., headers, metadata) is absorbed.
   - This ensures the authentication covers more than just the ciphertext.
3. **Encryption & Tag Generation**
   - The plaintext is absorbed and transformed using the Xoodoo permutation.
   - Ciphertext is extracted (squeezed) while authentication data is updated.
   - A cryptographic tag is generated for integrity verification.
4. **Decryption & Authentication**
   - The ciphertext is absorbed and transformed back into plaintext.
   - The computed authentication tag is compared with the received tag.
   - If the tags match, the message is valid; otherwise, decryption fails.

Hashing

Xoodyak can function as a cryptographic hash function, producing a fixed-length digest from variable-length input.

Steps:

1. **Initialization**
   - The state is reset and prepared for hashing.
2. **Absorbing Input Data**
   - The message is absorbed into the state in blocks.
   - Xoodoo permutation is applied after each block to ensure diffusion.
3. **Squeezing the Hash Output**
   - Once all input is absorbed, the hash output is extracted.
   - Xoodoo permutation continues to mix the state for extended-length hashes.

Key Derivation Function (KDF)

Xoodyak can derive cryptographic keys from an initial secret and optional context information.

Steps:

1. **Absorbing the Key Material**
   - The input secret (e.g., master key) and context data (e.g., salt) are absorbed.
2. **Applying Xoodoo Permutation**
   - The state is mixed to ensure security.
3. **Squeezing Derived Keys**
   - New cryptographic keys are extracted from the state.

The exchange of C-APDUs and R-APDUs between a secure element and a host can be seen as a special case of Authenticated Encryption with Associated Data (AEAD). APDUs can be conceptually divided into two parts: one that is always sent in clear (e.g., the header field) and another which may be encrypted (like the message payload). While encryption is applied only to one part of the message (the payload), verification of integrity is typically applied to the entire message, that is, even to the associated data that are sent in clear.

Xoodyak is a lightweight cryptographic primitive that employs a special mode of operation called Cyclist, and an underlying permutation called Xoodoo.

2.4.1 Xoodoo

A deck function [Farfalle][Deck functions] stands for *doubly extendable keyed cryptographic function*. A deck function is a function that by taking a key and a string as input, produces a seemingly random output.

A property that characterises a deck function is that the data input is not a single string, but a sequence of binary strings, and the output depends on this sequence. Moreover, a deck function must implement efficient incrementality properties. This means that both the input and the output must be extendable with an additional string without additional cost, except for cost derived from processing the extra string. This property is similar to that of an extendable output function (XOF), with the addition that in a deck function the input is extendable as well.

A deck function can readily be used for encryption, authentication and authenticated encryption. Moreover, their incrementality properties can simplify processing streams of data, with intermediate tags, and bi-directional communication. Another interesting use case is the transmission of long messages to low-end devices, where intermediate tags can authenticate the message in an incremental way.

A construction for building deck functions is Farfalle [Farfalle], of which Kravatte [Farfalle] and Xoofff [Xoodoo] are instances.

Xoodyak utilizes the Xoodoo permutation, which operates over a 384-bit state divided into 12 rounds and can be stored in 12 registers of 32 bits each, making it ideal for low-end 32-bit devices. Among its methods, it includes a ratchet mechanism (method ratchet()) that zeros out part of the state. This

feature is beneficial in scenarios involving certain types of attack, such as side-channel attacks, which could allow an attacker to retrieve the internal state of the cipher. In such situations, the attacker cannot discover the secret key after the ratchet is applied. Therefore, the ratchet mechanism is said to provide forward secrecy, at least within the context of a single session.

State Representation

The Xoodoo state consists of 384 bits, represented as a 3D array of 32-bit words:

- 3 planes
- 4 columns per plane
- Each word is 32 bits

This gives a 3 × 4 × 32-bit structure, visualized as:

```
        Plane 0              Plane 1              Plane 2
[ A0  A1  A2  A3 ]   [ B0  B1  B2  B3 ]   [ C0  C1  C2  C3 ]
```

Permutation Steps

Each Xoodoo round consists of five steps, ensuring strong diffusion and security:

1) Θ (Theta) – Mixing Across Columns

- XORs each column across all three planes.
- Ensures every bit influences multiple parts of the state.

2) ρ-west (Rho-west) – Bitwise Rotation

- Applies a cyclic shift (rotation) to each word.
- Increases the randomness in the state.

3) ι (Iota) – Round Constant Injection

- Introduces round constants to break symmetry and add non-linearity.
- Ensures resistance against differential cryptanalysis.

4) χ (Chi) – Non-linear Layer

- Applies a bitwise Boolean function across each row.
- Strengthens security by adding non-linearity.

5) ρ-east (Rho-east) – Additional Rotation

- Applies a second cyclic shift, further scrambling the state.

2.4.2 Cyclist

Cyclist should be seen as a stateful object with a set of methods[1] that manipulate the state so that it is always a reflection of the history of all preceding method invocations. Some of these operations

[1] Think of it as a C++ object.

can apply the Xoodoo permutation to the state itself so as to accomplish various cryptographic functions according to the two operating modes, namely:

1. **Hash mode.** Once the Cyclist object is initialized with constant values, Xoodyak can be used to absorb an input string (of arbitrary length) and produce ("squeeze") hash digests of the said input strings (methods absorb(), squeeze()).
2. **Keyed mode.** In this case, the Cyclist object is initialized with a key K. In this mode, Xoodyak is capable of performing, among other things, MAC computations and encryption / decryption (methods encrypt(), decrypt().

Xoodyak does not have any parameters that can be chosen by the user. This means that the user does not decide on the number of permutation rounds or the method of padding and splitting the input into blocks before encryption.

2.4.3 Implementations

The reference implementation for Xoodyak is provided by the Keccak Team at https://github.com/KeccakTeam/Xoodoo/blob/master/Reference/C%2B%2B/Sources/Xoodyak.cpp.

2.4.4 Final remarks

In this section, we delved into concepts that serve as valuable insights for developing secure channel protocols. JSON Web Tokens and Deck functions both present opportunities for advancing research in new frameworks and protocols. These constructs have the potential to become essential building blocks, offering the sought-after properties of authenticity and confidentiality within a protocol.

Finally, we investigated Xoodyak [Xoodyak] and related lightweight cryptography. The study of the state of the art motivated the protocol proposed in this work, in chapter 4. Xoodyak was selected as one of the finalists of the NIST Lightweight Cryptography standardization process (NIST-LWC), due to its adaptability and outstanding performance, as also evidenced by an evaluation conducted on the NIST-LWC finalists using RISC-V architectures [CLMLD19].

2.5 Conclusions

In this chapter we examined the state of the art in secure communication protocols within a device. For clarity in our exposition, we highlighted the key phases - Provisioning, Handshake, and Data exchange - for each protocol under examination.

Our analysis began with an examination of the widely used Secure Channel Protocol (SCP) family, followed by the Replay Protected Memory Block (RPMB) protocol. Subsequently, the proprietary protocols employed in Optiga products, Infineon's ATECC chips, and Ublox products, were scrutinised for their characteristics, performances, and security attributes.

The exploration further extended to the protocol frameworks, which provide foundational structures and building blocks for creating "customised" secure channel protocols. These frameworks, known for prioritising usability and abstraction, have significantly influenced the design of our new protocol, as detailed in the upcoming chapter.

The chapter concludes by shedding light on the emerging constructions of JWT and Deck functions, underscoring evolving methodologies in building and deploying security. The exploration undertaken here sets the stage for the subsequent chapter, where we delve into a weakness found in the use of JWTs for the IoTs. Furthermore, this work motivated us to conceive a new secure channel protocol, which we present in chapter 4.

Chapter 3 JWT Back to the future

The JWT standard described in RFC 7519 (JBS15b) specifies how to securely represent claims between two parties. It is one of the building blocks of the OAuth protocol (Lodderstedt, Richer), which allows authorization grants to be exchanged between services. For example, applications on computers, phones, TVs, printers etc. can use Google OAuth 2.0 to authorize access to Google's APIs (Goo23). This means for example that once you are logged in your phone, you can get access to your Google Drive documents without the need to input your credentials again.

JWT's have been developed in order to reduce both development complexity and friction in user adoption. Developers can use a single API to create and validate JWTs for different services. At the same time the user experience is simplified by requiring a Single-Sign-On (SSO) to use different services. While the user experience has effectively been improved, the same is not completely true for developers. The main problem lies in the versatility of JWTs which can be declined in several different forms, use many different cryptographic algorithms and solve many different problems.

Lately, Google Cloud IoT Core adopted the use of JWTs to authenticate the requests coming from devices in the field. The idea is that the JWT token is much more lightweight to produce than a TLS authentication, thus improving the efficiency of the "things", as described in (Goo18; HiveMQ).

The present work stems from the fact that the JWT standard does not require the inclusion of a nonce in the token construction, thus the server receiving the token cannot validate its freshness. Despite a claim named "nonce" being defined and registered with IANA for JWT (see (IANA)), none of the JWT claims are mandatory, thus in practice such nonces are never used. This remark is the starting point for replay attacks and also for a more subtle attack that we describe in this work as JWT Back to the future. We show an abuse of the JWTs in the supply chain, where a malicious attacker who can manipulate the time perceived by the iot device under production can obtain a set of valid JWTs that she can use in the future to authenticate custom messages sent to the Cloud. After the JWTs are produced, the device has no recollection or logs of the happening, and so the abuse cannot be detected. Naively, without a SE, an attacker may simply obtain the signing key by accessing the micro-controller internal memory in debug mode, during production. One of our main contributions is to show that the presence of a SE does not necessarily thwart the attack, and we present the few options currently available to protect a system against our finding, in Countermeasures.

In the following, we showcase our work by using the Arduino MKR 1010 and the Google Cloud IoT Core. This example is illustrated by Google in (Goo20). We specifically opted to use a now retired platform in order to avoid the abuse of our attack in real world scenarios (as the service has been completely shut down in 2023, existing connections were shut down and no device connecting to it should still remain in the field). However, this work straightforwardly applies to other uses of the JWT, like for example HiveMQ or the EMQX platforms (HiveMQ; EMQX), and also to other token standards, for example the CWT (JWET18), described by the OSCORE IETF working group (BS16;LV18) in the context of the RESTful environments for authentication and authorization for constrained environments (ACE) project. CWTs were used for example in the EU Digital Covid certificate, or in eat tokens from (LMOW24). Finally, we remark that our work also applies to the emerging technology of Electric Vehicles (EV) as the standard used to transfer charge information to the station is based on the JWT and CWT standard and is described in (Se).

Recently, Shingala (Shi19) published an interesting comparison between the use of JWTs versus the use of the mutual authentication from the TLS (TLS 1.2) protocol for authentications of the things on the Internet. Interested by such work and from the internet discussions on-going, we decided to contribute with our findings concerning the use of JWTs for IoT.

Interestingly, we remark that our work does not apply to the ARM PSA tokens (ARM), as such tokens include a mandatory nonce coming from the caller, to demonstrate freshness of the generated token (see ARM Section 4.1.1).

3.1 JWTs, JWSs and JWEs

In this section we present in more details the different aspects of the JWT, JWS and JWE standards, which collectively go under the JOSE acronym (for JSON Object Signing and Encryption), described respectively in RFC 7519, RFC 7515, and RFC 7516 (JBS15a; JBS15b;JH15)). Then we describe other token types, as the cwt, and eat, defined by the ACE group (LV18), and the RATS working group (LMOW24), respectively.

3.2 JWT

A JWT, described in RFC 7519 (JBS15b), is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JWS structure or as the plaintext of a JWE structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted. A JWT is divided into three fields: header, payload, and signature. The header provides information about the content of the JWT, like the cryptographic algorithms used in the other blocks. The payload contains the actual claims: a set of statements about an entity. Claims can be public or private and standardized by the IANA Web Token Registry or not. Finally the latter field consists of the signature of the previous blocks. These three fields, encoded in Base64, and separated by dots, compose the JWT. An example JWT from the site (Okta) is provided in the next picture.

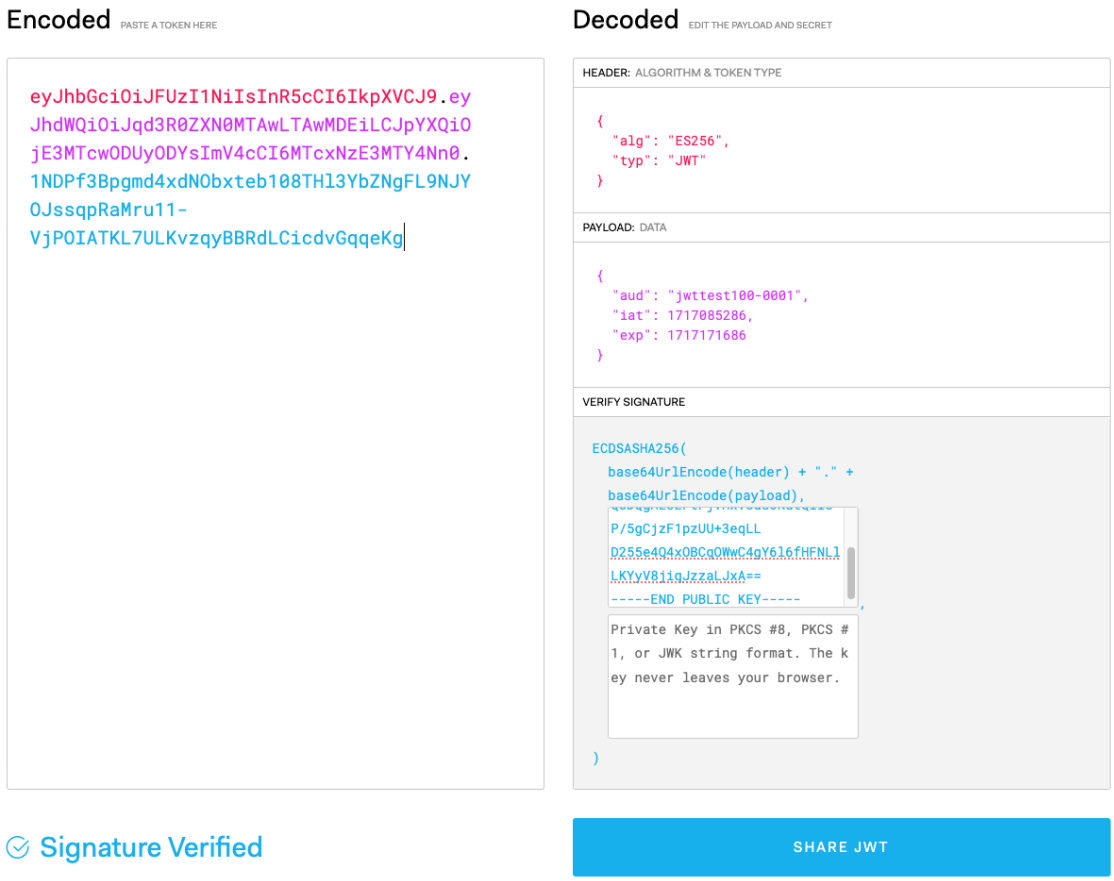


Figure 8: Example of a legitimate JWT

Obtained by using an Arduino MKR WiFi 1010 and and decoded by JWT.io. On the left is presented the base64 encoded value, while on the right are presented the decoded header and payload, and the signature verification.

3.3 JWT

A JWS, described in RFC 7515 (JBS15a), allows the creation of a digital signature or Message Authentication Code (MAC) over a JSON-based data structure to ensure data integrity and authenticity. Cryptographic algorithms and identifiers for use with this specification are described in the separate JSON Web Algorithms (JWA) specification and an IANA registry defined by that specification.

The JWS does NOT protect the confidentiality of the data. Anyone who can get a hold of the JWS can decode and read the bytes that were signed. Related encryption capabilities are described in the separate JSON Web Encryption (JWE) specification.

3.4 JWE

A JWE, described in RFC 7516 (JH15), is a JSON-based data structure which protects the confidentiality of the payload by encrypting it and can sometimes ensure integrity of the payload data (depending on the algorithm chosen). A JWE can be used to store/transport sensitive data.

