

iotops
Internet-Draft
Intended status: Informational
Expires: 14 June 2026

P. Liu, Ed.
R. Yang, Ed.
Pengcheng Laboratory
R. Chen, Ed.
China Mobile
R. Fu, Ed.
China Unicom
Y. Zhang, Ed.
Pengcheng Laboratory
11 December 2025

Modbus Serial Link Communication Security Protocol Specification
draft-liu-iotops-modbus-seriallink-sec-spec-04

Abstract

Currently, the industry has only standardized TLS-based security protocol for Modbus TCP communication. However, for EIA/TIA-485 multi-point systems, whether in 2-wire or 4-wire configurations, cables with a maximum baud rate of 9600 bit/s and AWG26 (or thicker) specifications can reach lengths over 1000m. For RS485 Modbus, sufficient cable diameter supports lengths over 1000m, and Category 5 cables can reach up to 600m. Despite its widespread use, Modbus serial link communication lacks standardized security mechanisms. As an application-layer industrial communication protocol, Modbus urgently requires targeted security standards for different deployment scenarios to enhance its security framework and ecosystem. For instance, in scenarios involving hardware side-channel attacks or long-distance relay and bridged networks, transmitting plaintext Modbus data over serial links risks data leakage or malicious modification by intermediaries, posing significant data and command security threats. To address this, it is critical to improve the communication security of Modbus in serial link mode by introducing encryption and authentication mechanisms. A proposed Modbus serial link security standard defines simple encryption and authentication methods to significantly enhance security while maintaining compatibility. This provides a straightforward upgrade path for existing Modbus-based devices, ensuring secure and practical applications in industrial control systems.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 14 June 2026.

Copyright Notice

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
2. Requirements Language	4
3. Terms defined in this document	4
4. Design Goals	5
5. Modbus Serial Link Communication Security Protocol Specification	5
5.1. Modbus Serial Link Communication Security Service Definition	6
5.2. Modbus Serial Link Security Protocol	7
5.2.1. System workflow	8
5.2.2. Modbus Serial Link Security Protocol Based on Public Key Certificate	10
5.2.3. Modbus Serial Link Security Protocol Based on Password or Pre-Shared Key	54
6. Appendix A	60
7. Appendix B	61
8. Appendix C	62
9. IANA Considerations	66
10. Security Considerations	66
11. References	67

11.1. Normative References	67
11.2. Informative References	67
Authors' Addresses	68

1. Introduction

The Modbus protocol with serial link communication as shown in Figure 1, is widely used in industrial communication, was not originally designed with security in mind. Its lack of encryption, authentication, and data integrity mechanisms makes it vulnerable to various cyberattacks, including replay, man-in-the-middle (MITM), and unauthorized access. Currently, for any EIA/TIA-485 multi-point system, whether it is a 2-wire or 4-wire configuration, the maximum length for cables with a maximum baud rate of 9600bit/s and AWG26 (or thicker) specifications can reach over 1000m. For RS485 Modbus, a sufficiently wide cable diameter should allow for a maximum length of over 1000m, and for RS485 Modbus using Category 5 cables, the maximum length can reach 600m. Currently, there is no relevant security communication standard for the existing Modbus serial link communication method. As an industrial communication protocol at the application level, with the evolution of precision attack technology, Modbus urgently needs to provide targeted security technology standards for different deployment scenarios to continuously improve the Modbus industrial application security guarantee system and ecology. For example, in the presence of modern advanced hardware side channel attacks, especially in the case of long-distance relay serial or bridged networks, transmitting Modbus industrial control protocol data in plaintext over a fully serial link medium can easily lead to data leakage or malicious modification by intermediaries, posing serious data or instruction security risks.

