Design, implementation and validation of NSCP: a New Secure Channel Protocol for hardened IoT*

Bushi, Joan
DEIB, Politecnico di Milano
20152 Milan, Italy

Battistello, Alberto
Security Pattern
25080 Mazzano (BS), Italy
a.battistello@securitypattern.com

Bertoni, Guido
Security Pattern
25080 Mazzano (BS), Italy
g.bertoni@securitypattern.com

Zaccaria, Vittorio
DEIB, Politecnico di Milano
20152 Milan, Italy
vittorio.zaccaria@polimi.it

Abstract

This paper deals with the design, implementation and validation of a new secure channel protocol to connect microcontrollers and secure elements. The new secure channel protocol (NSCP) relies on a lightweight cryptographic primitive (Xoodyak) and simplified operating principles to provide secure data exchange. The performance of the new protocol is compared with that of GlobalPlatform's Secure Channel Protocol 03 (SCP03), the current *de facto* standard for hardening the connection between a microcontroller and a secure element in industrial IoT. The evaluation was performed in two scenarios where the secure element was emulated with an ARM Cortex M4 and an OpenHW RISC-V MPU synthesized on an Artix FPGA. The results of the evaluation are an indicator of the potential advantage of the new protocol over SCP03: in the best case, the new protocol is able to apply cryptographic protection to messages from 3.64x to 4x with respect to SCP03 at its maximum security level. The speedup in the channel initiation process is also considerable, with a factor of up to 3.7. These findings demonstrate that it is possible to conceive a new protocol which offers adequate cryptographic protection, while being more lightweight than the present standard.

1 Introduction

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*. 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 paper, we propose an alternative to SCP03 consisting of a lightweight secure channel protocol that utilizes Xoodyak [1], 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 the paper 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

*This work was partially funded by the European Union under grant agreement no. 101070008 (ORSHIN project). 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.

on metrics such as throughput and memory occupation. Finally, we conclude with a discussion of the findings and potential future research directions.

2 Background

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

2.1 The secure channel protocol

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 [2]; SCP03 [3] [4] 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 [5] (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 [6].

During the SCP handshake (see Fig. 1), 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 ν_h (nonce). The secure element generates its own challenge ν_s and computes the session keys S_* using a CMAC-based key derivation function (KDF) [6]. 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 ν_s.

To validate χ_s, the host attempts to regenerate it using the pre-shared keys k_* and the exchanged challenges ν_*. 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.

To ensure confidentiality, SCP03 employs an "Encrypt-then-Authenticate" method (refer to Fig. 2) in which encryption is carried out using AES-CBC.[1]. 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 Fig. 2) guarantees the integrity of the command sequence produced by the host. In a

[1] 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 $\{m_i\}$, it produces recursively the encrypted representation y_i of m_i as

$$y_i = AES(m_i \oplus y_{i-1}, S), \quad y_0 = iv$$

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

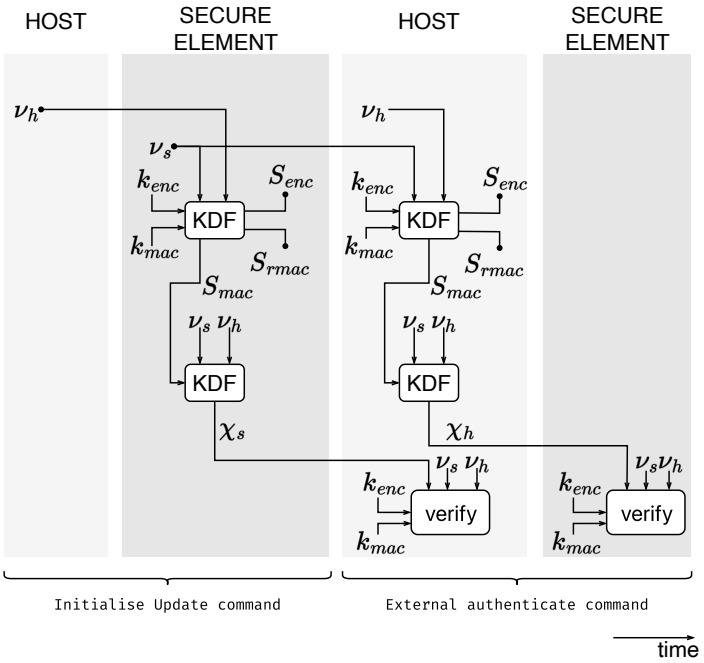

Figure 1: Handshake in SCP03. The aim is twofold: 1) to generate shared session keys S_* from exchanged nonces ν_* and 2) validate the presence of shared secrets between host and secure elements (i.e. the base keys k_*) through cryptograms χ_*.