3.5 Sign then encrypt

JWTs can be signed then encrypted to provide confidentiality of the claims. While it's technically possible to perform the operations in any order to create a nested JWT, senders should first sign the JWT, then encrypt the resulting message. So, sign-then-encrypt is the preferred order for the following reasons: it prevents attacks in which the signature is stripped, leaving just an encrypted message; it provides privacy for the signer; finally, signatures over encrypted text are not considered valid in some jurisdictions. Certain papers advocate applying a second signature after the encryption (Dav01). This isn't required with standard JWE algorithms due to their use of authenticated encryption (Bla05).

In the following, we will use the term JWT to denote a JWT being it in plaintext, signed or encrypted (JWE, JWS). In particular, we will focus on the authenticity property, thus on JWSs, for which we demonstrate that the attacker can bypass the authenticity of some of the fields contained in the claims in Back to the future.

3.6 Other formats: CWTs and EATs

The Authentication and Authorization for Constrained Environments (ACE) working group from IETF (LV18) is responsible for the standardization of a solution framework to enable the protection of exchanges between a client and a server in a constrained environment. The method chosen by the working group to protect the exchanges uses tokens similar to the JWTs, but based on the CBOR format (Bormann and Hoffman 2020; Ericsson, n.d.), called CWTs. A Go implementation for CWTs can be found for example in (COSE). Furthermore, the RATS IETF working group developed a draft (LMOW24) in which they describe the eat token (LMOW24) as a JWT or cwt token, with some attestation-oriented claims. Such tokens are used by a relying party, server or service to determine the type and degree of trust placed in the entity, like a smartphone, IoT device, network equipment and so on.

3.7 The IoT generic architecture

We consider a generic IoT environment, composed of a device which connects to the Cloud in order to securely send and receive data from its own sensors. The device itself is composed of a micro-controller, which implements the application, and various modules in charge of different tasks. For example the micro-controller may implement the connectivity itself, or delegate it to an WiFi, BLE, or

GSM module. Similarly, the micro-controller may use the cryptographic functionalities exposed by a SE.

In our scenario, we assume that the cryptographic primitives necessary for the security operations are provided by an SE module connected to the MCU, for example by the I2C channel (NXP21), and not operated by the MCU itself. Such a situation is usually depicted as more secure, thanks to the presence of the SE and the assurance provided by it (see for example (Goo18)). Then a WiFi module (for example connected to SPI bus (Mot84)) is in charge of establishing the TLS connection to a Cloud endpoint. This architecture is commonly realized by many IoT devices.

As described in (Goo18), the use of JWTs works as follows. The device will establish a secure connection to the global Cloud endpoint using TLS by using the WiFi module, but instead of triggering the mutual authentication it will generate a very simple JWT, sign it with its private key and pass it as a password. The JWT is received by the Cloud, the public key for the device is retrieved and used to verify the JWT signature. If valid, the mutual authentication is effectively established. As the SE offers the possibility to sign JWTs securely without ever exposing the private key, this scenario is generally considered more secure than in the absence of an SE.

We depict in Figure 9 and Figure 10 the two different architectures described above in the following pictures. The latter architecture is realized for example by the Arduino MKR WiFi 1010. We use it in the following by connecting it to the Google IoT Core. Such a combination has been suggested in (Goo18; Ard19).

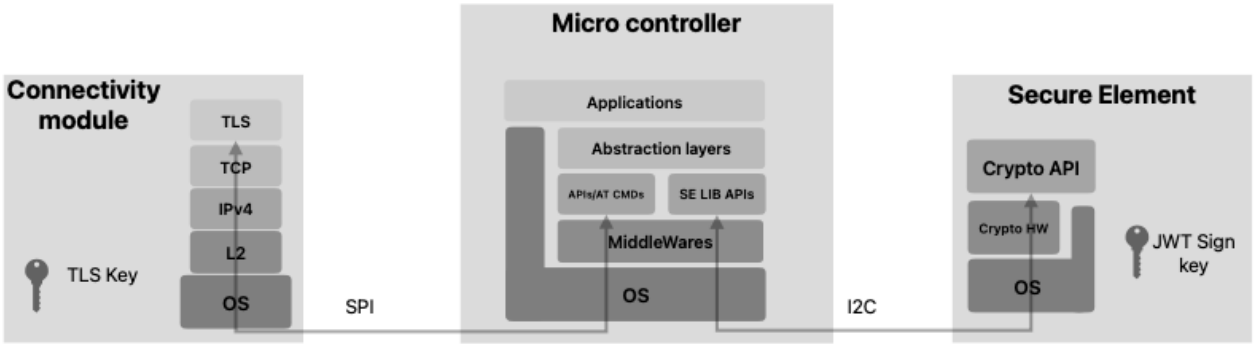


Figure 9: Architecture with different modules for the connectivity and security

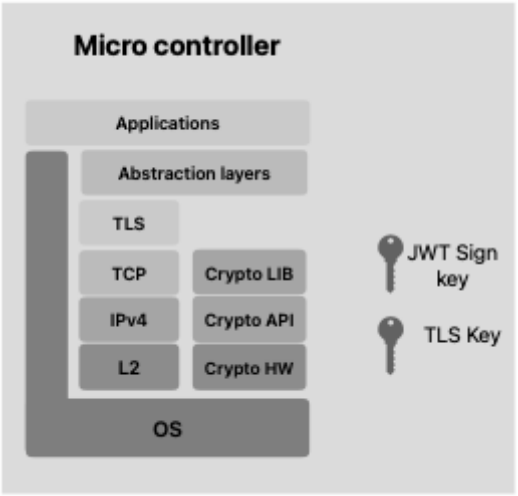


Figure 10: Example of an IoT hardware and software stack architecture

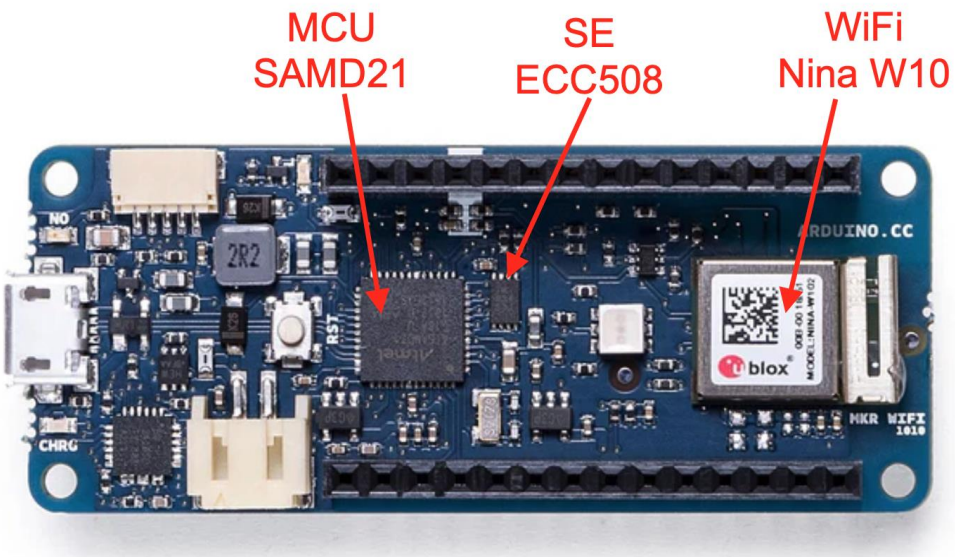


Figure 11: Organization of the Arduino MKR WiFi 1010 module

3.8 Device's lifecycle

This section presents an example life-cycle of an IoT device that we will use to illustrate the attack. The life-cycle of an IoT device is split into two main parts: manufacturing and deployment. Such splitting is useful to distinguish those steps which need to be performed before and after the device is deployed in the field.

3.9 Manufacturing

During the first stages of life, the device is assembled, in particular all sub-modules are soldered together, the SE is connected to the micro-controller which is itself connected to the Wi-Fi module for communication. Afterwards, tests are performed on the complete device to verify all connections and functionalities. This step in general involves the generation of a cryptographic private key inside the SE on-boarded with the device, and the publication of the associated public key, together with an ID of the device, onto the Cloud Provider's fleet management system. Alternatively, it is possible that the SE comes pre-provisioned by the founder. For Google Cloud IoT Core, the instructions for device on-boarding can be found on different sources (GooBP;NXPA71H). The final firmware is then loaded onto the micro-controller and thus the device is ready to go in the field. It is quite common that the manufacturing process is performed by a dedicated third-party company, where the use of secure manufacturing lines is not guaranteed. So the personnel having access to the devices during the manufacturing could be malicious and may try tampering, in particular, with the cryptographic assets. This aspect is commonly referred to as supply chain attacks, since it extends to all the steps of the supply chain, from component acquisition to manufacturing up to last mile shipping. During manufacturing, a supply chain attack can be problematic because the malicious attacker can act on the complete population of produced devices. Loading malicious firmware is the most naive supply chain attack. In such an environment, a remote attestation can be used to prove the device's software and hardware integrity to a remote party, providing cryptographic evidence that it's running as expected and hasn't been tampered with. However there are attacks, such as the one we present here, that leave no evidence on the final device, and thus are hard to detect.

3.10 Deployment

After manufacturing is finished, the device is deployed in the field. From that moment on, every time the device needs to connect to the cloud to transfer data, the micro-controller connects to the Wi-Fi module through the SPI channel, and requests the current time to a network time server. The current time thus retrieved is used to prepare a JWT. Claims in the JWT are generated by the micro-controller and signed by the SE. The signature generation is requested and returned through the I2C channel. The micro-controller then requires the opening of a TLS session with the cloud service to the Wi-Fi module; the interconnection between the micro-controller and the Wi-Fi module is ensured by the SPI channel. After the TLS session is established the micro-controller passes the signed JWT to the Wi-Fi module, to be sent to the Cloud service. Once the Cloud receives the JWT, it validates it and acts accordingly.

We want to stress that since the SPI and I2C channels are not protected, the micro-controller and the SE are not capable of understanding if the input data are coming from legitimate sources. This is true in particular for the signature request and data to be signed from the SE perspective, or time/date incoming from the SPI in the case of the micro-controller perspective.

3.11 Threat Model

This work considers a typical IoT device production scenario where a designer provides a factory with the hardware design and firmware. The factory's role is to manufacture a specified number of devices, each loaded with the designer's chosen firmware and secrets. It is assumed that the designer uses a SE to protect sensitive information and employs JWTs to ensure message authenticity between the IoT device and the Cloud after production.

For the purpose of this work, we can assume that the private key used for signing messages is the only secret, generated and accessible exclusively to the SE on the device. A key assumption is that factory employees are untrusted and could potentially tamper with devices during production to extract sensitive information like the signing key, by accessing the micro-controller internal memory in debug mode.

Specifically, attackers are assumed to be capable of eavesdropping and manipulating data on exposed communication channels (buses) before, during, and after firmware installation, to retrieve all secret material. However, the model excludes physical attacks on devices, such as side-channel or fault injection attacks.

In this context, the attacker's goal is to generate valid communication that the Cloud provider will accept, effectively impersonating legitimate IoT devices. The authors highlight that while JWTs are intended to provide authentication, they fail to protect against attackers who gain access to the private key. This also implies that JWTs cannot protect against attackers who have access to the private key at any time, but also that even if the attacker cannot access the secret, the JWT construction fails to protect the communications that use the secret key.

In this perspective we want to remark two main aspects. First, if the SE is not present, then the attacker may simply obtain the signing key by accessing the micro-controller internal memory in debug mode. Second, in order to thwart such a naive attack, an SE is usually mounted on the device. Although very surprisingly, using a SE in this context opens a new attack scenario (Back to the future) which was not identified by the manufacturer nor the Cloud platform (Goo18).

3.12 Back to the future

In this section we show that JWTs used in such a context create a security threat to the system. The JWT is created by the device based on its own time representation, while the Cloud side has no interaction in the generation of the JWT. This means that a JWT can be created at any time by the device or by anyone having access to the private key associated with the device. This represents a

weakness in the authentication protocol, it can be used for example to mount replay attacks, by sending twice the same JWT.

We found a more subtle attack than a replay, and show in this section how to generate custom JWTs that will be valid in the future.

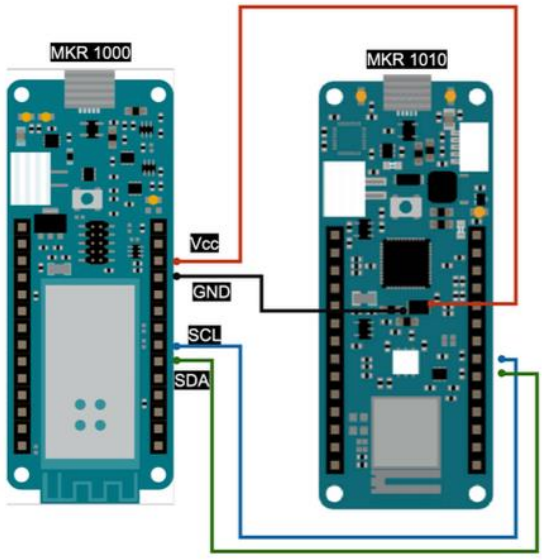
Our attack applies when the device is provided with a SE, being it pre-provisioned or not. An attacker having access to the device during manufacturing can interact with the SE and request the signature of a JWT for whatever moment in time. This is possible since the I2C between the micro-controller and the SE is not protected. Despite the possibilities for commercial SEs to establish a secure channel over the I2C with the micro-controller, we observe that this is not a sufficient countermeasure to thwart the attack presented in this work. As the key used to protect the I2C channel needs to be exchanged/rotated on the very same I2C channel on first boot, an attacker that can eavesdrop on such a communication would know the key and would thus be able to observe/modify all further communications.

Similarly, since the SPI connection between the micro-controller and the Wi-Fi module is not encrypted, an attacker can tamper with the time communicated by the Wi-Fi module as well. The attacker eavesdrops on the SPI channel and waits for the micro-controller to connect to a network time server. The attacker then alters the response, setting the time to some moment in the future. The micro-controller then receives the modified time and requests the signature on a JWT with a wrong time, to the SE. In this way, the micro-controller obtains a signed JWT with a custom time field ("iat"), which is controlled by the attacker, and can be used in the future.

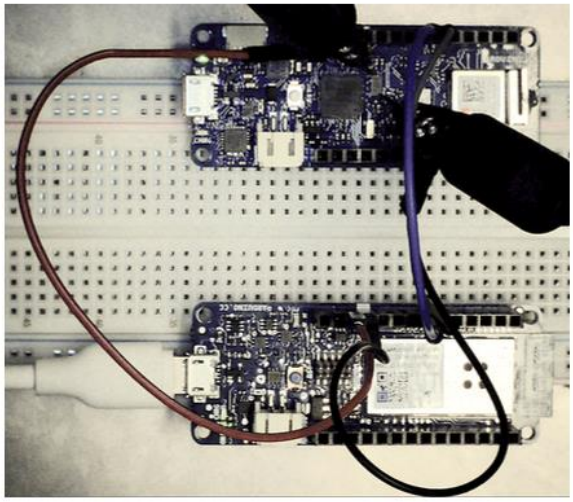
We observe that by targeting the Wi-Fi channel, only the JWT "iat" can be misused, and the attack is limited with respect to the one targeting the micro-controller-SE connection.

3.13 Attack

We present our attack by using the Arduino MKR WiFi 1010 (Ard19) as the victim's board. The attacker uses a second Arduino, the Arduino MKR 1000 WiFi, which is not provided with an SE, to abuse the SE of the victim and make it generate a JWT for authenticating to the Cloud. As shown in Figure 12(a), the SE on the MKR WiFi 1010 is easily accessible. The experiment setup is depicted in Figure 12(b). It shows that the GND, SDA and SCL signals are connected from the PINs of the attacker to the corresponding PINs on the victim's board, while the power (VCC) is connected from the PIN of the attacker, directly to the VCC PIN of the SE.