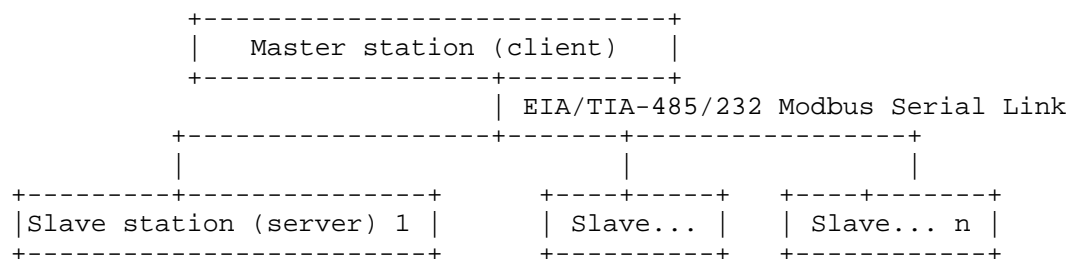


Figure 1: Modbus Serial Link Communication Network

For Modbus TCP, significant progress has been made to address these vulnerabilities. The Modbus Security Specification introduces Transport Layer Security (TLS) to encrypt and authenticate communications. TLS 1.2 or higher is mandated, ensuring protection against eavesdropping and MITM attacks. Additionally, X.509v3 certificates enable mutual authentication between clients and

servers, while role-based access control (RBAC) allows granular security configurations. The Chinese standard GB/T 41868-2022 has further expanded Modbus TCP security by adopting TLS encryption while maintaining compatibility with the original Application Data Unit (ADU) format. Modbus TCP Security uses IANA-registered port 802 for secure communication, providing a clear upgrade path for existing systems.

However, the security of serial link communication (e.g., RS485-based Modbus) remains an unresolved challenge. Serial Modbus transmits data in plaintext, leaving it susceptible to interception, modification, and hardware-based side-channel attacks. There are currently no formal standards addressing security for Modbus over RS485, which is still widely used due to its simplicity and long-distance capabilities. This lack of encryption and authentication mechanisms poses risks, particularly in scenarios involving relays or bridged networks. While Modbus TCP has seen notable advancements in security through TLS-based solutions, serial Modbus communication requires urgent attention. Introducing lightweight encryption and authentication mechanisms for serial links could provide a practical way to enhance security and protect legacy systems without significant infrastructure changes. Therefore, it is urgent to improve and enhance the communication security of Modbus protocol under serial link mode, for reference by relevant institutions and organizations in various industries, in order to ensure the practical application security of various industrial control systems. With the introduction of the Modbus serial link security standard, the introduction of encryption and authentication mechanisms in the serial link communication channel of the Modbus protocol significantly improves its security, providing a relatively simple and direct upgrade path for existing devices that use Modbus extensively.

2. Requirements Language

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

3. Terms defined in this document

1. Trust anchor: An entity that is trusted by relying parties to validate the ownership, delegation, and authenticity of network resources. It serves as a root of trust for establishing and verifying resource allocation chains across multiple infrastructure services through cryptographic mechanisms and distributed consensus.
2. Modbus device: A device that implements Modbus on a serial link and follows its technical specifications.
3. Secure authenticated channel: A logical channel established using password methods between the master and slave stations that supports authentication and encryption of data.
4. Master client: The master client is a device on the serial communication link that sends instruction request and there is usually only one such device on a serial communication link.
5. Slave server: The slave client is a device on the serial communication link that sends instruction response. Generally, there can be multiple such devices on a single serial communication link.
6. Uimsbf: Unsigned integer most significant bit first
7. Bslbf: Bit serial leftmost bit first

4. Design Goals

The Modbus serial link communication security protocol specification aims to provide:

1. The Modbus serial bus security service definition, protocol specifications, system dependencies, and corresponding requirements for the Modbus serial link communication security protocol.
2. Reference application standards for institutions and organizations in the development or testing of Modbus serial communication security products

5. Modbus Serial Link Communication Security Protocol Specification

5.1. Modbus Serial Link Communication Security Service Definition

The Modbus application layer protocol and its standard function codes used on serial links are described in detail in the national standards Modbus Security Specification, GB/T 19582.1-2008 and GB/T 19582.2-2008. The function code field of the Modbus data unit is encoded with one byte, and the valid function code range is decimal 1-255 (with 128-255 reserved for exception response). When sending a message from the client to the server device, the function code field notifies the server of which operation to perform, where the function code '0' is invalid. In the standard Modbus Security Specification, GB/T 19582.1-2008, the following function codes and sub-codes are not included in the content of this standard. These function codes and sub codes are specially reserved in the format of function codes/sub codes, or only function codes, and all sub-codes (0-255) are reserved: 8/19, 8/21-65535, 9, 10, 13, 14, 41, 42, 90, 91, 125, 126, and 127. Please refer to Appendix A of the standard GB/T 19582.1-2008. For the standard function codes defined in Modbus Security Specification, GB/T 19582.1-2008 and GB/T 19582.2-2008, this specification does not make any modifications to the function definitions. In addition, this specification does not occupy the valid business function code resources defined in the standard Modbus Security Specification, GB/T 19582.1-2008 to implement security service functions, while maintaining compatibility with the Modbus serial link protocol. this specification defines the function code "0" for the serial link secure communication service, which occupies an 8-bit byte and is used to define the content related to the Modbus serial link secure communication protocol. Among them, the original Modbus serial link protocol function codes and data fields are included as relevant parameters in the security protocol data to ensure that the corresponding business function codes and data are confidential or complete during serial link communication.