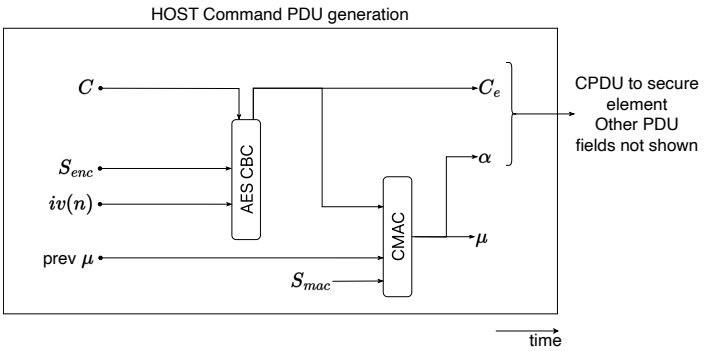

Figure 2: SCP03 encrypt then MAC applied to command PDUs sent from the host. On the left, C is the command to be sent, S_{enc} is the encryption key, $iv(n)$ is a session dependent initialization value, μ is the MAC chaining value (16 bytes, at the beginning prev μ is 0). On the right C_e is the encrypted 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).

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 [7, 8].

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 [1] and is the cornerstone of the protocol proposed in this paper. Xoodyak was selected due to its adaptability and outstanding performance, as also evidenced by an evaluation conducted on the NIST-LWC finalists using RISC-V architectures [9].

2.2 The Xoodyak primitive

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[2] 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.

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 [10]. However, both the MCU and secure element are assumed to have defenses against physical attacks, including side-channel attacks (e.g., masking [11]) and fault injection to protect pre-shared keys. Performance-wise, NSCP has been conceived to operate under principles simpler than SCP03. In

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

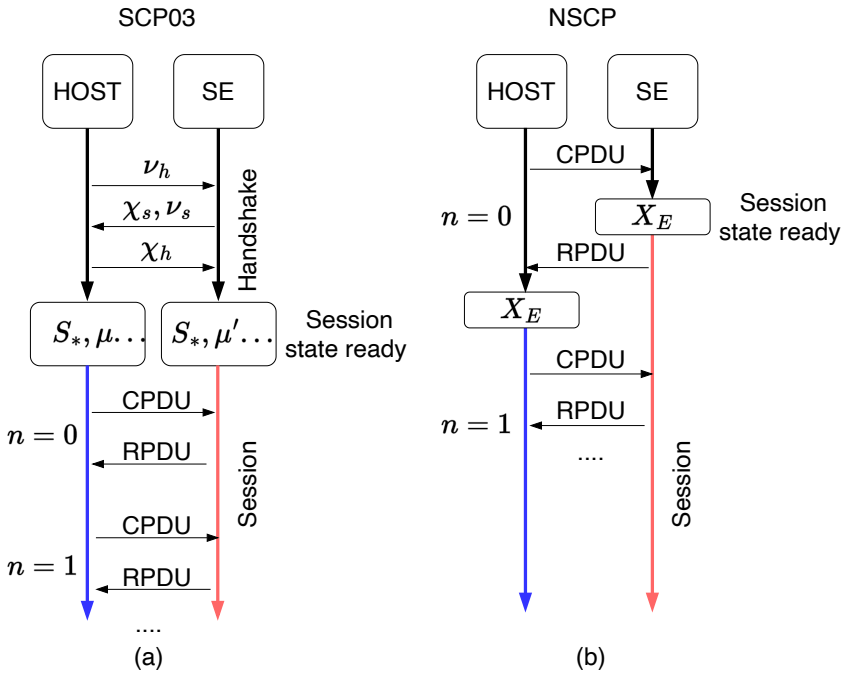

Figure 3: High-level comparison of session management in SCP03 and NSCP. SCP03 needs a handshake phase to mutually authenticate the host and the secure element. In NSCP, a valid initial session state X_E in both host and the secure element ensures they are mutually authenticated and the states in both of them evolve in a mirrored way.

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.

SCP03 and NSCP also differ in the way sessions are established (see Fig. 3). SCP03 uses a single handshake to exchange nonces and cryptograms for mutual authentication and session keys S_* (see Fig. 3(a)). NSCP, however, uses the Xoodyak object's state X as the session state[3], initializing its value during the first APDU exchange before the RPDU is created by the secure element (see Fig. 3(b)).