(a) Attack schematics



(b) Attack experiment

Figure 12: Attack schematics (a) and experimental setup (b)

We demonstrate our attack by using the sketch given by Arduino in (Ard19) to connect the MKR to the Cloud Provider, in particular the version for the Google IoT Cloud. In order to emulate a pre-provisioned keypair, on the victim board we first load the Arduino ECCX08JWSPublicKey sketch (Ard19) to generate the cryptographic material inside the SE. Further instructions on these procedures can be found in (ARD24).

The obtained public key (necessary to validate the signature of the JWT) is:

-----BEGIN PUBLIC KEY-----

MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEoEPtFj

VHxvodsOKutQilCP/5gCjzF1pzUU+3eqLLD255e4Q4

xOBCqOWwC4gY6l6fHFNLlLKYyV8jiqJzzaLJxA==

-----END PUBLIC KEY-----

After registering the corresponding public key on the Cloud provider, we load the Arduino GCP_IoT_Core_WiFi (Ard19) sketch to authenticate the node to the Cloud. This sketch executes the following steps: connects to the WiFi, retrieves the time information from the network, signs a JWT that contains the retrieved timing information in the “iat” claim, then uses the signed JWT to authenticate to the MQTT server to publish and retrieve information. An example of a legitimate JWT generated by the victim is provided in the next picture.

The attacker’s board performs the same actions, but it cannot authenticate to the server as it has no SE connected. So it needs to “steal” it from the victim’s board. In order to do so, she disconnects the power from the victim, and wires the two boards as depicted. With the GND and VCC from the attacker to the victim’ SE PINs, and the SDA and SCL signals to the corresponding signals of the victim’s board.

Once this is done, the attacker can use the SE to sign any JWT of her choice, as if the SE was on its own board. So what she can do for example is to create a set of signed JWTs for the future, by manipulating the time, and store them for further use. We have performed such an attack, and edited the “iat” claim by setting it to a date in the future. The corresponding obtained JWT is depicted below.

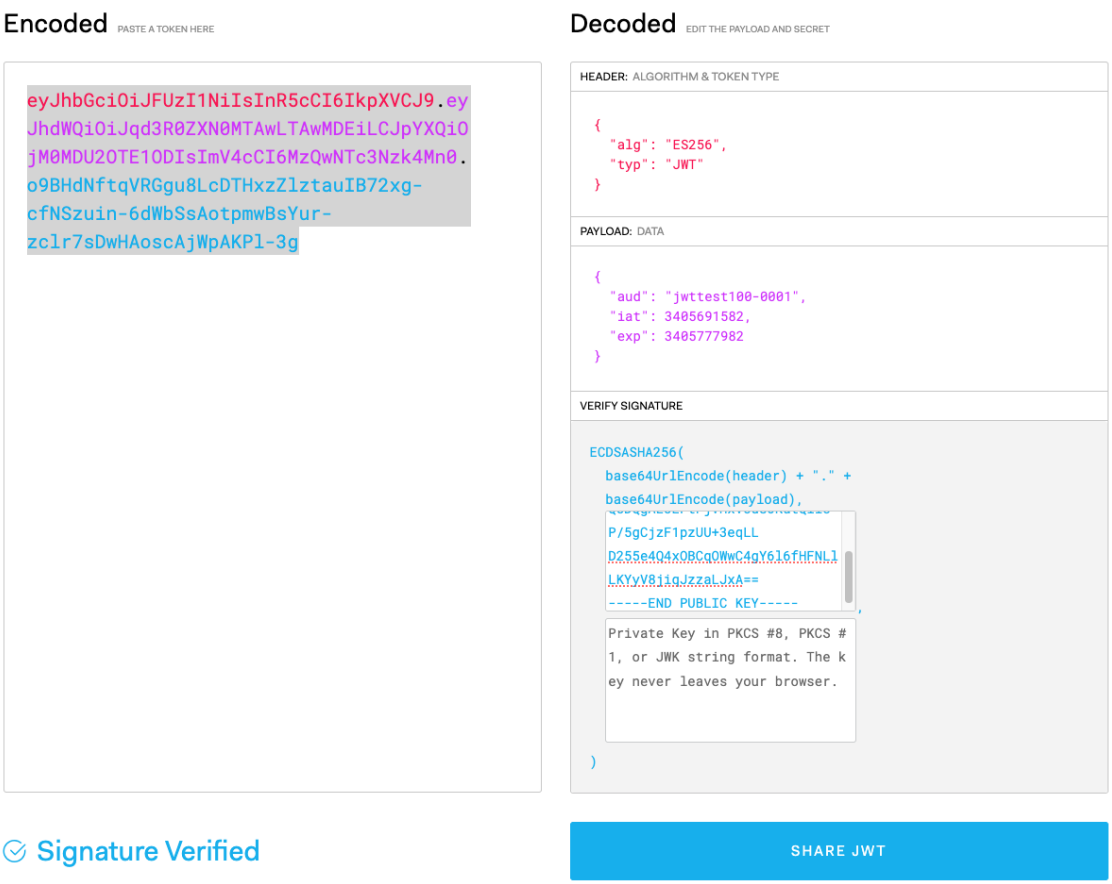


Figure 13: Example of a JWT generated by the attacker with “iat” claim set to a date in the future

We would like to stress that once the attacker obtains the JWTs, she doesn’t need the devices anymore, and the JWTs can be used on any other device, i.e. a software tool executing an MQTT (Sta19) client. Finally, as far as we know, if the attack is detected by the Cloud backend, the only possibility to mitigate it is to revoke all of the devices’ keys. Alternatively, if the devices’ private key can’t be changed, dispose of all the devices.

3.14 A JWT weakness

JWTs were initially conceived to be used to authenticate users between interconnected servers, where all servers were controlled by the same entity and no (or little) clock skew was possible between them. The attack described above is the result of the use of the JWTs in a different context, where the token producer and the token consumer are different devices, under control of different entities. Thus, probably, the JWTs mechanism would require some adaptation in order to be securely applied to such use cases. Alternatively, one can argue that the problem is present due to a weakness in the JWT standards. Such weakness can be identified in the lack of communication between the Cloud and the device during the JWT generation. One of the building blocks of many security protocols is a random nonce generated by the party that verifies the claims and included in the signature by the party that wants to prove its identity. The lack of such nonce in the JWT specification makes it difficult to prove the protocol security, and opens the way to attacks as the one demonstrated in this work.

3.15 Countermeasures

In this section we present possible countermeasures that can be applied to thwart our attack. The applicability of the countermeasure obviously depends on the particular setup, device at hand, supply chain condition, resources, etc.

3.16 Use the TLS authentication

One simple solution is to use the TLS authentication mechanism instead of the JWT one. This has the obvious drawback that not all Cloud providers allow it, and of a heavier authentication mechanism. However it allows to thwart the attack as the nonces included in the TLS handshake prevent attackers from reusing previous commitments or generating JWTs valid in the future.

3.17 Pre-provisioned keys

A second solution would be to take advantage of the presence of pre-provisioned private-keys inside the SE. For example, many commercial SEs contain a private key pre-provisioned by the founder (commonly referred to as attestation key). This countermeasure assumes that such pre-provisioned keys are installed by a trusted party, in a trusted environment. The attestation key can only be used to sign data internal to the SE or data internal to the SE and some additional external bytes. The module also provides slots to store additional self generated private keys, which can be used to sign external data. We suggest the following protocol to thwart the attack described in this work. During provisioning, the public key corresponding to the attestation key is published to the Cloud. After deployment, the Cloud sends a random nonce to the SE. After receiving the nonce, the SE generates a private key in one of the free slots. Then, the SE uses the attestation key to sign the digest of the public key corresponding to the private self-generated key, and the nonce received from the Cloud. The signature is sent to the Cloud, which, upon verification, updates the public key to use during JWT verification with the one just received. THen, after this first “provisioning” phase, the signature of subsequent JWTs can be signed by the self-generated private key, without need for further nonces from the Cloud. In such a way the Cloud can trust that the JWTs have been generated only after the nonce has been communicated during the “provisioning” phase. This solution has the advantages of avoiding the need to implement a TLS stack on the micro-controller, and offering a robust mechanism to update the device’s private keys in the field. The drawback of this solution is that a new step needs to be implemented in the Cloud provider’s back-end, and the device must implement the new protocol. Another advantage of such countermeasures is that compromised keys can be revoked and compromised devices can be re-provisioned, simply by generating another signing key inside the SE and repeating the nonce procedure.

3.18 Use the nonce claim

A fourth solution to thwart the attack is similar to the previous one, but does not require the use of pre-provisioned keys in the SE module. It simply requires the Cloud to send the nonce to the device after deployment, prior to creating JWTs. Once received, the device includes the Cloud’s nonce into the JOSE header of the JWTs. Afterwards, each JWT must contain the signed JOSE header containing the nonce, thus providing an assurance to the cloud concerning the generation time of the JWT. The main advantage of such a solution is that it is the simplest solution to implement. The drawbacks involve a new step to be implemented in the Cloud provider’s back-end. Similarly to the solution with pre-provisioned keys, a further advantage is that compromised keys can be revoked and compromised devices can be re-provisioned by generating new sign keys and refreshing the nonce step.

3.19 Key use limit

A fifth solution takes advantage of the presence of monotonic counters in the SE, paired with a private key slot. Each time the key in the paired slot is used, the counter is incremented. By using an additional JWT field, the micro-controller can include the value of the counter in the JWT. The Cloud then verifies that the value in the counter is monotonic increasing, in order to mitigate the attack described in this work. The advantage of this solution is that it is simple and does not require a TLS stack on the micro-controller. On the other hand, a new step is to be implemented in the Cloud provider's back-end, and a new field needs to be implemented in the JWT. Furthermore, differently from the solutions proposed above, this solution does not allow revocation of the used key if it is compromised.

3.20 Trusted supply chain

One final mention has to go to the simplest solution of all, at least from the technological point of view, which is of having a secure supply chain, or at least partially secure supply chain. Definition of a trusted supply chain is out of scope of this work, however we consider a supply chain to be trusted if it consists of a comprehensive and verifiable system encompassing all stages of an IoT device's lifecycle – from design and component sourcing to manufacturing, distribution, deployment, operation, maintenance, and eventual disposal – that ensures the security, integrity, and resilience of the device and its associated data throughout its entire existence. This would allow the first steps of the personalization of the SE to be performed in a trustworthy environment. The following steps, where the attack can't be mounted anymore, can be delegated to an untrusted supply chain. This solution, however, has an intrinsic drawback of dividing the production into two different sites, the secure and insecure one, with obvious costs and troubles.

3.21 Conclusions

This work presents JWT Back to the future, the possibility of (ab)using an IoT device during production for preparing credential claims (JWTs) that will be later used by the malicious actor to connect to the Cloud service. A malicious user in the supply chain might have the possibility to collect a large number of JWTs for mounting a massive attack in the future.

The flaw is related to the lack of a mandatory nonce in the JWT standard. We show that it is possible to take advantage of such a flaw to abuse the authentication of IoT devices in the field, when JWT is used. In particular we show that an attacker in the supply chain may be able to make the IoT device generate some JWTs which are valid in the future and use them afterwards to impersonate the device with respect to the Cloud Provider.

Interestingly, we describe our attack against devices with different architectures. Showing that a Secure Element (SE) attached to the micro-controller cannot protect from such an attack. We present a practical attack against an off-the-shelf IoT device by Arduino (Ard19) and the mechanism used by the Google Cloud IoT Core to authenticate the devices. We demonstrate that an attacker on the production line can request a number of signatures on self generated JWTs, from the SE connected to the micro-controller, to be used in the future to connect to the Cloud.

We showcase our attack on the - now retired - Google Cloud IoT Core, in order to avoid malicious use of our findings, but our discovery can be applied to other services that provide token-based authentication. For example we further show that the same weaknesses apply to other tokens like the CWT and the EAT, and to platforms like HiveMQ and EMQX (HiveMQ; EMQX;LMOW24;JWET18) providing a much wider attacker scope than merely a single token type or Cloud provider. Furthermore, our attack also applies to the OCMF standard, used for recording meter readings from charging stations for eV[Se].

In order to thwart the presented attack we provide a few countermeasures that can be applied, depending on the IoT infrastructure at hand. The simplest countermeasure is to use TLS

authentication, while other countermeasures involve implementing a nonce-like mechanism in the JWTs or the authentication itself.

Chapter 4 A New Secure Channel Protocol

4.1 Overview

In this chapter, the NSCP (New Secure Channel Protocol), a novel secure communication protocol designed for industrial IoT environments, will be presented. It focuses on enhancing the security of the connection between microcontrollers and Secure Elements while improving efficiency compared to SCP03, the current industry standard. By leveraging the Xoodyak cryptographic primitive, NSCP achieves strong security with significantly lower computational overhead, making it ideal for resource-constrained devices.

The Internet of Things (IoT) has become a popular technology in the industrial sector that demands high reliability, robustness, and security. In industrial IoT, security can have various implications. For example, to connect to the cloud, IoT devices need to handle sensitive information such as pre-shared secrets or certification authority's PKI certificates. As another example, security is crucial for continuous operation and proper machinery functionality if we think about an attacker tampering with simple sensors that drive any type of industrial plant. Therefore, protecting IoT devices against both remote and physical threats is becoming increasingly important.

An effective way to improve security in the Internet of Things is by incorporating secure integrated circuits, also known as Secure Elements (SE). These specialized hardware components are certified to resist tampering and serve as secure storage for confidential information and cryptographic operation. A secure element typically provides its services over serial interfaces by creating protected channels using standardized protocols such as GlobalPlatform's Secure Channel Protocol (SCP). SCP03, in particular, is a resource intensive protocol based on shared secrets that is typically used in such environments. However, as we will show, it may not be optimal when bandwidth (such as sensor data protection) is at stake.

In this chapter, we propose an alternative to SCP03 consisting of a lightweight secure channel protocol that utilizes Xoodyak (Xoodyak), a cryptographic primitive known for its efficiency and minimal resource requirements. The new protocol aims to simplify the operational framework to provide adequate security while maximizing throughput.

The remainder of this chapter is organized as follows. First, we provide an overview of related work and existing secure channel protocols with a focus on SCP03. Next, we detail the design and implementation of the proposed protocol, called the New Secure Channel Protocol (NSCP), highlighting its key innovations. This is followed by a comprehensive performance evaluation on an ARM-based STM32 microcontroller and a RISC-V one, including comparisons with SCP03 on metrics such as throughput and memory occupation. Finally, we conclude with a discussion of the findings and potential future research directions.

4.2 Background

The protocol proposed in this chapter aims to present a suitable alternative to SCP03 for embedded devices. For this reason, we will first cover the fundamentals around SCP03 then discuss the Xoodyak stateful object on which our protocol is based.

4.2.1 The Secure Channel Protocol

In this section we present the Secure channel protocol. Despite having presented it in Chapter 2, we recall it here as we recall some more in-depth explanation, necessary for understanding the rest of the work.

The Secure Channel Protocol (SCP) is a suite of protocols published by GlobalPlatform (GP), a technical organization focused on the standardization of secure component technologies

(GlobalPlatform); SCP03 (SCP03)(Card Specification) is the latest iteration of SCP protocols that operate with symmetric encryption and authentication primitives to exchange data between a host MCU and a secure element (SE) over a bus such as I2C. SCP is built above ISO/IEC 7816, a standard used to represent command / response messages as application program data units (APDUs). In this short exposition, we will gloss over the packet encoding format and headers and focus on the main cryptographic functions implemented in the protocol.

In SCP03, the host typically specifies the security level for the subsequent command APDUs (C-APDUs) and response APDUs (R-APDUs). The security level might correspond to either message authentication only, or both message authentication and encryption. Encryption is performed using AES in Cipher Block Chaining (AES-CBC) (AES-CBC) mode while authentication is achieved by appending an 8-byte (or, in some versions, 16 bytes) Message Authentication Code (MAC) produced by an AES-based CMAC (AES-CMAC).