The implementation of serial link secure communication defined in this specification can be achieved by adding a separate security external module or security proxy module to the master and slave devices, or directly integrating relevant security modules on the master and slave devices. The specific implementation form is not within the scope of this specification. In the secure mode of serial link communication, when using Modbus RTU transmission mode, the corresponding relationship between the original Modbus RTU frame and the Modbus secure RTU frame is shown in Figure 2, where the function code and data field of the original RTU frame are encrypted and included in protocol data field of the secure RTU frame:

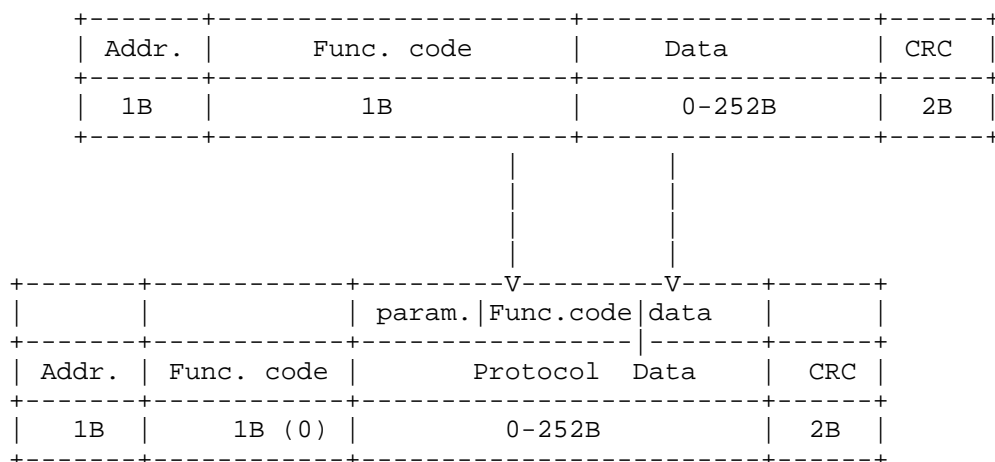


Figure 2: Modbus Secure RTU Message Frame

In the secure mode of serial link communication, when using Modbus ASCII transmission mode, the corresponding relationship between the original Modbus ASCII frame and the Modbus secure ASCII frame is shown in Figure 3, where the function code and data field of the original ASCII frame are encrypted and included in protocol data field of the secure ASCII frame:

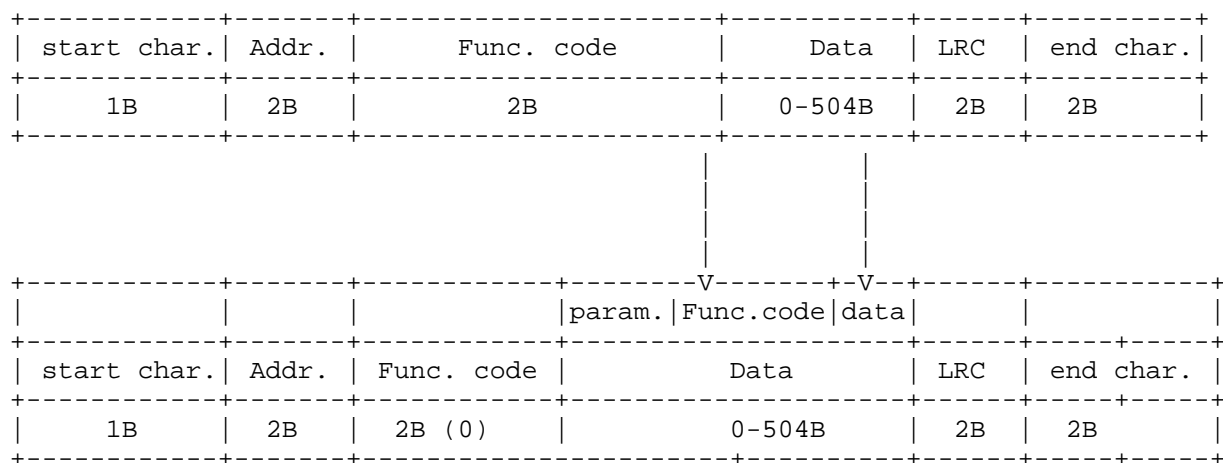


Figure 3: Modbus Secure ASCII Message Frame

5.2. Modbus Serial Link Security Protocol

5.2.1. System workflow

The Modbus serial link security protocol specification defined in this specification aims to protect the transmission of industrial control system protocol parameters between master/slave stations (clients/servers) through the serial bus interface using the Modbus protocol, ensuring that protocol parameters are not maliciously modified or intercepted by intermediaries. On a Modbus serial link, the master station on the serial bus acts as the client and the slave station acts as the server. The specific workflow varies depending on the security protocols used. This specification provides multiple security protocols for industry reference and implementation.

The Modbus serial link security protocol based on public key certificates is mainly implemented through the following six processes, which will be described separately in the following chapters.

1. Bidirectional public key authentication between client and server. When the client and server are bound for the first time, a mutual authentication protocol needs to be executed. Successfully executing the protocol can make the client and server confident in each other's reliability, and can also share a master key between both parties for secure processing of the protocol.
2. Re-authentication of client and server. On the basis of the first binding and sharing of the master key, the mutual authentication protocol is generally not executed when booting up again, but the re authentication protocol is executed.
3. SAC secure channel establishment. After successful authentication, the system enters the process of establishing the SAC secure channel. SAC secure channel is a secure channel established by the original Modbus protocol function code and data encryption and authentication key (referred to as content key).
4. Content key establishment. On the SAC secure channel, the client and server can establish content keys for encrypting the original Modbus protocol function codes and data.
5. Protocol function code and data encryption transmission. On the basis of establishing the content key, when sending a request, the client transmits the encrypted original Modbus protocol request function code and data to the server through Modbus secure RTU or ASCII message frames. The encrypted original Modbus protocol function code and data are decrypted and further

processed on the server. When sending a response, the server transmits the encrypted original Modbus protocol response function code and data to the client through Modbus secure RTU or ASCII message frames. The encrypted original Modbus protocol function code and data are decrypted and further processed on the client.

6. For Modbus broadcast protocol interaction, based on the SAC secure channel, when establishing a content key, the client directly sends the corresponding individual broadcast content key to all slave servers on the bus. The content key for Modbus communication is the same for all slave servers. When sending a broadcast request, the client uses the broadcast content key to transmit the encrypted original Modbus protocol broadcast request function code and data to the server through Modbus secure RTU or ASCII message frames. The encrypted original Modbus protocol function code and data are decrypted and further processed on the server. When sending a response, the server uses the broadcast content key to transmit the encrypted original Modbus protocol response function code and data to the client through Modbus secure RTU or ASCII message frames. The encrypted original Modbus protocol function code and data are decrypted and further processed on the client.

The Modbus serial link security protocol based on passwords or pre-shared keys mainly includes the following four processes, which will be described separately in the following chapters.

1. Bidirectional authentication between client and server. When the client and server are bound for the first time, a mutual authentication protocol needs to be executed. Successfully executing the protocol can make the client and server confident in each other's reliability, and can also share a master key between both parties for secure processing of the protocol.
2. Content encryption key establishment. The client and server establish two content encryption keys based on passwords or pre shared keys to encrypt and protect the original Modbus protocol function codes and data, one for broadcasting Modbus communication and the other for separate communication between the slave and master stations. The content key used for broadcasting Modbus communication is the same for all slave servers.
3. Protocol function code and data encryption transmission. On the basis of establishing the content key, when sending a request, the client transmits the encrypted original Modbus protocol request function code and data to the server through Modbus

secure RTU or ASCII message frames. The encrypted original Modbus protocol function code and data are decrypted and further processed on the server. When sending a response, the server transmits the encrypted original Modbus protocol response function code and data to the client through Modbus secure RTU or ASCII message frames. The encrypted original Modbus protocol function code and data are decrypted and further processed on the client.