In a valid session, both host's and secure element's state evolve synchronously through Xoodyak's methods (see Fig. 4 and 5). 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 Figs. 4 and 5). 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 [1]), 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 Fig. 4), the entropy of the initial state X_A increases by *absorbing* a 128-bit nonce ν_h (as suggested in [1]) 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 [1]) 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 Fig. 5), 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 (Fig. 4, 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[4].

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

[4]Obviously, such a mirroring can be subject rare problems where 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

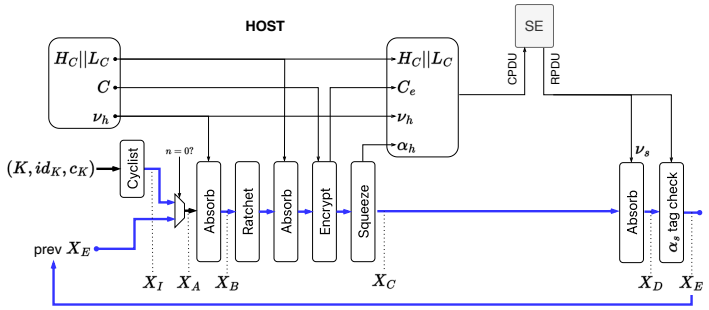

Figure 4: NSCP internal data on the host (when communicating with the secure element SE) are delineated as follows: C represents the plain text command, whereas C_e denotes the corresponding encrypted command. H_C is identified as the command header, and L_C specifies the command's length. ν_h (ν_s) 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.

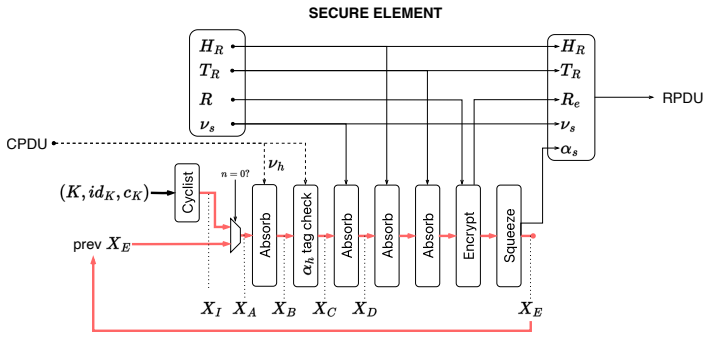

Figure 5: NSCP internal data on the secure element are delineated as follows: R denotes the plain text response, and R_e represents its encrypted counterpart. H_R refers to the response header, whereas T_R indicates the response trailer. Additionally, ν_s is the nonce generated by the secure element (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.

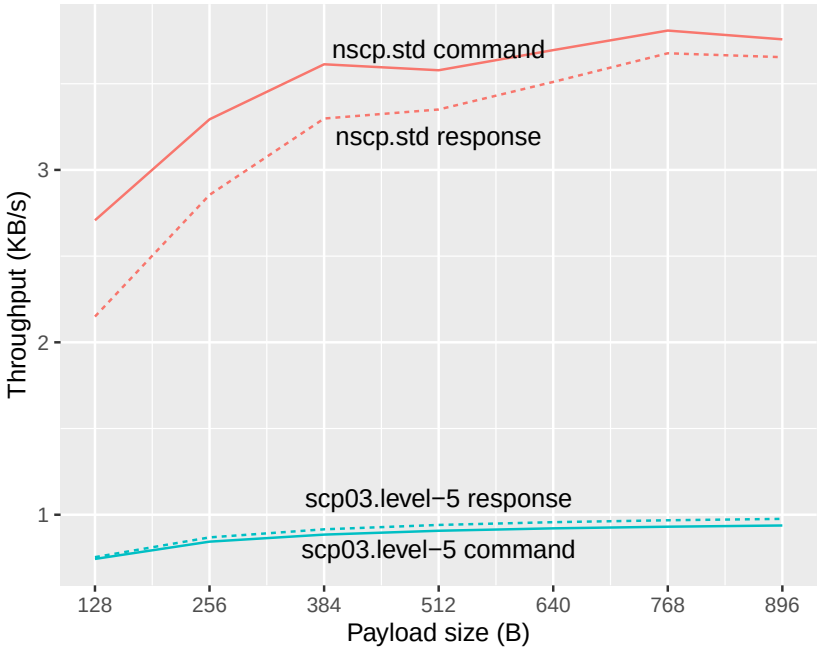

Figure 6: Throughput comparison of NSCP and SCP03 level-5 across various payload sizes (host: RPI, SE: ARM Cortex M4, bus: I2C)

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[5] , primarily due to the lighter cryptographic primitive and simpler design.

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 [12]. 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.