During the SCP handshake (see Figure 14), the host and the secure element use pre-shared keys (K_{enc}, K_{mac}) to generate three session keys: $\{S_{enc}, S_{mac}, S_{rmac}\}$. These keys are used for encryption and to authenticate the following commands and responses. The generation of session keys is done with a specific command sent by the host (initialise update) that carries a host challenge v_h (nonce). The secure element generates its own challenge v_s and computes the session keys S_* using a CMAC-based key derivation function (KDF) (AES-CMAC). Then it uses S_{mac} to produce a cryptogram χ_s with another KDF based on CMAC. Finally, χ_s is sent back to the host together with v_s.

To validate χ_s, the host attempts to regenerate it using the pre-shared keys k_* and the exchanged challenges v_*. Upon successful validation, the host uses the external authenticate command to send its cryptogram χ_h to the secure element, which will validate it similarly. At the end of this process, apart from having proven that they have the same pre-shared keys (k_*), both the host and the secure element have agreed on common session keys (S_*) derived from the shared base keys.

The external authenticate command is also used by the host to specify one of the five security levels of the following communication:

1. Level 1: authentication of commands.
2. Level 2: encryption and authentication of commands
3. Level 3: authentication of commands and responses.
4. Level 4: encryption and authentication of commands; authentication of responses.
5. Level 5: encryption and authentication of commands and responses.

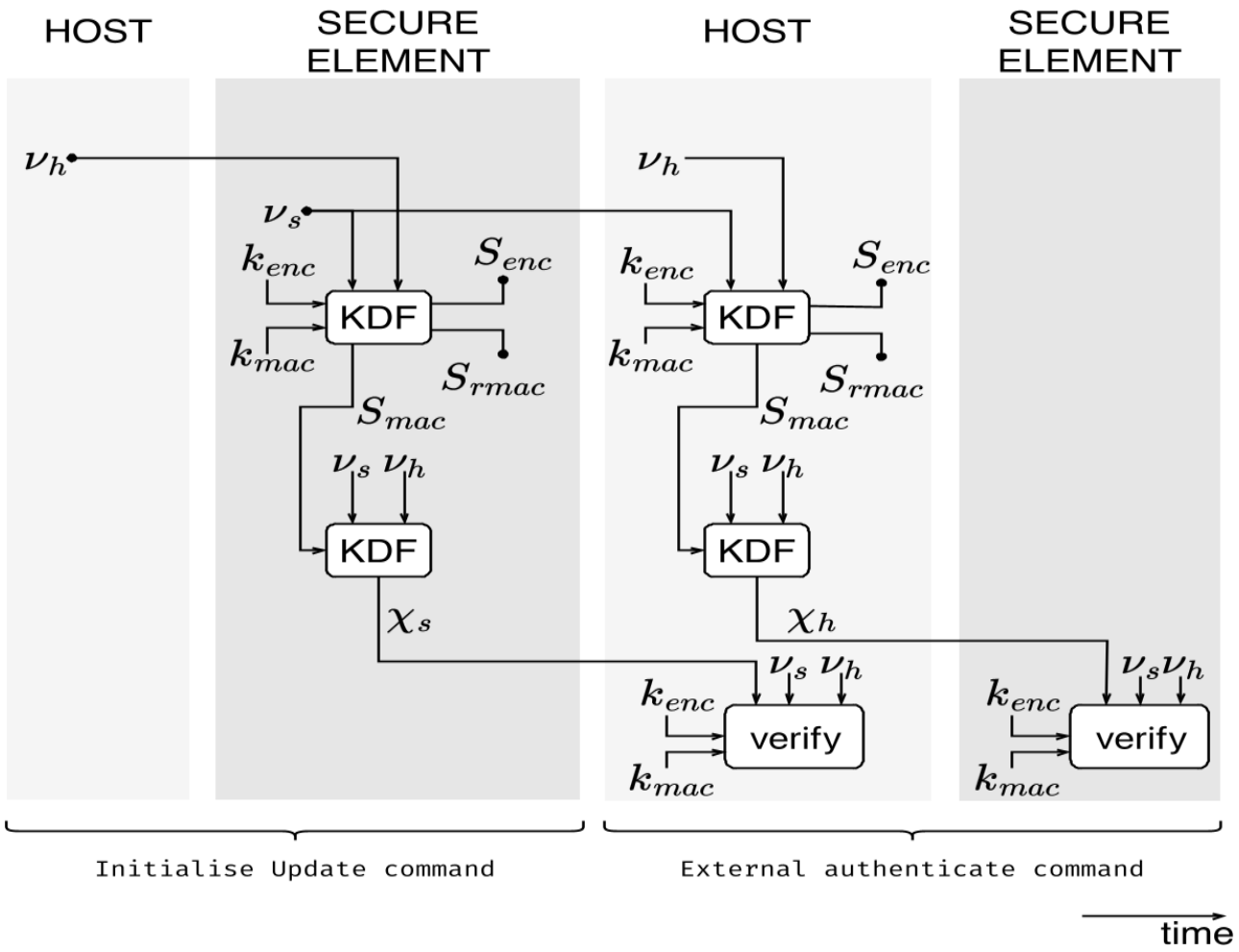


Figure 14: Handshake in SCP03

The aim is twofold: 1) to generate shared session keys S_* from exchanged nonces ν_* and 2) validate the presence of shared secrets between host and SE (i.e. the base keys k_*) through cryptograms χ_*.

To ensure confidentiality, SCP03 employs an "Encrypt-then-Authenticate" method (refer to Figure 15) in which encryption is carried out using AES-CBC.[2]. The initial chaining value (ICV) used by AES-CBC ($iv(n)$) depends on the current message counter n. The counter increases with each command sent from the host to the secure element, making it dependent on the number of commands sent n. Using $iv(n)$, identical payloads within the same session will be encrypted differently, preventing chosen plaintext attacks (CPA).

[2]AES-CBC is a stream cipher built from AES which works as follows: given an arbitrarily long message M decomposed in a sequence of N 16-byte blocks {mi}, it produces recursively the encrypted representation yi of mi as

$$yi=AES(miyi-1,S),\ y0=iv$$

where S is the session key and iv is the initial chaining value.

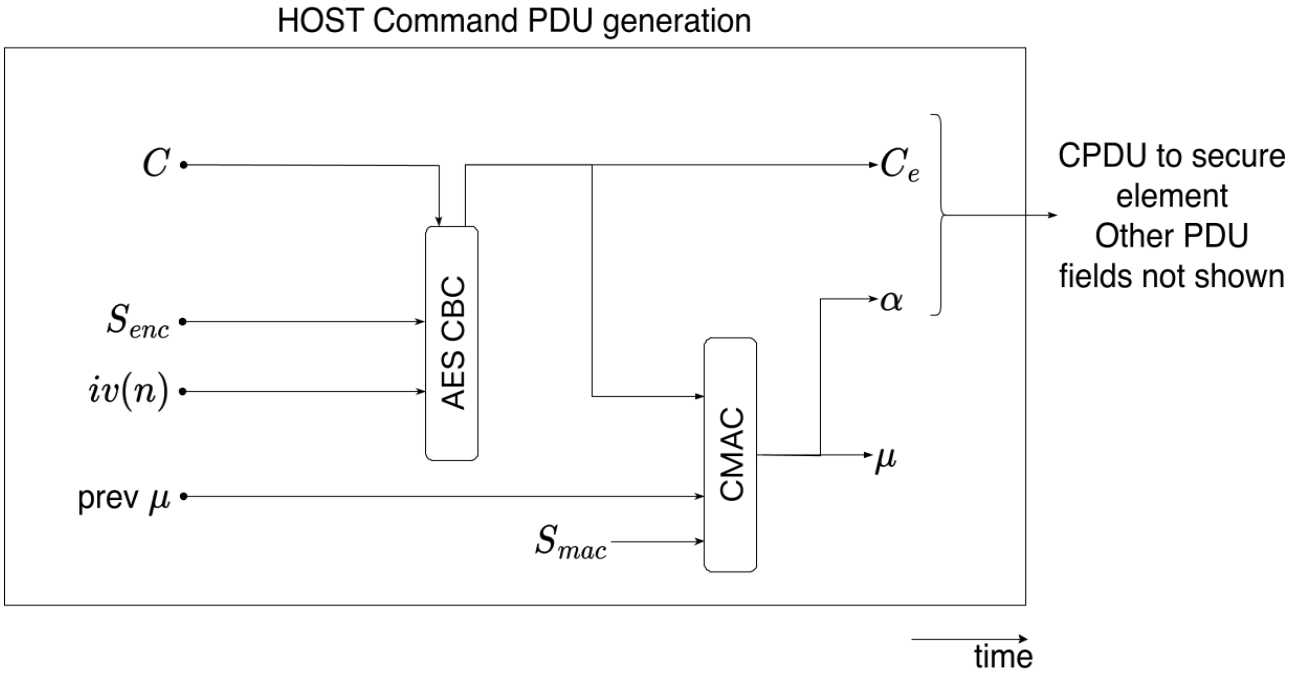


Figure 15: SCP03 encrypt then MAC applied to command PDUs sent from the host

On the left, C is the command to be sent, S_{enc} is the encryption key, $iv(n)$ is a session dependent initialization value, μ is the MAC chaining value (16 bytes, at the beginning $prev\ \mu$ is 0). On the right C_e is the encrypted command, α is the authentication tag (upper 8 bytes) of μ, S_{mac} is the message authentication key, μ is the new MAC chaining value (used in the next session).

For message authentication, authentication tags are generated using MAC chaining values. At any moment, the MAC chaining variable in the host (μ in Figure 15) guarantees the integrity of the command sequence produced by the host. In a way, it can be thought of as a summary of the session's history. The authentication tag α associated with the message is just the most significant 8 bytes (or, in some versions, 16 bytes) of the current chaining value μ which is computed from the previous one with the current command ciphertext and the S_{mac} key. Using such a chaining value "captures" the entire command history up to message n. This effectively nullifies attempts at replay attacks, as two identical commands or responses will have different authentication tags. Thanks to its design, SCP03 has been proven secure against replay attacks, out-of-order attacks, algorithm substitution attacks, and more (SCP CardLogic;SCP Cryptanalysis).

The exchange of C-APDUs and R-APDUs between a secure element and a host can be seen as a special case of Authenticated Encryption with Associated Data (AEAD). APDUs can be conceptually divided into two parts: one that is always sent in clear (e.g., the header field) and another which may be encrypted (like the message payload). While encryption is applied only to one part of the message (the payload), verification of integrity is typically applied to the entire message, that is, even to the associated data that are sent in clear. In recent years, newer AEAD algorithms have been proposed that might promise better performance and productivity trade-offs with respect to the current SCP standard. One of them is based on the sponge construction and is called Xoodyak (Xoodyak) and is the cornerstone of the protocol proposed in this chapter. Xoodyak was selected as a finalist of the NIST Lightweight Cryptography standardization process (NIST-LWC), due to its adaptability and outstanding performance, as also evidenced by an evaluation conducted on the NIST-LWC finalists using RISC-V architectures (CLMLD19).

4.2.2 The Xoodyak Primitive

We briefly recall here some information on Xoodyak, for ease of reading. Xoodyak is a lightweight cryptographic primitive that employs a special mode of operation called Cyclist, and an underlying

permutation called Xoodoo. Cyclist should be seen as a stateful object with a set of methods[3] that manipulate the state so that it is always a reflection of the history of all preceding method invocations. Some of these operations can apply the Xoodoo permutation to the state itself so as to accomplish various cryptographic functions according to the two operating modes, namely:

1. **Hash mode.** Once the Cyclist object is initialized with constant values, Xoodyak can be used to absorb an input string (of arbitrary length) and produce ("squeeze") hash digests of the said input strings (methods absorb(), squeeze()).
2. **Keyed mode.** In this case, the Cyclist object is initialized with a key K. In this mode, Xoodyak is capable of performing, among other things, MAC computations and encryption / decryption (methods encrypt(), decrypt().

Xoodyak does not have any parameters that can be chosen by the user. This means that the user does not decide on the number of permutation rounds or the method of padding and splitting the input into blocks before encryption. Xoodyak utilizes the Xoodoo permutation, which operates over a 384-bit state divided into 12 rounds and can be stored in 12 registers of 32 bits each, making it ideal for low-end 32-bit devices. Among its methods, it includes a ratchet mechanism (method ratchet()) that zeros out part of the state. This feature is beneficial in scenarios involving certain types of attack, such as side-channel attacks, which could allow an attacker to retrieve the internal state of the cipher. In such situations, the attacker cannot discover the secret key after the ratchet is applied. Therefore, the ratchet mechanism is said to provide forward secrecy, at least within the context of a single session.

4.3 NSCP: a new secure channel protocol for hardening communications in industrial IoT

A secure element safeguards cryptographic keys and provides cryptographic services to an IoT-integrated microcontroller (MCU). These elements are specifically bound to the MCU, which ensures that only the MCU can access their security services. This binding can occur at various manufacturing stages and can be adjusted for different security levels based on the device's needs and MCU features.

Our proposal for NSCP is designed for connecting a secure element to a host MCU and meets several requirements. Firstly, it ensures security against major threat models. Secondly, to support secure intensive applications, the protocol is fast, and efficient on low-end devices as it exploits the Xoodyak object for lightweight cryptography. Concerning the threat model, here we assume that the attacker can perform any type of eavesdropping/tampering with the communication buses between the host MCU and the secure element (Murdoch07). However, both the MCU and secure element are assumed to have defenses against physical attacks, including side-channel attacks (e.g., masking (PBSHCSL23) and fault injection to protect pre-shared keys. Performance-wise, NSCP has been conceived to operate under principles simpler than SCP03. In keeping with this goal, the new protocol offers only one possible level of cryptographic protection for APDUs, which corresponds to the highest one that can be specified through the 'External Authenticate' command in SCP03: the level which demands that all messages be encrypted and authenticated.

[3] Think of it as a C++ object.

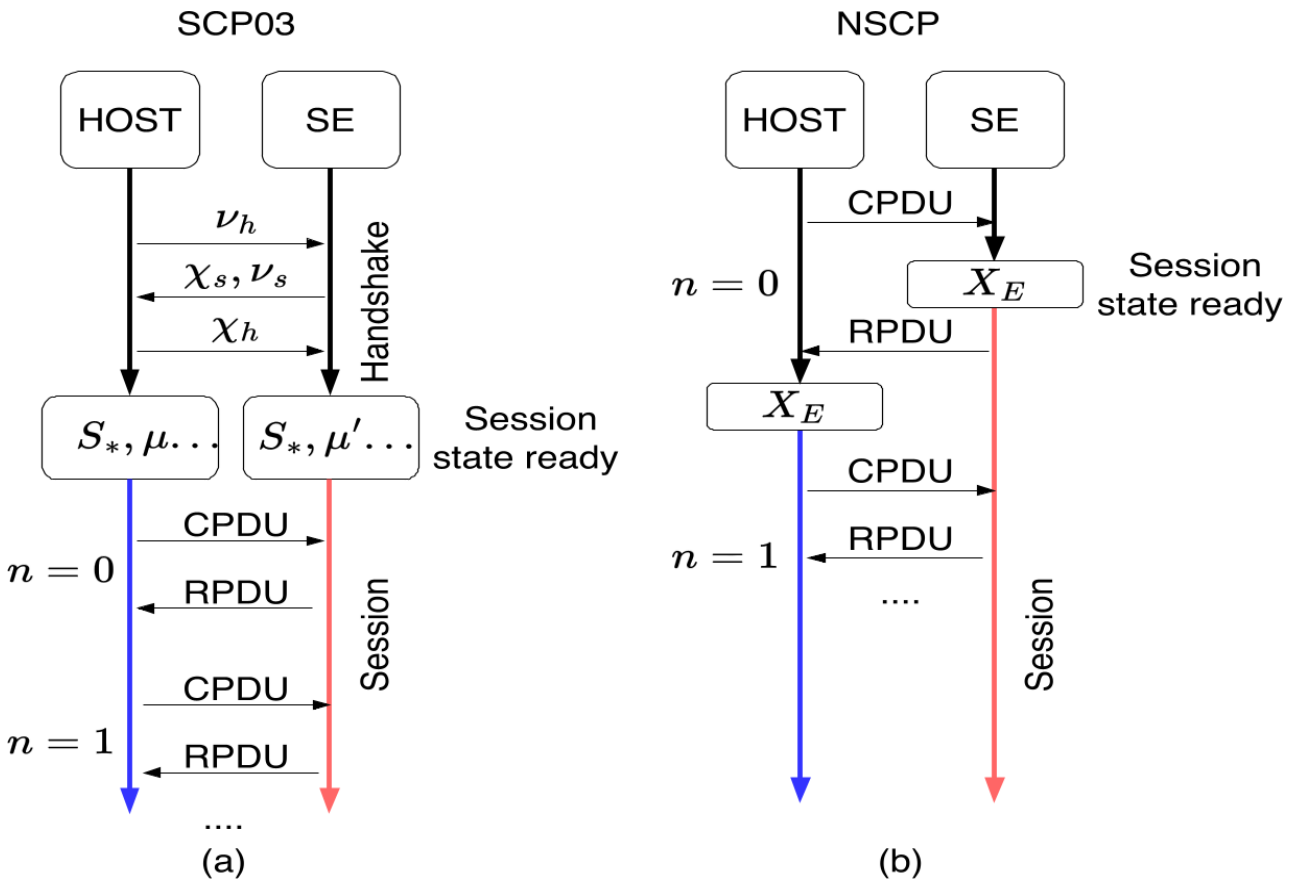


Figure 16: High-level comparison of session management in SCP03 and NSCP

SCP03 needs a handshake phase to mutually authenticate the host and the secure element. In NSCP, a valid initial session state X_E in both host and the secure element ensures they are mutually authenticated and the states in both of them evolve in a mirrored way.