4. Regarding Modbus broadcast protocol interaction. When sending a broadcast request, the client uses the broadcast content key to transmit the encrypted original Modbus protocol broadcast request function code and data to the server through Modbus secure RTU or ASCII message frames. The encrypted original Modbus protocol function code and data are decrypted and further processed on the server. When sending a response, the server uses the broadcast content key to transmit the encrypted original Modbus protocol response function code and data to the client through Modbus secure RTU or ASCII message frames. The encrypted original Modbus protocol function code and data are decrypted and further processed on the client.

5.2.2. Modbus Serial Link Security Protocol Based on Public Key Certificate

5.2.2.1. Key hierarchy structure

The key management in the Modbus serial link security protocol based on public key certificates is suitable for high-value and high security factories and devices. Its trust model can be found in Appendix C, and the key management scheme is divided into four levels: Credentials layer, Authentication layer, SAC secure channel layer, and Content key layer. The names, storage, and usage of the main parameters are summarized as below, where the requirement for random numbers generators can be found in Appendix B.

1. root certificate: Certificate of system trust root, which is non-volatile stored and local use.
2. Brand certificate: Brand certificate of device manufacturer, which is non-volatile stored and used for exchange.
3. Device certificate: Public key certificates for the main and slave devices under the brand, which is non-volatile stored and used for exchange.
4. EC_p: ECC Parameters, which is non-volatile stored and local use.

5. EC_a: ECC Parameters, which is non-volatile stored and local use.
6. EC_b: ECC Parameters, which is non-volatile stored and local use.
7. EC_N: ECC Parameters, which is non-volatile stored and local use.
8. EC_G: ECC Parameters, which is non-volatile stored and local use.
9. MDP/HDP: Device Public Key, which is non-volatile stored and exchange use.
10. MDQ/HDQ: Device private key, which is non-volatile stored and local use.
11. DHX/DHY: The power of DH modular exponentiation operation, which is volatile stored and local use.
12. DHPM/DHPH: The public key generated after DH modular exponentiation operation, which is volatile stored and exchange use.
13. DHSK: DH key (master key), which is non-volatile stored and local use.
14. AKM: Master authentication key, which is non-volatile stored and local use.
15. AKH: Slave authentication key, which is non-volatile stored and exchange use.
16. Ns_M/Ns_H: Random numbers used for SAC channel establishment, which is volatile stored and exchange use.
17. SEK: SAC data encryption key, which is volatile stored and local use.
18. SAK: SAC Data Authentication Key, which is volatile stored and local use.
19. SIV: SAC initialization vector, which is non-volatile stored and local use.
20. Kp: Precursor used to calculate CK key, which is volatile stored and exchange use.

21. CK/BCK: Content encryption key for Modbus protocol function code and data, which is volatile stored and local use.
22. CIV/BCKIV: Content encryption initialization vector, which is volatile stored and local use.

The keys at the credentials layer mainly include ECC public cryptographic protocol parameters, private keys of the master client and slave server, and public key certificate chains. The public key certificate chain includes root certificates, brand certificates, and device certificates. The device certificate includes the device's ID number, ECC public key, and other attribute information of the certificate. The key data at this level is mainly used to execute authentication protocols. After device initialization, the certificate must not be changed or replaced. It is recommended to use mature OTP memory based on eFuse or Anti-Fuse technology, one-time programmable non-volatile memory, and data that cannot be reversibly modified after writing. The definition of protection methods for these data is not within the scope of this specification.

For the master client, the keys at the authentication layer are mainly DHSK and AKM; for the slave server, there are mainly two keys, namely DHSK and AKH. Among them, DHSK is a shared master key negotiated after a two-way authentication protocol, while AKM and AKH are exported by DHSK. These three keys are long-term keys that should be stored in the device's NVM and must be guaranteed not to be modified or intercepted by third parties. It is recommended to use a mature TEE environment, and the protection methods for these keys are not within the scope of this specification.