Fig. 6 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

synchronously.

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

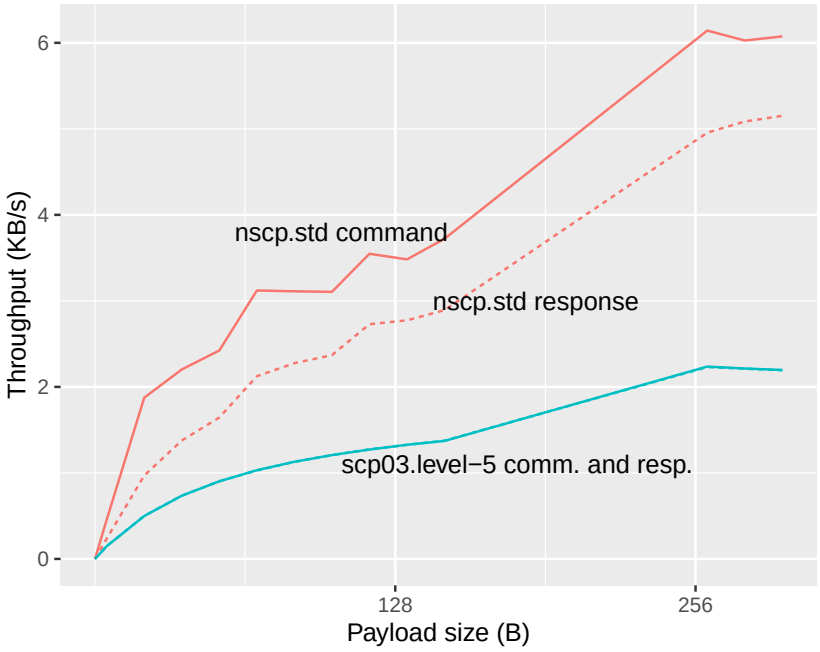

Figure 7: Throughput comparison of NSCP and SCP03 across various payload sizes (host: RPI, SE: RISC-V@10MHz, bus: I2C)

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 1 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.

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%

Table 1: Resource usage comparison between NSCP and SCP03. Positive delta values correspond to a reduction in resource consumption of NSCP with respect to SCP03

To corroborate our findings on a different architecture, we also benchmarked and 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 Fig. 7 shows, NSCP still shows 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.

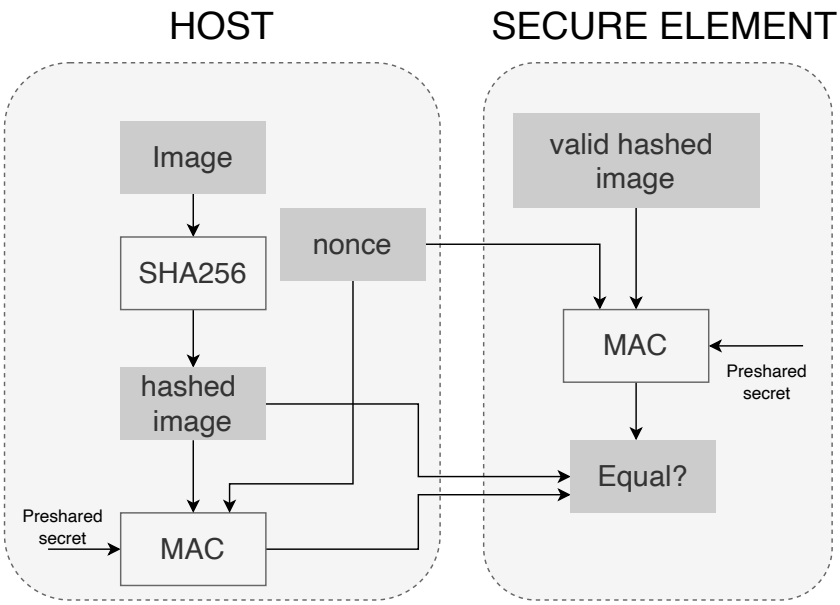

Figure 8: MCU secure boot implemented supported by a secure element

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* [13] where the MCU uses the secure element to attestate the integrity of the application image to be executed (see Fig. 8). 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 (Fig. 6) 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[6] 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 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 Fig. 9). [7]

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 [14]. A simpler approach, as proposed here, might be beneficial.

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 [1]. More in detail, it enjoys the security strength levels expressed in Corollary 2 of [1], 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

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

[7] 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.