SCP03 and NSCP also differ in the way sessions are established (see Figure 16). SCP03 uses a single handshake to exchange nonces and cryptograms for mutual authentication and session keys S_* (see Figure 16(a)). NSCP, however, uses the Xoodyak object's state X as the session state[4], initializing its value during the first APDU exchange before the RPDU is created by the secure element (see Figure 16(b)).

[4] In the following description, we will add a subscript to identify a specific point in time in which that state must be considered, so XC must be considered the state of the Xoodyak object at stage C.

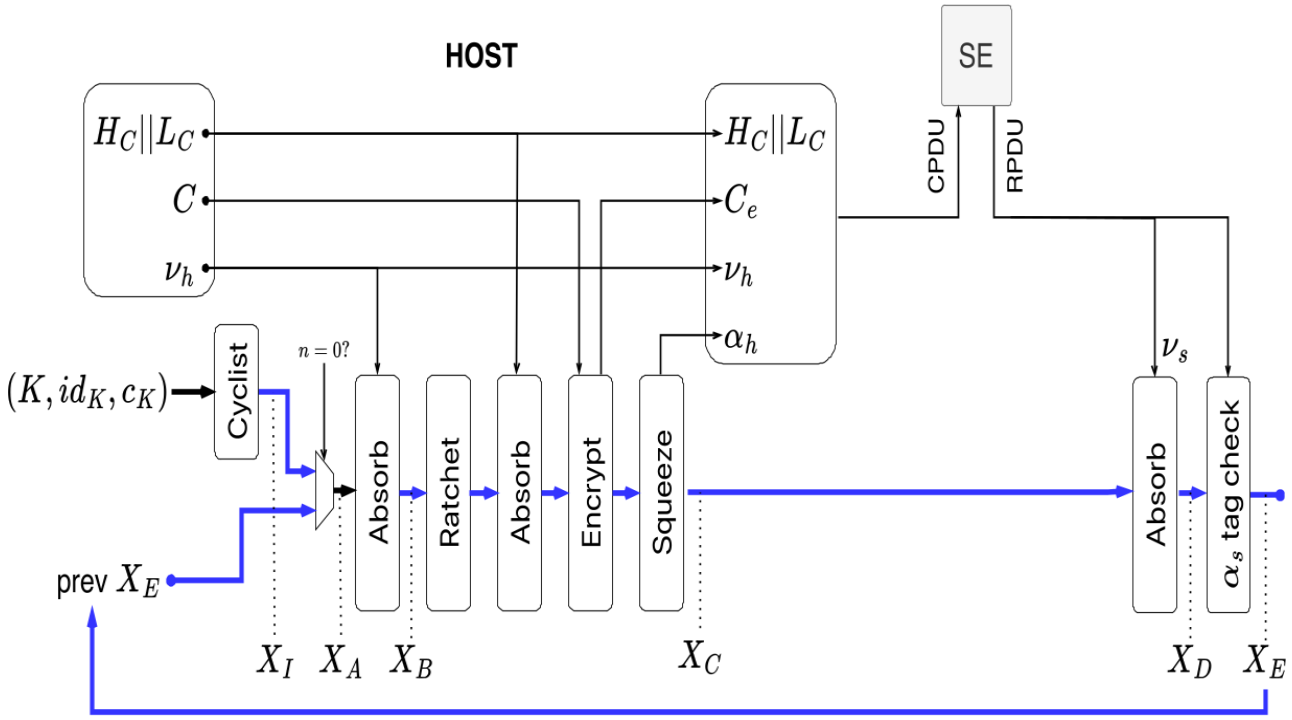


Figure 17: NSCP internal data on the host

When communicating with the secure element SE: C represents the plain text command, whereas C_e denotes the corresponding encrypted command. H_C is identified as the command header, and L_C specifies the command's length. ν_h (ν_s) is the 128-bit nonce generated by the host (secure element), and α_h is the authentication tag (128 bits), K is the pre-shared key (128 bits), id_K is the identifier of the key while c_K is the session counter for that particular key. X is the intermediate state of the session (i.e., the Xoodyak state), also highlighted in blue color. Note that Cyclist is initialized only at the beginning of the session (X_I), otherwise the previous state X_E of the session is used as the initial state X_A of the n-th message exchange.

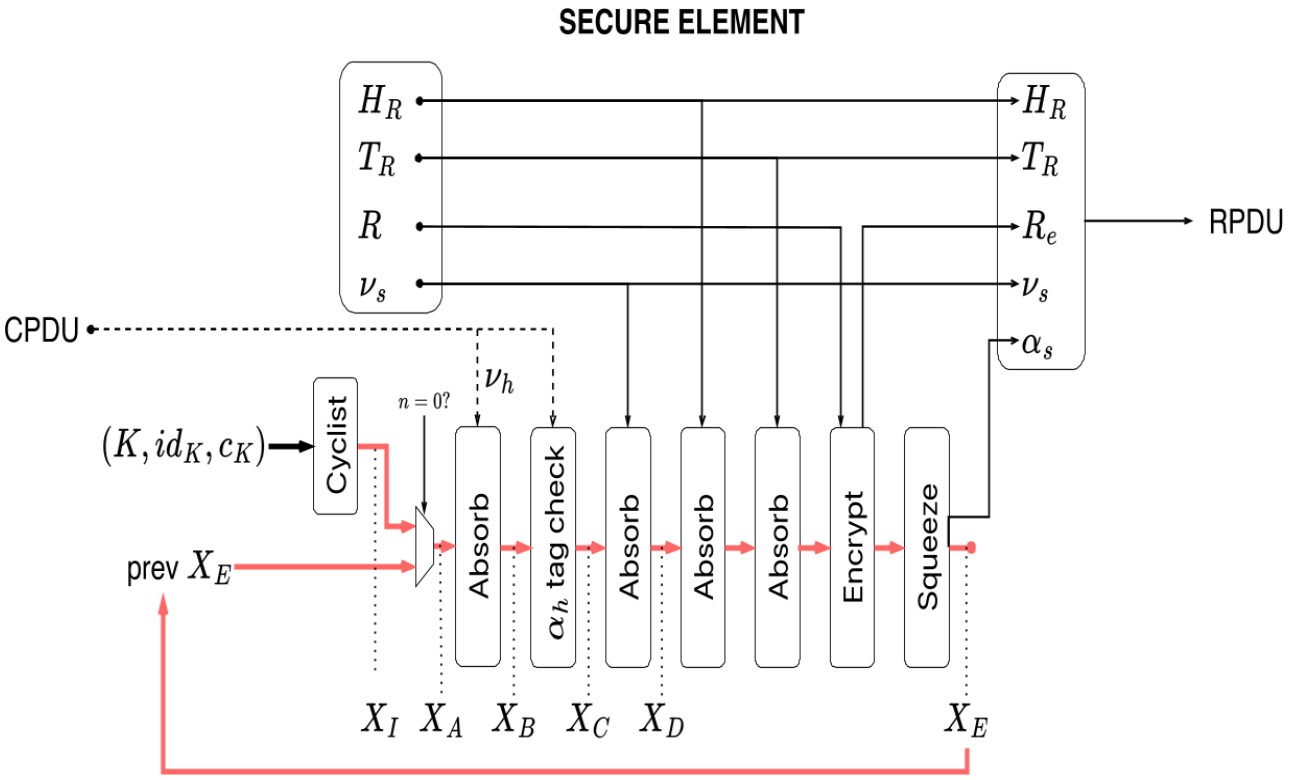


Figure 18: NSCP internal data on the secure element

Delineated as follows: R denotes the plain text response, and R_e represents its encrypted counterpart. H_R refers to the response header, whereas T_R indicates the response trailer. Additionally, ν_s is the nonce generated by the secure element (128 bits), and α_s is the authentication tag (128 bits). Note that the Xoodyak state of the secure element (red) mirrors the one on the host.

In a valid session, both host's and secure element's state evolve synchronously through Xoodyak's methods (see Figure 17 and Figure 18. The initial CPDU/RPDU exchange initializes the Xoodyak state X to X_I for both the host and the secure element. Subsequent CPDU/RPDU exchanges use the final state X_E of the exchange $(n-1)$ as the initial state X_A for the exchange n (see feedback loops in Figure 17 and Figure 18). The initial session state X_I is created by both the host and the secure element with the Cyclist constructor in keyed mode, using the static key K (128 bits as suggested in (Xoodyak), the key identifier id_K and a static key usage counter c_K (incremented with each static key use).

During the session in the host (see Figure 17), the entropy of the initial state X_A increases by absorbing a 128-bit nonce ν_h (as suggested in (Xoodyak) then the state X_B is irreversibly transformed by a cryptographic Ratchet. After absorbing the APDU header and length, the session state is used as a key to encrypt the command C into C_e. In addition, it is also used to generate an authentication tag α_h (128 bits, as suggested in (Xoodyak) that is used (by the secure element) to verify the integrity of the communication. At this point, the secure channel is in the state X_C and the C-APDU is sent to the secure element.

The secure element (see Figure 18), after extracting the associated data ν_h from the C-APDU, can validate the integrity of the message and reconstruct the state of the session X_C as it ended on the host. This is then used to absorb a nonce generated internally (ν_s) and encrypt the response payload. In addition, it also produces an authentication tag α_s that the host will use to validate the integrity of the response. Once the host has received the RPDU (Figure 17, right), it absorbs the nonce ν_s. At this point, it has reconstructed state X_D and is thus able to both verify the integrity of the entire RPDU and decrypt the response payload (tag check block). If successful, both the host state and the state

of the secure element have reached the state X_E. In the next exchange, this will become the initial state X_A[5].

4.4 Evaluation

The purpose of this section is to corroborate, through a benchmarking study, that NSCP outperforms SCP03 even in its highest security and integrity settings[6] , primarily due to the lighter cryptographic primitive and simpler design.

4.4.1 Experimental setup

The performance of both protocols was tested using a RaspberryPi acting as host in two scenarios, one emulating a sophisticated secure element with an ARM Cortex M4 at 168 MHz (STMicroelectronics STM32f439zi SoC) and the second emulating a smaller factor secure element with a RISC-V core (OpenHW CV32E40P) synthesized on an Artix-7 FPGA (clock frequency 10MHz).

The secure element and the host MCU are connected via the I2C bus in normal speed mode (up to 12.5KB/s). The devices were programmed to simulate a real-life scenario of a host and a secure element exchanging data through a secure channel using the NXP's nano-package library [NXPNano]. To enable extensive benchmarking of protocols, we introduced a new 'Echo' command-response APDU pair and used the chaining functionality of the ISO / IEC 7816-3 (T = 1) protocol to exchange messages of arbitrary length and measure burst performance by varying burst size.

The measurement campaign aims to assess the overall throughput of secure element protocols, evaluating wall clock time of critical functions divided by bytes processed, with SCP03 focusing on CMAC, AES-CBC decryption/encryption, and ICV generation, while NSCP focuses on command authentication/decryption and response encryption/authentication.

In the ARM Cortex M4 scenario, we observed that the new protocol significantly improves handshake execution time, reducing it from more than 8.2×10^4 (SCP03) to approximately 2.2×10^4 clock cycles (for NSCP), achieving a speed-up factor of about 3.7.

[5] Obviously, such mirroring can be subject to rare problems where the host and secure element go out of sync, aborting the current session and creating a new one. Note that also SCP03 suffers from this problem as both host and secure-element MAC chaining values must evolve synchronously.

[6] Corresponding to the activation of all SCP03 options C_MAC, R_MAC, C_DECRYPTION and R_ENCRYPTION.

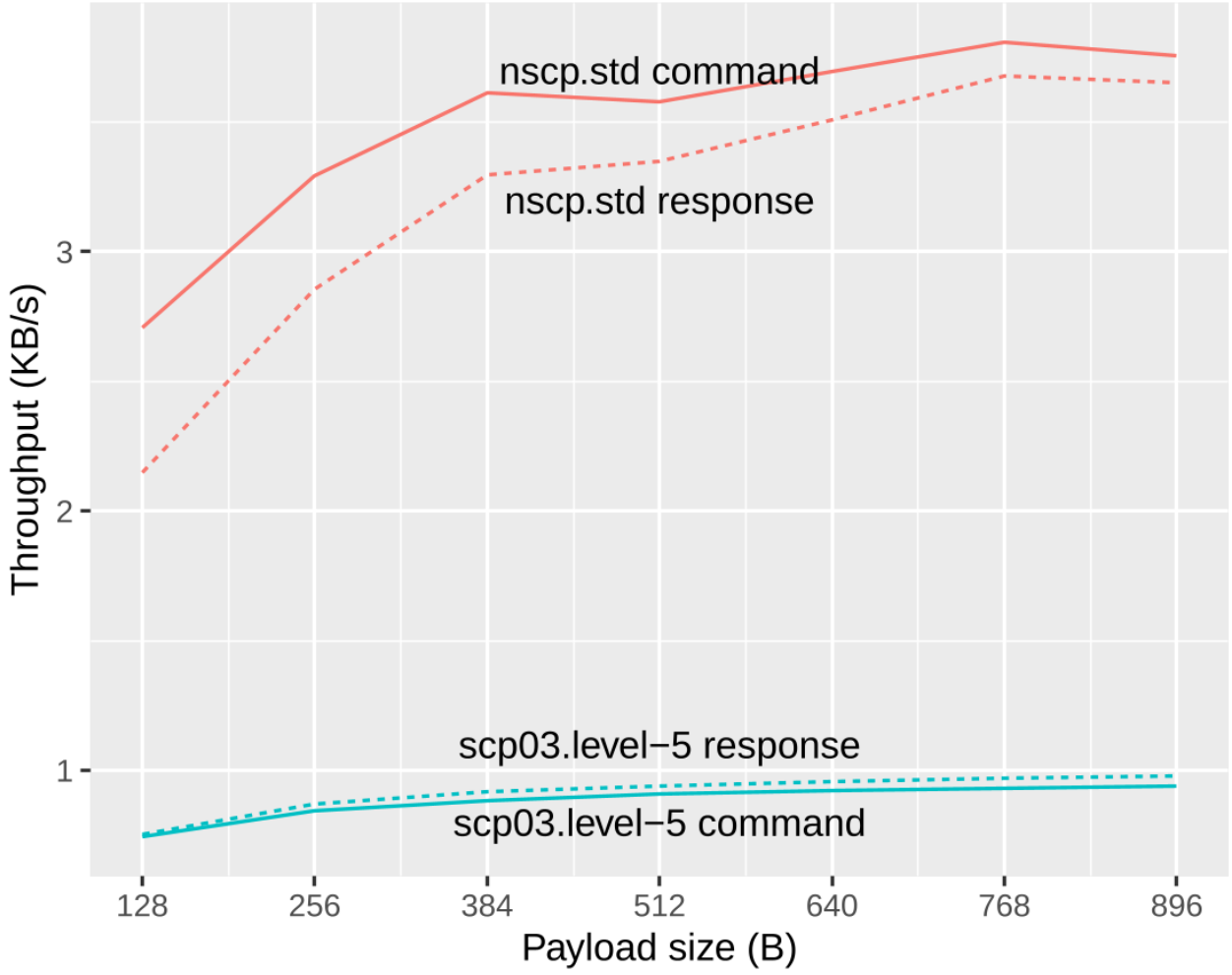


Figure 19: Throughput comparison of NSCP and SCP03 level-5 across various payload sizes (host: RPI, SE: ARM Cortex M4, bus: I2C)

Figure 19 illustrates the performance of NSCP throughput (denoted nscp.std) compared to SCP03 at security level 5, in different payload sizes (ranging from 128 to 896 bytes) and type (CPDU vs RPDU). The Y-axis measures throughput in KB/s, while the X-axis depicts the payload size in bytes.