The SAC secure channel layer has two keys, SAK and SEK. The former is used for message authentication based on SAC, while the latter is used for message encryption based on SAC. The SAK and SEK keys are generated through the SAC initialization protocol executed by both parties and are temporary keys shared between the client and server. It should be stored in the NVM of the device and must ensure that it cannot be modified or accessed by third parties. It is recommended to use a mature TEE environment, and the definition of protection methods for SAK and SEK keys is not within the scope of this specification.

The content key layer first generates the content key CK and an initial vector CIV based on the SAC secure channel, while generating the keys BCK and BCIV required for broadcast communication. Then, the content key CK and initial vector CIV are used to encrypt and decrypt Modbus communication data using AES. The generation of CK key and vector CIV key is led by the master client, and the slave server obtains the same CK key and vector CIV through the SAC secure

channel. Both are generated temporarily, and under certain conditions, the system must update the content key to provide high-strength security. Please refer to the following chapters for detailed instructions.

The overall key generation and usage process is as follows:

After power on start-up, the master and slave devices first perform a power on self-test. If it is normal, the master client should check if it can be re authenticated after power on; If the master client and slave server devices have been successfully bound before, re authentication is not necessary. Under the premise of successful binding, the two stages of certificate verification, DH key exchange, and authentication key verification can be skipped and directly enter the SAC channel establishment process. Otherwise, SAC authentication key generation for identity authentication must be performed first; After the SAC channel is established, the master and slave devices enter the content key establishment process. After the SAC channel and content key are established, the master and slave devices use content key encryption to protect the corresponding business protocol parameters during business operation. If for some reason, such as connection establishment failure or device busy, the protocol runs out of time and SAC or content key cannot be updated in a timely manner, corresponding errors can be prompted to device users or administrators on the industrial site.

5.2.2.2. Certificate and identity authentication mechanism

Authentication based on public key certificates includes certificate verification and DH key exchange, as well as verification of intermediate authentication keys. Firstly, the authentication process of the first stage master client and slave server begins by exchanging their respective certificate chains and DH public keys. The master client must verify the certificate chain of the slave server and the checksum generated by the slave server, and similarly, the slave server must verify the certificate chain of the master client and the checksum generated by the master client. During the verification process, the slave server will extract the CLIENT ID (64 bit data) from the master client's public key certificate, and similarly, the master client will also extract the SERVER ID (64 bit data) from the slave server's public key certificate.

Then, in the second stage, the data exchanged between the slave server and the master client using the authentication protocol in the first stage can obtain a long-term DH secret key and a long-term authentication key. The authentication key is obtained by calculating the DH secret key and the IDs of both parties. Both parties should store the authentication key in NVM for future

calculation of SAC secure channel encryption key and content key. Afterwards, the slave server will send a request to the master client to confirm the authentication key. Upon receiving the confirmation message, the slave server must compare the received key value with the stored key value. If the two match, it can prove that the master client is legitimate.

The steps of the device authentication protocol based on public key certificates are shown in Figure 4, and the message semantics and process are explained as follows:

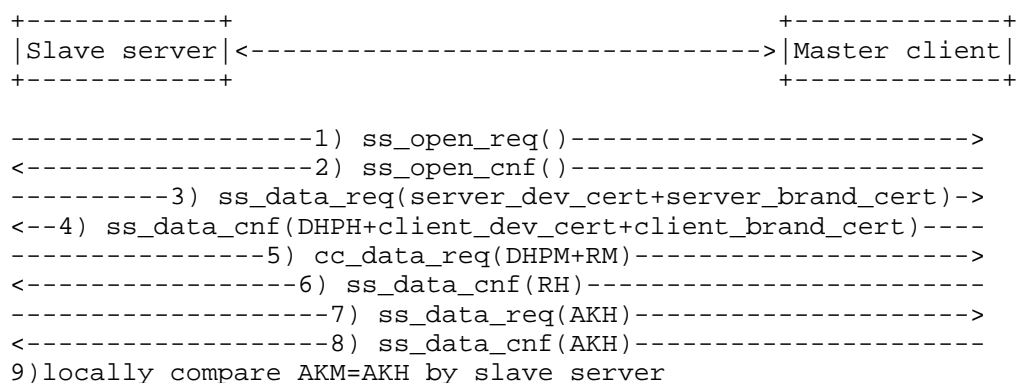


Figure 4: Authentication Message Exchange process

1. Send ss_open_req APDU from the slave server to the master client.
2. The master client responds with ss_open_cnf APDU.
3. Send ss_data_req APDU from the slave server to the master client. The message contains: 1) Device certificate (Server_dev_Cert) of slave server; 2) Brand certificate (Server_Brand_Cert) of slave server; 3) ID of the requested data type.
4. The master client responds with ss_data_cnf APDU. The message contains: 1) DH public key (DHPH) of master client; 2) Device certificate for master Client (Client_Dev_Cert); 3) Brand certificate (Client_Brand_Cert) of master client.
5. The slave server then sends the ss_data_req APDU. The message contains: 1) DH public key (DHPM) of slave server; 2) Verification value RM; 3) ID of the requested data type.
6. The master client responds with ss_data_cnf APDU. The message contains: 1) A verification value RH.

7. The master client responds with `ss_data_cnf` APDU. The message contains: 1) The data type ID of the requested AKH.
8. The master client responds with `ss_data_cnf` APDU. The message contains a AKH, Either it is valid or it is all zero, indicating illegality.
9. The slave server compares the received AKH with the locally calculated AKM.

1) The definition of `ss_open_req` APDU is as follows:

The slave server sends this APDU to inquire about the ID mask of the encryption system supported by the master client. The syntax is shown in Figure 5.

syntax	No. of bits	Mnemonic
<pre>ss_open_req() { ss_open_req_tag length_field()=0 }</pre>	24	uimsbf

Figure 5: `ss_open_req` APDU syntax

1. `ss_open_req_tag`: 0x9F 90 01.
2. `length_field`: The length of the APDU payload, expressed in ASN. 1 BER format.

2) The definition of `ss_open_cnf` APDU is as follows:

The master client sends this APDU to the slave server, notifying them of the encryption system IDs supported by the slave server. The syntax is shown in Figure 6.

syntax	No. of bits	Mnemonic
ss_open_cnf() {		
ss_open_cnf_tag	24	uimsbf
length_field()=0		
ss_system_id_bitmask	8	bslbf
}		

Figure 6: ss_open_cnf APDU syntax

1. ss_open_cnf_tag: 0x9F 90 02.
2. ss_system_id_bitmask: Each of the 8 bits indicates the encryption system version supported by a master client. The slave server can choose the highest version supported by both ends. The lowest bit indicates version 1, there is no version 0. The encrypted version defined in this section is 1. ss_system_id_bitmask lowest position 1 (indicating support for version 1).
- 3) The definition of ss_data_req APDU is as follows:

The slave server sends an ss_data.req message to the master client to transmit protocol related data, while requesting a response from the master client. The ss_data_req APDU syntax is shown in Figure 7. ss_data_req APDU is used for data transmission that does not require authentication or encryption. Mainly used for authentication, authentication key verification, and SAC key calculation.

syntax	No. of bits	Mnemonic
ss_data_req() {		
ss_data_req_tag	24	uimsbf
length_field()=0		
ss_system_id_bitmask	8	bslbf
send_datatype_nbr	8	uimsbf
for (i=0;i<send_datatype_nbr;		
i++) {		
datatype_id	8	uimsbf
datatype_length	16	uimsbf
data_type	8*datatype_length	bslbf
}		
request_datatype_nbr	8	uimsbf
for (i=0;		
i<request_datatype_nbr;		
i++){		
datatype_id	8	uimsbf
}		
}		

Figure 7: ss_data_req APDU syntax

1. ss_data_req_tag: 0x9F 90 03.
2. ss_system_id_bitmask: Each of the 8 bits indicates the encryption system version supported by a master client. The slave server can choose the highest version supported by both ends. The lowest bit indicates version 1, there is no version 0. The encrypted version defined in this section is 1. ss_system_id_bitmask lowest position 1 (indicating support for version 1).
3. send_datatype_nbr: This message contains the number of data categories.
4. datatype_id: Data type ID, see Appendix A.
5. datatype_length: The byte length of datatype shaped data.
6. data_type: The specific data represented by datatype_id.
7. request_datatype_nbr: The number of data categories that the master client should include in its response.

8. datatype_id: The data number that should be included in the response of the master client, see Appendix A.