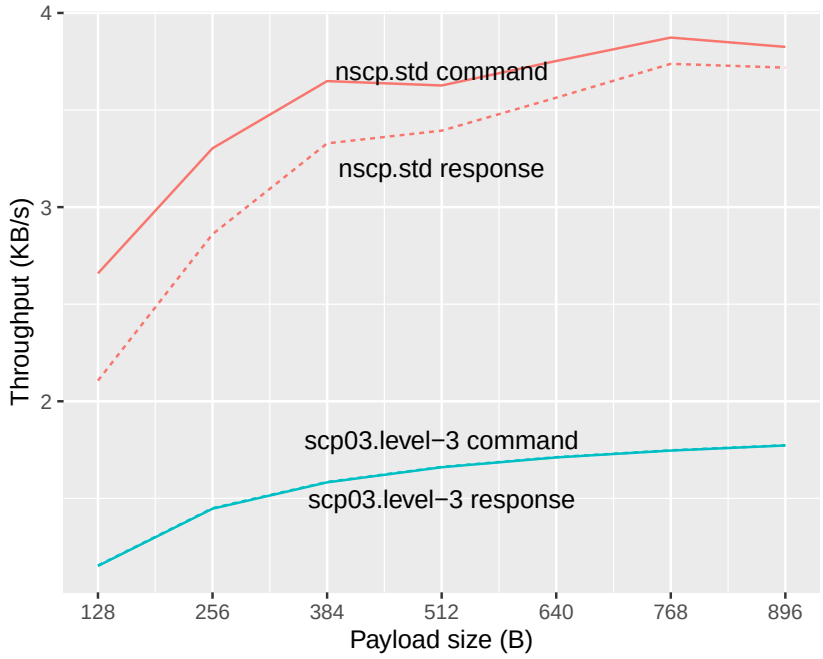

Figure 9: Throughput comparison of NSCP and SCP03 level-3 across various payload sizes (host: RPI, SE: ARM Cortex M4, bus: I2C)

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 [15]. 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[8].

7 Conclusions and future work

This paper presented the design and implementation of a new secure channel protocol for connecting microcontrollers to secure elements. Starting from SCP03, the *de facto* standard in connecting secure elements 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 paper. Future studies may also explore its suitability in secure elements 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.

References

[1] J. Daemen, S. Hoffert, M. Peeters, G. Assche, and R. Keer, "Xoodyak, a lightweight cryptographic scheme," *IACR Transactions on Symmetric Cryptology*, pp. 60–87, 06 2020.

[8]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 [16].

[2] GlobalPlatform. (2024) About GlobalPlatform. [Online]. Available: https://www.globalplatform.org/about-globalplatform/

[3] ——, "Secure Channel Protocol '03' – Public Release v1.1.2," 2019. [Online]. Available: https://globalplatform.org/wp-content/uploads/2014/07/GPC_2.3_D_SCP03_v1.1.2_PublicRelease.pdf

[4] ——, "Card specification – public release v2.3.1," 2019. [Online]. Available: https://globalplatform.org/wp-content/uploads/2018/05/GPC_CardSpecification_v2.3.1_PublicRelease_CC.pdf

[5] M. D. (NIST), "Recommendation for block cipher modes of operation: Methods and techniques," 2001. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

[6] ——, "Recommendation for block cipher modes of operation: The cmac mode for authentication," 2005.

[7] CardLogix. (2023) SCP03. [Online]. Available: https://www.cardlogix.com/glossary/scp03-secure-channel-protocol-3/

[8] M. Sabt and J. Traoré, "Cryptanalysis of GlobalPlatform Secure Channel Protocols," 2017.

[9] F. Campos, L. Jellema, M. Lemmen, L. Müller, D. Sprenkels, and B. Viguier, "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*. Springer, 2020, pp. 526–545.

[10] S. J. Murdoch, "Emv flaws and fixes: vulnerabilities in smart card payment systems," in *COSIC seminar, June*, vol. 11, 2007.

[11] S. Peng, B. Yang, S. Yin, H. Zhao, C. Zhao, S. Wei, and L. Liu, "A low-randomness first-order masked xoodyak," in *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2023, pp. 48–56.

[12] (2023). [Online]. Available: https://github.com/NXPPlugNTrust/nano-package/tree/master

[13] T. Schläpfer and A. Rüst, "Security on iot devices with secure elements," 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:108326638

[14] "Information technology — Trusted Platform Module Library — Part 1: Architecture," International Organization for Standardization, Geneva, CH, Standard, Dec. 2015.

[15] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[16] L. Bettale, E. Dottax, and L. Grémy, "Post-quantum secure channel protocols for eSIMs," Cryptology ePrint Archive, Paper 2024/2005, 2024. [Online]. Available: https://eprint.iacr.org/2024/2005