On average, NSCP consistently demonstrates higher throughput relative to SCP03, ascending from 2.4 KB/s at the smallest payload size (128 bytes) to about 3.7 KB/s at the highest payload size tested (896 bytes). NSCP however presents different performance on command and response, which are probably due to the additional actions that the secure element must perform to validate the tags. This difference tends to attenuate as the packet size increases, probably because tag verification becomes negligible. The speedup of NSCP on SCP03 ranges from approximately 3.64x to 4.0x.

Table 3 provides a comparative analysis of resource utilization between NSCP and SCP03, when implemented on an embedded ARM M4 architecture. The comparison examines several aspects of memory consumption, namely SRAM, FLASH, and designated memory sections (.rodata, .data, .bss, .text). From the data, it is evident that NSCP is significantly more efficient across all categories. Specifically, NSCP reduces SRAM usage by 25%, FLASH by 37%, and shows substantial savings in the .rodata, .data, .bss, and .text sections — 94%, 17%, 28%, and 27% respectively, compared to SCP03. These savings indicate that NSCP not only requires less volatile (SRAM) and nonvolatile memory (FLASH), but also optimizes the storage of read-only data, initialized global variables, zero-initialized data, and executable code segments. The practical implication of this efficiency is that NSCP could offer more headroom for other functionalities on resource-constrained embedded systems.

Table 3: Benchmarks of NSCP vs SCP03

Resource usage comparison between NSCP and SCP03. Positive delta values correspond to a reduction in resource consumption of NSCP with respect to SCP03.

Protocol	SRAM	FLASH	.rodata	.data	.bss	.text
nscp.std	10047	31667	416	176	8346	30587
scp03.level-5	13350	49912	7475	212	11592	41789
difference	25%	37%	94%	17%	28%	27%

To corroborate our findings on a different architecture, we also benchmarked an RPI+RISC-V platform where the payload size range was limited to smaller values (slightly higher 256 bytes) due to the fixed I2C buffer size in the synthesized core. As Figure 20 shows, NSCP still offers significantly higher performance throughput compared to SCP03 in all payload sizes tested. Again, the disparity in throughput between command and response can be attributed to the additional validation operations performed by the secure element when processing responses, a pattern observed consistently across all presented payload sizes. SCP03 throughput exhibits a more moderate growth trajectory, starting from near zero and eventually reaching slightly more than 2KB/s.

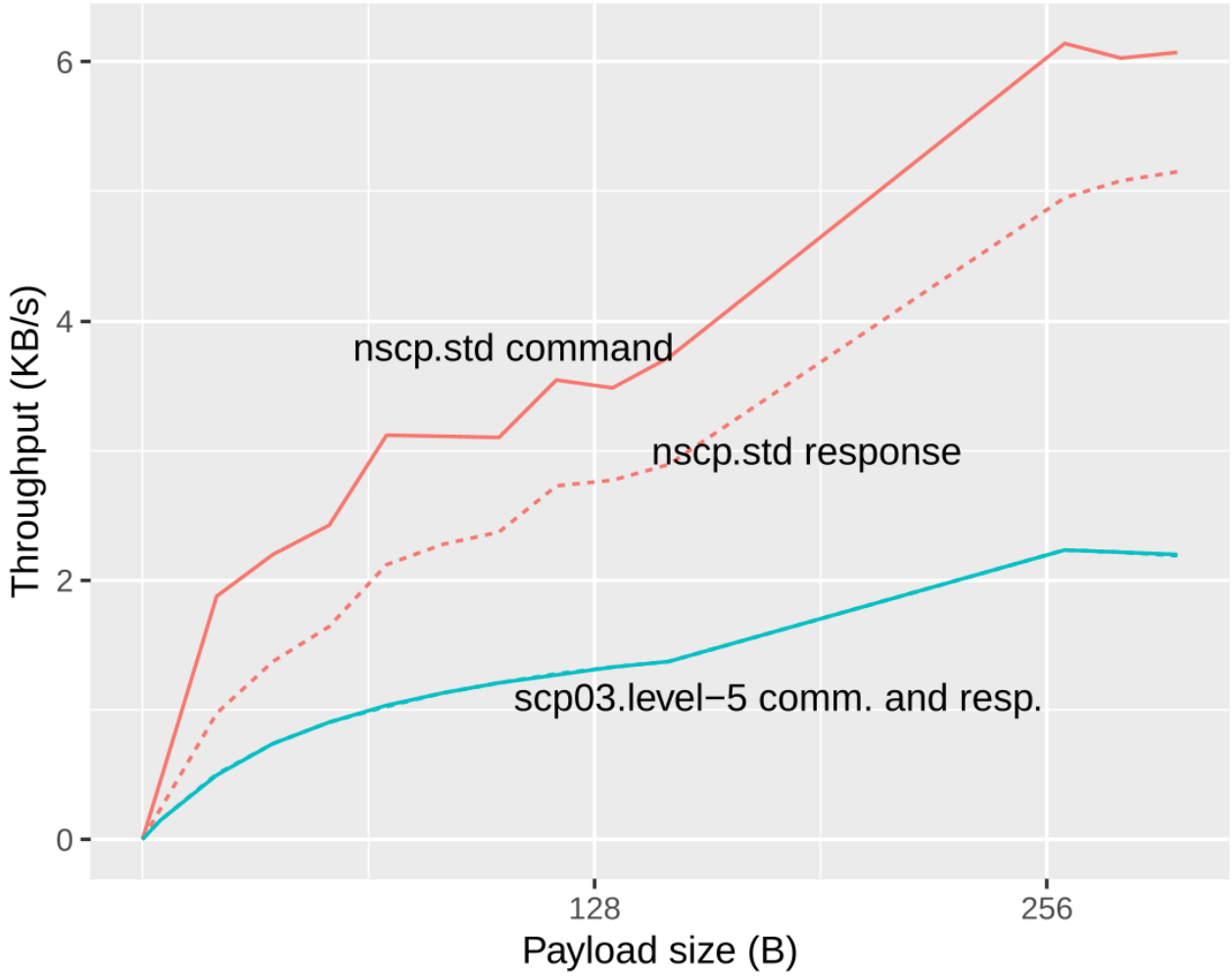


Figure 20: Throughput comparison of NSCP and SCP03 across various payload sizes (host: RPI, SE: RISC-V@10MHz, bus: I2C)

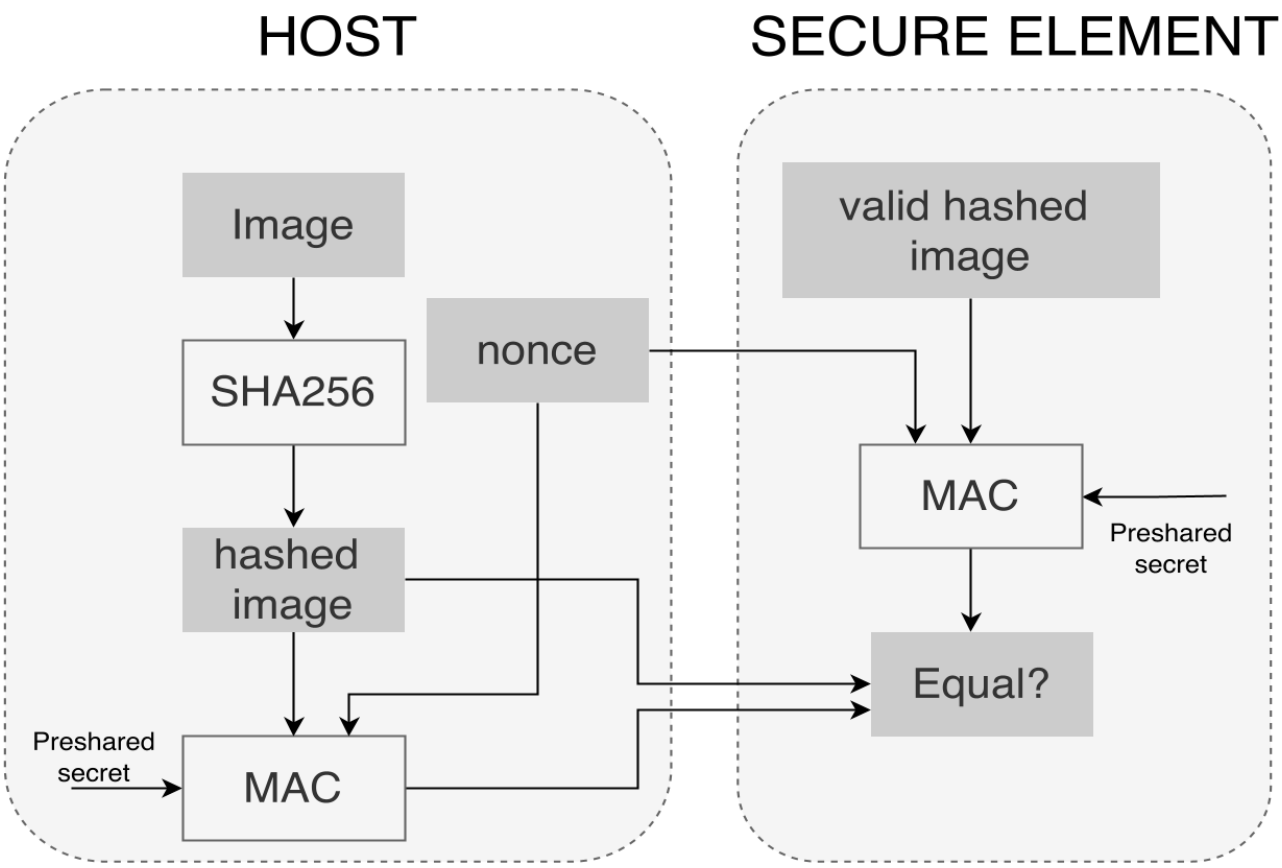


Figure 21: MCU secure boot implemented supported by a secure element

4.5 Implications for realistic scenarios

In this section, we will discuss some usage scenarios and the implications of NSCP. We will start first with a common scenario in IoT, that is, secure boot (SA19) where the MCU uses the secure element to attestate the integrity of the application image to be executed (see Figure 21). The MCU hashes (e.g., with SHA-256) the application image, and the resulting 32-byte SHA-256 digest is sent to the secure element together with a message authentication code (MAC) generated using the image digest, a pre-shared secret, and a 16-byte challenge nonce. The secure element independently reproduces the MAC (using the challenge nonce and a valid image hash) and compares it with the MAC received for attestation. In an illustrative example using a ARM Cortex M4 (Figure 19) assuming that 80 bytes of data (32-byte digest, 16-byte nonce, and 32-byte MAC) need to be transferred, an SCP I2C channel channel would transmit data in 108.1 ms, while NSCP I2C would transmit data within 26.9 ms with a potential 4x improvement on secure boot related communications.

Another scenario involves sending sensor data from the MCU to the cloud. Here, the secure element would store the cloud service's public key[7] to encrypt the data coming from the MCU, which is attached to the sensor. It would also handle decrypting the cloud service's responses. From our experiments on ARM Cortex M4 as a secure element, by batching data up to 224 single-precision floating point values, a throughput of 3.7kB/s could be reached using NSCP, in contrast to just

[7] Or the shared secret produced by a TLS handshake.

0.9kB/s with SCP03 at level 5. It is important to note that NSCP remains faster even if one opts not to encrypt data over I2C (for example, by using SCP03 level-3, see Figure 22). [8]

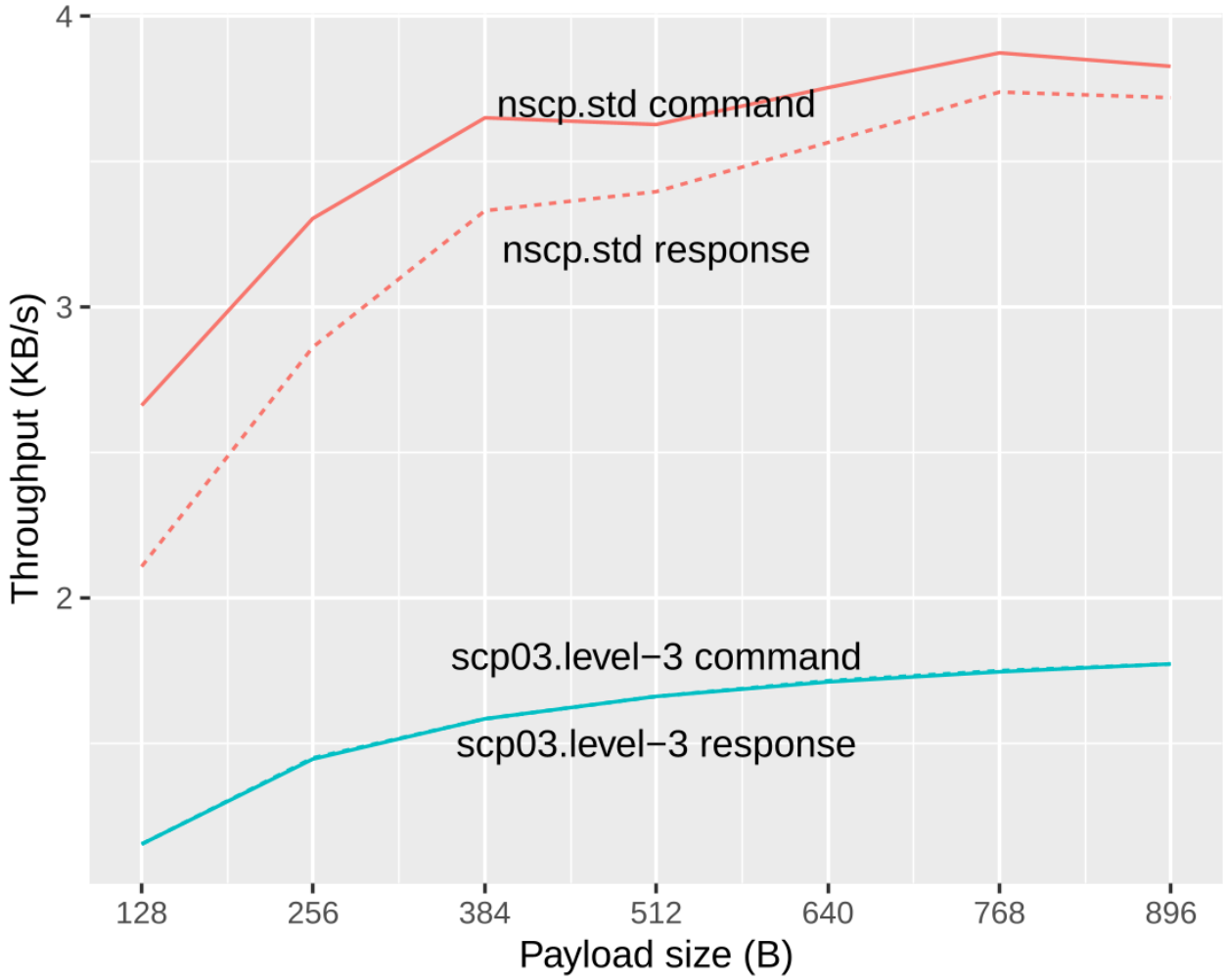


Figure 22: Throughput comparison of NSCP and SCP03 level-3 across various payload sizes (host: RPI, SE: ARM Cortex M4, bus: I2C)

Our protocol could also be applied to Trusted Platform Modules (TPMs) in PCs. TPMs enable CPU bus data encryption, but TPM2.0 encryption is malleable to attacks and requires additional mitigations ("Information technology — Trusted Platform Module Library — Part 1: Architecture" 2015). A simpler approach, as proposed here, might be beneficial.

4.5.1 Comparison with other Secure Channel Protocols

We provide in Table 4 a comparison on the key features of the different Secure Channel Protocols detailed in this work.

[8] NSCP's performance isn't weighed against that of SCP03 level-1 because SCP03 level-1 doesn't impose any confidentiality or integrity on the response messages.

Table 4: Comparison of Secure Channel Protocols

Protocol	typical implementation (HW/SW)	Body	Offers Open source implementation for both client and server	Reference or alternative implementation
SCP-03	HW	Global Platform	No	https://github.com/martinpaljak/GlobalPlatformPro/
Optiga shielded connection	HW	Infineon	No	https://github.com/Infineon/mtb-example-optiga-crypto/tree/6f712e2de35d4aa7203d963bf6cd9401a1ca0223
ATECC secure communication	HW	Microchip	No	https://github.com/MicrochipTech/cryptoauthlib
U-Blox Chip2Chip	HW	UBlox	No	https://github.com/u-blox/ubxlib
NaCl	SW	https://nacl.cr.yp.to/	Yes	NaCl website
Noise	SW	Trevor Perrin	Yes	Noise
Blinker	SW	*Markku-Juhani O. Saarinen*	No	
Strobe	SW	Mike Hamburg	Yes	*Strobe paper*
JWT	SW	IETF	Yes	https://jwt.io/libraries
Xoodyak	HW	Keccak Team	Yes	https://github.com/KeccakTeam/Xoodoo/blob/master/Reference/C%2B%2B/Sources/Xoodyak.cpp
NSCP	HW	ORSHIN Project	Yes	ORSHIN open source repository

4.6 Considerations on security

Regarding confidentiality and integrity, it is easy to see that our protocol follows Xoodyak's recommended Authenticated Encryption strategy coupled with a cryptographic Ratchet (Xoodyak).

More in detail, it enjoys the security strength levels expressed in Corollary 2 of (Xoodyak), i.e., the confidentiality and integrity of plain text, as well as the integrity of associated data correspond to 128 bits of computational complexity strength and 160 bits of data complexity strength. These values can be derived from the length $\kappa = 128$ bits of NSCP's static keys as well as the dimension $t = 128$ bits of NSCP's authentication tags.

However, when dealing with a secure protocol, an additional security property that must be considered is forward secrecy. Forward secrecy ensures that the confidentiality of data exchanged in previous sessions remains intact even if long-term secrets, such as the private key K, are compromised. In our scenario, forward secrecy is achievable through Diffie-Hellman ephemeral (DHE) key exchange (DH). Before starting the session i, both the host and the secure element can create a local ephemeral public/private key pair, swap public keys and establish a shared secret A_i following the DHE protocol. During the initialization of the Cyclist object, $K \oplus A_i$ can be fed to the constructor instead of only K. Thus, a compromise of K's secrecy would not allow decryption of a previous session due to the reliance of the Xoodyak state on these ephemeral secrets[9].

4.7 Conclusions and future work

This chapter presented the design and implementation of a new secure channel protocol for connecting microcontrollers to SEs. Starting from SCP03, the de facto standard in connecting SEs to MCUs in the IoT domain, we addressed the question of whether it is possible to design an even more lightweight secure channel protocol using sponge primitives. The new protocol operates under simpler principles than SCP03, as it requires that the entities share only one static key, has a reduced handshake phase integrated in the exchange of usual application data, mandates only one security level, and does not require the usage of a MAC chaining value mechanism. Experimental results show that the new protocol is faster than SCP03 even considering resource-constrained requirements while being easier to understand and implement. In future work, it would be beneficial to implement and test the protocol on a larger set of platforms, especially ones that employ processors that are different from the STM32 board used in this chapter. Future studies may also explore its suitability in SEs that use protocols that are not APDU-based to exchange application data with their hosts (e.g., RPMB devices such as SD cards) or to facilitate a secure element-supported DTLS communication.

[9] Note that forward secrecy is also provided by SCP11 (the secure channel protocol for eSIMs). However, unlike NSCP, SCP11 is based on public key cryptography (Bettale, Dottax, and Grémy 2024).

Chapter 5 Summary and Conclusion

This work presents part of the outcomes of the ORSHIN open-source resilient hardware and software solutions for Internet of Things (IoT) security project, and in particular the successful results of Task 5.3 “Essential s&p guarantees for intra-device communication” and Task 5.4 “Beyond essential s&p guarantees for intra-device communication”.

In the second chapter of this document, we present the research done within the ORSHIN project about the state of the art in secure communication protocols within a device. We started by describing the different phases of a secure communication protocol (Provisioning, Handshake, and Data exchange) then detailed them for each protocol under examination.

Our analysis began with an examination of the widely used Secure Channel Protocol (SCP) family, in particular the SCP-03 protocol used for securing the communication on the i2c channel with a secure element. Then we presented the Replay Protected Memory Block (RPMB) protocol. Subsequently, the proprietary protocols employed in Infineon’s Optiga products, Microchip's ATECC, and Ublox products, were scrutinised for their characteristics, performances, and security properties.

The exploration further extended to the protocol frameworks, where we delved into details about different scientific publications, which provide foundational structures and building blocks for creating “customised” secure channel protocols. These frameworks, known for prioritising usability and abstraction, are widely adopted in many different commercial products.

Finally, we introduced emerging constructions such as that of JWT and Deck functions, underscoring evolving methodologies in building and deploying security. The exploration undertaken here sets the stage for the subsequent chapters.

In the third chapter we present the ORSHIN contribution of a new vulnerability discovered in an open-source protocol for the Internet of Things security. During the study of JWTs, in particular, we found a weakness in the protocol that allows an attacker to gain access to sensitive resources months after the attack was perpetrated.

Chapter four details the ORSHIN project's contribution to developing a secure protocol for IoT intra-device communication. We specify and implement our newly developed protocol, called NSCP, which employs lightweight primitives to supersede the existing SCP03 protocol for secure communication. This result was published at DATE25 conference [Date25].

The NSPC protocol, as well as the classical SCP03, are implemented in the ORSHIN demonstrators reported in D5.3.

Chapter 6 List of Abbreviations

Abbreviation	Translation
AES	Advanced Encryption Standard
AEAD	Authenticated Encryption with Associated Data
AKE	Authenticated Key Exchange
API	Application Programming Interface
AT	("AT" meaning 'attention') a set of instructions to control a modem
C2C	Chip to Chip
CA	Certificate Authority
CBC	Cipher Block Chaining
C-MAC	Command MAC
DES	Data Encryption Standard
DH	Diffie-Hellman
DNS	Domain Name System
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
EdDSA	Edwards-curve Digital Signature Algorithm
EEPROM	Electrically Erasable Programmable Read-Only Memory
eMMC	embedded MultiMediaCard
ES	Ephemeral Static
FPGA	Field Programmable Gate Array
GCM	Galois Counter Mode
HMAC	Hash-Based Message Authentication Code
HTTP	Hypertext Transfer Protocol
I2C	Inter Integrated Circuit
IV	Initialization Vector
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
JWA	JSON Web Algorithm
JWE	JSON Web Algorithm

Abbreviation	Translation
JWS	JSON Web Signature
JWT	JSON Web Token
KDF	Key Derivation Function
MAC	Message Authentication Code
MCU	Microcontroller Unit
mEAC	Modular Extended Access Control
MSEQ	Master Sequence Number
NaCl	Networking and Cryptography Library
NVM	Non-Volatile Memory
NVMe	Non-Volatile Memory express
OAEP	Optimal Asymmetric Encryption Padding
OEM	Original Equipment Manufacturer
OTP	One-Time Programmable
PACE	Password Authentication Connection Establishment
PreSSec	Pre-Shared Secret
PRF	Pseudo Random Function
PRNG	Pseudo Random Number Generator
PKI	Public Key Infrastructure
RFC	Request For Comments
RISC-V	Reduced Instruction Set computer (RISC) Five
RoT	Root of Trust
RPMB	Replay Protected Memory Block
RSA	Rivest–Shamir–Adleman
RSASSA	RSA Signature Scheme with Appendix - Probabilistic Signature Scheme
SCP	Secure Channel Protocol
SHA	Secure Hash Algorithm
SSEQ	Slave Sequence Number
TLS	Transport Layer Security
TLV	Tag, Length, and Value
UFS	Universal Flash Storage
XOF	Extendable Output Function
XOR	Exclusive-OR

Chapter 7 Bibliography

AES

J. Daemen, V. Rijmen, "AES Proposal: Rijndael", 2003.

Available http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf#page=1

AES-CBC

NIST, “Recommendation for block cipher modes of operation: Methods and techniques,” 2001.

Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

AES-CMAC

NIST, “Recommendation for block cipher modes of operation: The CMAC mode for authentication,” 2005.

Available: https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-38b.pdf

AES-GCM Cipher Suites for TLS

Joseph Salowey, “RFC 5288: AES Galois Counter Mode (GCM) Cipher Suites for TLS”, 2008.

Available: https://www.rfc-editor.org/rfc/rfc5288

AN2589

Microchip Technology, ‘AN2589: Differences Between the ATECC608A and ATECC508A CryptoAuthentication™ Devices’, 2018

Available: https://ww1.microchip.com/downloads/en/Appnotes/00002589A.pdf

Ard19

Arduino cloud provider examples, 2019.

Available: https://github.com/arduino/ArduinoCloudProviderExamples

Ard20

Arduino S.r.l. Arduino security primer, 2020.

Available: https://blog.arduino.cc/2020/07/02/arduino-security-primer/

ARD24

Arduino S.r.l. Securely Connecting a MKR GSM 1400 to Google Cloud IoT Core, 2024

Available: https://docs.arduino.cc/tutorials/mkr-gsm-1400/securely-connecting-a-mkr-gsm-1400-to-google-cloud-iot-core/

ARM

ARM PSA.

Available: https://datatracker.ietf.org/doc/html/draft-tschofenig-rats-psa-token

ATECC508A

Microchip, 'ATECC508A, ATECC508A CryptoAuthentication Device Complete Data Sheet', 2017.

Available:

https://cdn.sparkfun.com/assets/learn_tutorials/1/0/0/3/Microchip_ATECC508A_Datasheet.pdf

ATECC608A

Microchip Technology, 'ATECC608A, CryptoAuthentication™ Device Summary Datasheet', 2018

Available:
https://ww1.microchip.com/downloads/aemDocuments/documents/SCBU/ProductDocuments/DataSheets/ATECC608A-CryptoAuthentication-Device-Summary-Data-Sheet-DS40001977B.pdf

ATECC608A-Laser

Ledger Donjon, 'Defeating a Secure Element with Multiple Laser Fault Injections', 2021

Available: https://www.sstic.org/media/SSTIC2021/SSTIC-actes/defeating_a_secure_element_with_multiple_laser_fau/SSTIC2021-Article-defeating_a_secure_element_with_multiple_laser_fault_injections-heriveaux.pdf

ATECC608A-TFLXTLS

Microchip Technology, 'ATECC608A-TFLXTLS, ATECC608A-TFLXTLS CryptoAuthentication™ Data Sheet', 2019.

Available: https://ww1.microchip.com/downloads/en/DeviceDoc/ATECC608A-TFLXTLS-CryptoAuthentication-Data-Sheet-DS40002138A.pdf

ATECC608B

Microchip Technology, 'ATECC608B, CryptoAuthentication™ Device Summary Data Sheet', 2018

Available: https://ww1.microchip.com/downloads/en/DeviceDoc/ATECC608B-CryptoAuthentication-Device-Summary-Data-Sheet-DS40002239A.pdf

BS16

Victoria Beltran and Antonio F. Skarmeta, "An overview on delegated authorization for coap: Authentication and authorization for constrained environments (ace)".

In 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT), pages 706–710, 2016.

Bla05

John Black, Authenticated encryption, 2005.

BLAKE2

M.-J. Saarinen and J.-P. Aumasson, 'RFC 7693: The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC).', 2015.

Available: http://www.ietf.org/rfc/rfc7693.txt

Blinker

M. Saarinen, 'Beyond Modes: Building a Secure Record Protocol from a Cryptographic Sponge Permutation', 2013.

Available: https://eprint.iacr.org/2013/772.pdf

Card Specification

GlobalPlatform, “Card Specification, Version 2.3.1”, 2018.

Available: https://globalplatform.org/wp-content/uploads/2018/05/GPC_CardSpecification_v2.3.1_PublicRelease_CC.pdf

CEN/EN 419 212

CEN, “Application Interface for smart cards used as Secure Signature Creation Devices, Part 1 (Basic services) & Part 2 (Additional services)”, 2014.

ChaCha20

D. J. Bernstein, ‘ChaCha, a variant of Salsa20.’, 2008.

Available: https://cr.yp.to/papers.html#chacha.

CLMLD19

Campos, Fabio, Lars Jellema, Mauk Lemmen, Lars Müller, Daan Sprenkels, and Benoit Viguier. 2020. “Assembly or Optimized c for Lightweight Cryptography on RISC-v?” In Cryptology and Network Security: 19th International Conference, CANS 2020, Vienna, Austria, December 14–16, 2020, Proceedings 19, 526–45. Springer.

COSE

Keys, Algorithms, COSE and CWT in Go.

Available: https://github.com/ldclabs/

CryptoAuthentication™

Microchip, CryptoAuthentication™ family, https://www.microchip.com/en-us/products/security/security-ics/cryptoauthentication-family

Curve25519

Daniel J. Bernstein, ‘Curve25519: new Diffie-Hellman speed records’, 2006.

Available: http://cr.yp.to/papers.html#curve25519.

Curve25519 Montgomery ladder

Daniel J. Bernstein, ‘Curve25519: new Diffie-Hellman speed records’, 2006. Available: http://cr.yp.to/papers.html#curve25519.

Date25

J. Bushi, A. Battistello, G. Bertoni and V. Zaccaria, "Design, Implementation and Validation of NSCP: A New Secure Channel Protocol for Hardened IoT," *2025 Design, Automation & Test in Europe Conference (DATE)*, Lyon, France, 2025, pp. 1-7, doi: 10.23919/DATE64628.2025.10992943.

Dav01

Don Davis. Defective sign & encrypt in s/mime, pkcs# 7, moss, pem, pgp, and xml. In USENIX Annual Technical Conference, General Track, pages 65–78, 2001.

Available: https://www.usenix.org/legacy/publications/library/proceedings/usenix01/full_papers/davis/davis.pdf

DNSCurve

Daniel J. Bernstein, "DNSCurve: Usable security for DNS", 2009. http://dnscurve.org/

Deck functions

KECCAK Team, Refactoring symmetric cryptography with deck functions, 2022. https://keccak.team/2022/refactoring_with_deck_functions.html

DES

NIST, "FIPS-46: Data Encryption Standard (DES).", 1979.

Available: http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf

DH

Diffie, W., and M. Hellman. 1976. "New Directions in Cryptography." IEEE Transactions on Information Theory 22 (6): 644–54.

Available: https://doi.org/10.1109/TIT.1976.1055638

EdDSA

D. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang, 'High-speed high-security signatures', 2011.

Available: https://eprint.iacr.org/2011/368

EMQX

Migrate Your Business from GCP IoT Core 03 | Use JSON Web Token (JWT) to Verify Device Credentials.

Available: https://www.emqx.com/en/blog/migrate-your-business-from-gcp-iot-core-03

eSTREAM

ECRYPT, The eSTREAM project, http://www.ecrypt.eu.org/stream/.

ETSI TS 102 225

ETSI, "TS 102 225: Smart Cards; Secured packet structure for UICC based applications", 2014

Available: https://www.etsi.org/deliver/etsi_ts/102200_102299/102225/12.00.00_60/ts_102225v120000p.pdf

ETSI TS 102 226

ETSI, “TS 102 226: Smart Cards; Remote APDU structure for UICC based applications”, 2015
Available: https://www.etsi.org/deliver/etsi_ts/102200_102299/102226/12.00.00_60/ts_102226v120000p.pdf

Farfalle

G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, “Farfalle: parallel permutation-based cryptography”, 2017.
Available: https://eprint.iacr.org/2016/1188.pdf

FIPS 180-4

NIST, “FIPS 180-4. Secure Hash Standard (SHS),” 2012.
Available: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

GlobalPlatform

GlobalPlatform website, https://globalplatform.org/

GooBP

Google Inc. Best practices for running an iot backend on google cloud.
Available: https://cloud.google.com/architecture/connected-devices/bps-running-iot-backend-securely

Goo20

Google Cloud Platform IoT arduino examples, 2020.
Available: https://github.com/GoogleCloudPlatform/google-cloud-iot-arduino

Goo23

gcp-iot-core-examples, 2023.
Available: https://cloud.google.com/iot/docs/how-tos/credentials/jwts

Goo18

Securing cloud-connected devices with cloud iot and microchip, 2018
Available: https://cloud.google.com/blog/products/gcp/securing-cloud-connected-devices-with-cloud-iot-and-microchip

HiveMQ

HiveMQ GmbH. Step Up Your MQTT Security with JWT Authentication on HiveMQ Cloud Starter. Posted: 2024-03-18.
Available: https://www.hivemq.com/blog/step-up-mqtt-security-jwt-authentication/

HMAC-SHA

Eastlake, D. and T. Hansen, "US Secure Hash Algorithms (SHA and HMAC-SHA)", RFC 4634, 2006.
Available: https://www.rfc-editor.org/rfc/rfc4634.html

IANA

IANA JSON Web Token registered Claims.

Available: https://www.iana.org/assignments/jwt/jwt.xhtml

ID Token

OpenID Connect, ID Token, https://openid.net/specs/openid-connect-core-1_0.html#CodeIDToken

IFX I2C

Infineon, 'IFX I2C Protocol, Protocol Specification', 2020.

Available: https://github.com/Infineon/optiga-trust-m/blob/develop/documents/Infineon_I2C_Protocol_v2.03.pdf

ISO 7816

ISO/IEC 7816-8:2021(en): Identification cards — Integrated circuit cards — Part 8: Commands and mechanisms for security operations, 2021.

Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:7816:-8:ed-5:v1:en

I2P

NTCP 2, I2P, https://geti2p.net/spec/ntcp2

JBS15a

M. Jones, J. Bradley, and N. Sakimura. JSON Web Signature (JWS). RFC 7515, RFC Editor, 5 2015.

JBS15b

M. Jones, J. Bradley, and N. Sakimura. JSON Web Token (JWT). RFC 7519, RFC Editor, 5 2015.

JEDEC

JEDEC Organisation website, https://www.jedec.org/about-jedec.

JH15

M. Jones and J. Hildebrand. JSON Web Encryption (JWE). RFC 7516, RFC Editor, 5 2015.

Jon15a

M. Jones. JSON Web Algorithms (JWA). RFC 7518, RFC Editor, 5 2015.

Jon15b

M. Jones. JSON Web Key (JWK). RFC 7517, RFC Editor, 5 2015.

JWA

Michael B. Jones, "JSON Web Algorithms (JWA)", 2015.

Available: https://www.rfc-editor.org/rfc/rfc7518

JWT

Michael B. Jones, “JSON Web Token (JWT)”, 2015.
Available: https://www.rfc-editor.org/rfc/rfc7519.

JWT with PKI

My way with Java blog, JWT Signing And Validation With A PKI Keypair, 2020, https://jarirajari.wordpress.com/2020/08/14/jwt-signing-and-validation-with-a-pki-keypair/

JWS

Michael B. Jones, “JSON Web Signature (JWS)”, 2015.
Available: https://www.rfc-editor.org/rfc/rfc7515

JWE

Michael B. Jones, “JSON Web Encryption (JWS)”, 2015.
Available: https://www.rfc-editor.org/rfc/rfc7516

JWET18

Michael B. Jones, Erik Wahlstroem, Samuel Erdtman, and Hannes Tschofenig.
CBOR Web Token (CWT). RFC 8392, May 2018.

JWA

Michael B. Jones, “JSON Web Algorithms (JWS)”, 2015.
Available: https://www.rfc-editor.org/rfc/rfc7518

KECCAK

G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, ‘The KECCAK reference’, 2011.
Available: https://keccak.team/files/Keccak-reference-3.0.pdf

Keytool

Keytool, Oracol Java SE Documentation, https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html

Lighting Network

Lighting Network, https://github.com/lightning/bolts/blob/master/08-transport.md

LMOW24

Laurence Lundblade, Giridhar Mandyam, Jeremy O’Donoghue, and Carl Wallace

The Entity Attestation Token (EAT). Internet-Draft draft-ietf-rats-eat-26, Internet Engineering Task Force, May 2024. Work in Progress.

LV18

Tim Hollebeek Loganaden Velvindron.

Authentication and Authorization for Constrained Environments. Internet-Draft draft-ietf-ace-about, Internet Engineering Task Force, 2018. Work in Progress.

Mic17

Microchip. Atecc508a, 2017.

Available: https://www.microchip.com/en-us/product/ATECC508A

Mic18a

Microchip. Atecc608a, 2018.

Available: https://www.microchip.com/en-us/product/ATECC608A

Mic18b

MicrochipTech. gcp-iot-core-examples, 2018.

Available: https://github.com/MicrochipTech/gcp-iot-core-examples

Montgomery ladder

Peter L. Montgomery, 'Speeding the Pollard and elliptic curve methods of factorization', 1987. Available: http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/

Mot84

Motorola. Single-Chip Microcomputer Data, 1984.

Available:
https://archive.org/details/bitsavers_motoroladaSingleChipMicrocomputerData_68061538

Murdoch07

Murdoch, Seven J. 2007. "EMV Flaws and Fixes: Vulnerabilities in Smart Card Payment Systems." In COSIC Seminar, June. Vol. 11.

Sta19

OASIS Standard. MQTT version 5.0. Retrieved June, 22:2020, 2019.

NaCl website

NaCl: Networking and Cryptography library website, https://nacl.cr.yp.to/

NaCl paper

D. Bernstein1 , T. Lange , and P. Schwabe, 'The security impact of a new cryptographic library', 2012.

Available: https://cr.yp.to/highspeed/coolnacl-20120725.pdf

NIST SP 800-38B

NIST, “NIST Special Publication 800-38B: Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication”, 2016.
Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38b.pdf

NIST SP 800-38C

NIST, ‘NIST Special Publication 800-38C: Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality’,
Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38c.pdf

NIST SP 800-56A

NIST, “Special Publication 800-56A: Recommendation for Pair-Wise KeyEstablishment Schemes Using Discrete Logarithm Cryptography, Revision 3”, 2018.
Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar3.pdf

Nebula

Nebula project, https://github.com/slackhq/nebula

Noise

T. Perrin, ‘The Noise protocol framework’, 2018.
Available: https://github.com/noiseprotocol/noise_spec/blob/master/output/noise.pdf

NVMe

NVMe, Non-Volatile Memory express, https://nvmexpress.org/specifications/

NXP21

NXP. I2C-bus specification and user manual, 1 October 2021.
Available: https://www.nxp.com/docs/en/user-guide/UM10204.pdf

NXPA71H

NXP. A71ch for secure connection to google cloud iot core.
Available: https://www.nxp.com/docs/en/application-note/AN12199.pdf

NXPNano

NXP Nano package. 2023
Available: https://github.com/NXPPlugNTrust/nano-package/tree/master

Okta

Inc. Okta. JWT.io
Available: https://jwt.io

On Dec(k) Functions

G. Van Assche, “On Dec(k) Functions”, 2018.

Available: https://keccak.team/files/OnDecAndDeckFunctions-Indocrypt2018.pdf

Opacity Secure Channel

GlobalPlatform, “Opacity Secure Channel, Card Specification v2.3 – Amendment G”, 2016.

Available: https://globalplatform.org/wp-content/uploads/2016/11/GPC_2.3_G_OpacitySecureChannel_v1.0.pdf

Optiga Trust M

OPTIGA, “OPTIGA™ Trust M Solution Reference Manual v3.50”, 2022.

Available: https://github.com/Infineon/optiga-trust-m/blob/develop/documents/OPTIGA%E2%84%A2%20Trust%20M%20Solution%20Reference%20Manual.md

Optiga shielded connection

Infineon, “OPTIGA™ Trust M: Shielded connection – KBA235350”, 2022, https://community.infineon.com/t5/Knowledge-Base-Articles/OPTIGA-Trust-M-Shielded-connection-KBA235350/ta-p/354380

PBSHCSL23

Peng, Shuohang, Bohan Yang, Shuying Yin, Hang Zhao, Cankun Zhao, Shaojun Wei, and Leibo Liu. 2023. “A Low-Randomness First-Order Masked Xoodyak.” In 2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 48–56.

Available: https://doi.org/10.1109/HOST55118.2023.10133290

Poly1305

Daniel J. Bernstein, ‘The Poly1305-AES message-authentication code’, 2005.

Available: http://cr.yp.to/papers.html#poly1305.

Poly1305-AES

Daniel J. Bernstein, ‘The Poly1305-AES Message-Authentication Code’, 2005.

Available: https://link.springer.com/chapter/10.1007/11502760_3

Remote Application Management over HTTP

GlobalPlatform, “Remote Application Management over HTTP, Card Specification v2.2 – Amendment B, Version 1.1.3”, 2015.

Available: https://globalplatform.org/wp-content/uploads/2018/06/GPC_2.2_B_Remote_Application_Mgmt_over_HTTP_v1.1.3.pdf

Res18

E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC8446, RFC Editor, 8 2018.

RFC 3447

J. Jonnson, 'RFC 3447: Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography, Specifications Version 2.1', 2003.
Available: https://www.rfc-editor.org/rfc/rfc3447

RFC 5288
J. Salowey, A. Choudhury, D. McGrew, 'AES Galois Counter Mode (GCM) Cipher Suites for TLS', 2008.
Available: https://www.rfc-editor.org/rfc/rfc5288

RFC 5652
R. Husley, "RFC 5652: Cryptographic Message Syntax (CSM)", 2009.
Available: https://www.rfc-editor.org/rfc/rfc5652

RFC 7539
Y. Nir and A. Langley, 'RFC 7539: ChaCha20 and Poly1305 for IETF Protocols.', 2015.
Available: http://www.ietf.org/rfc/rfc7539.txt

RFC 9396
T. Lodderstedt, J. Richer, B. Campbell, 'RFC 9396: OAuth 2.0 Rich Authorization Requests', 2023.
Available: https://www.rfc-editor.org/rfc/rfc9396.html

RPMB
Western Digital, 'A detailed overview of the different security methods one can use in an e.MMC storage device', 2017.
Available: https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/white-paper/white-paper-emmc-security.pdf

RPMB blog
Sergio Prado, 'RPMB, a secret place inside the eMMC', 2023.
Available: https://sergioprado.blog/rpmb-a-secret-place-inside-the-emmc/

SA19
Schläpfer, Tobias, and Andreas Rüst. 2019. "Security on IoT Devices with Secure Elements."
Available: https://api.semanticscholar.org/CorpusID:108326638

Salsa20
Daniel J. Bernstein, 'The Salsa20 family of stream ciphers', 2008.
Available: https://cr.yp.to/snuffle/salsafamily-20071225.pdf

SARA-05 "00B"
Samuele Falcomer, "SARA-R5 "00B" product version initial production", 2020.
Available: https://content.u-blox.com/sites/default/files/SARA-R5-IP_IN_%28UBX-20037360%29.pdf

Schnorr

C. Schnorr, 'Efficient signature generation by smart cards', 1991.

Available: https://link.springer.com/article/10.1007/BF00196725

SCP CardLogic

CardLogic, Secure Channel Protocol (SCP),

https://www.cardlogix.com/glossary/secure-channel-protocol-scp01-scp02-scp03/

SCP Cryptanalysis

M. Sabt and J. Traore, "Cryptanalysis of GlobalPlatform Secure Channel Protocols," 2017.

SCP02 Attack

G. Avoine, L. Ferreira, "Attacking GlobalPlatform SCP02-compliant Smart Cards Using a Padding Oracle Attack", 2018.

Available: https://tches.iacr.org/index.php/TCHES/article/view/878/830

SCP03

GlobalPlatform, "Secure Channel Protocol 03, Version 2.2", 2009.

Available: https://globalplatform.org/wp-content/uploads/2019/03/GPC_2.2_D_SCP03_v1.0.pdf

SCP10 Flaws

D. De Almeida Braga, P. Fouque, M. Sabt, "The Long and Winding Path to Secure Implementation of GlobalPlatform SCP10", 2020.

Available: https://tches.iacr.org/index.php/TCHES/article/view/8588/8155

SCP11

GlobalPlatform, "Secure Channel Protocol '11', Card Specification v2.3 – Amendment F, Version 1.2", 2018.

Available: https://globalplatform.org/wp-content/uploads/2017/09/GPC_2_3_F_SCP11_v1.2_PublicRelease.pdf

Se

SAFE-eV ECMF.

Available: https://github.com/SAFE-eV/OCMF-Open-Charge-Metering-Format/blob/master/OCMF-en.md

SHA-3

NIST, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, 2015.

Available: https://csrc.nist.gov/pubs/fips/202/final

Shi19

Krishna Shingala. JSON web token (JWT) based client authentication in message queuing telemetry transport (MQTT). CoRR, abs/1903.02895, 2019.

SHJ20

Y. Sheffer, D. Hardt, and M. Jones. JSON Web Token Best Current Practices. RFC 8725, RFC Editor, 2 2020.

Sponge Duplex

G. Bertoni , J. Daemen , M. Peeters, G. Van Assche, 'Duplexing the sponge: single-pass authenticated encryption and other applications', 2011.

Available: https://keccak.team/files/SpongeDuplex.pdf

Sta19

OASIS Standard. Mqtt version 5.0. Retrieved June, 22:2020, 2019.

STROBE paper

M. Hamburg, 'The Strobe protocol framework', 2017.

Available: https://eprint.iacr.org/2017/003.pdf

STROBE website

STROBE protocol framework website, https://strobe.sourceforge.io

TLS 1.2

E. Rescorla, T. Dierks, 'The Transport Layer Security (TLS) Protocol, Version 1.2', RFC 5246, 2008

Available: https://datatracker.ietf.org/doc/html/rfc5246

Twisted Edward Curves

Twisted Edward Curves, https://en.wikipedia.org/wiki/Twisted_Edwards_curve

ubxlib

Available: https://github.com/u-blox/ubxlib

ubxlib C2C

ubxlib sample code of C2C implementation, https://github.com/u-blox/ubxlib/tree/v1.2.0/example/security/c2c

UFS

JEDEC, UFS (Universal Flash Storage), https://www.jedec.org/standards-documents/focus/flash/universal-flash-storage-ufs

Xoodoo

J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, “Xoodoo Cookbook”, 2018. Available: https://eprint.iacr.org/2018/767.pdf

Xoodyak

J. Daemen, Joan, Seth Hoffert, Michael Peeters, Gilles Assche, and Ronny Keer. 2020. “Xoodyak, a Lightweight Cryptographic Scheme.” IACR Transactions on Symmetric Cryptology, June, 60–87. Available: https://doi.org/10.46586/tosc.v2020.iS1.60-87

WireGuard

WireGuard VPN, https://www.wireguard.com/