4) The definition of ss_data_cnf APDU is as follows:

The master client sends an ss_data_cnf message to transmit protocol related data to the slave server. The ss_data_cnf APDU syntax is shown in Figure 8. ss_data_cnf APDU is used for data that does not require authentication or encryption, mainly for authentication, authentication key verification, and SAC key calculation. For data that requires authentication or encryption, use ss_sac_data_cnf APDU.

syntax	No. of bits	Mnemonic
ss_data_cnf() {		
ss_data_cnf_tag	24	uimsbf
length_field()=0		
ss_system_id_bitmask	8	bslbf
send_datatype_nbr	8	uimsbf
for (i=0;i<send_datatype_nbr;		
i++) {		
datatype_id	8	uimsbf
datatype_length	16	uimsbf
data_type	8*datatype_length	bslbf
}		

Figure 8: ss_data_cnf APDU syntax

1. ss_data_cnf_tag: 0x9F 90 04.
2. ss_system_id_bitmask: Each of the 8 bits indicates the encryption system version supported by a master client. The slave server can choose the highest version supported by both ends. The lowest bit indicates version 1, there is no version 0. The encrypted version defined in this section is 1.
 ss_system_id_bitmask lowest position 1 (indicating support for version 1).
3. send_datatype_nbr: This message contains the number of data categories.
4. datatype_id: Data type ID, see Appendix A.
5. datatype_length: The byte length of datatype shaped data.
6. data_type: The specific data represented by datatype_id.

The protocol APDU format that includes specific data types in the authentication protocol is shown as follows:

When sending public key certificate chain from the slave server to the master client, ss_data_req APDU is sent with following Parameters in Figure 9.

content		
send_datatype_nbr=2		
i	datatype_id	datatype_len
0	11(Server_Dev_Cert)	1280 bits
1	4(Server_Brand_Cert)	1280 bits
request_datatype_nbr=3		
i	datatype_id	
0	8 (DHPH)	
1	10 (Client_Dev_Cert)	
2	3 (Client_Brand_Cert)	

Figure 9: ss_data_req APDU content

When the master client sends DHPH and two public key certificates, ss_data_cnf APDU is sent with following Parameters in Figure 10.

content		
send_datatype_nbr=3		
i	datatype_id	datatype_len
0	8(DHPH)	512 bits
1	10(Client_Dev_Cert)	1280 bits
2	3(Client_Brand_Cert)	1280 bits

Figure 10: ss_data_cnf APDU content

When sending DHPM and checksum RM from the slave server, ss_data_req APDU is sent with following Parameters in Figure 11.

content		
send_datatype_nbr=4		
i	datatype_id	datatype_len
0	9(DHPM)	512 bits
1	12(RM)	256 bits
2	11(Server_Dev_Cert)	1280 bits
3	4(Server_Brand_Cert)	1280 bits
request_datatype_nbr=1		
i	datatype_id	
0	20 (status_field)	

Figure 11: ss_data_req APDU content

When the master client sends the verification value RH, ss_data_cnf APDU is sent with following Parameters in Figure 12.

content			
send _datatype_nbr=1			
i	datatype_id	datatype_len	
0	13(RH)	256 bits	

Figure 12: ss_data_cnf APDU content

When requesting AKH from the slave server to the master client,
ss_data_req APDU is sent with following Parameters in Figure 13.

content			
send _datatype_nbr=1			
i	datatype_id		
0	17(AKH)		

Figure 13: ss_data_req APDU content

When the master client sends AKH, ss_data_cnf APDU is sent with
following Parameters in Figure 14.

content			
send _datatype_nbr=1			
i	datatype_id	datatype_len	
0	17(AKH)	256 bits	

Figure 14: ss_data_cnf APDU content

Note: During the certificate authentication phase, APDU messages will encapsulate the Modbus serial link data unit field with a success number of 0. For APDUs larger than 252 bytes (the maximum data field length of Modbus RTU frames), such as ss_data_req APDUs, which exchange certificates with approximately 422 bytes, they can be divided into two Modbus serial link message frames for transmission. Since the transmission operation is strictly serial and has a sequence, the upper layer application can concatenate two or more segmented APDUs based on the tag (ss_data_cnf_tag) and length_field () fields of the protocol.

The authentication process of the slave server is briefly described as follows:

1. Before starting the authentication process from the server, all sessions should have been opened;
2. Send its own certificate chain from the slave server;
3. The slave server waits for a response message from the master client;
4. The slave server must verify the validity of the certificate responded by the master client. If the certificate is invalid, proceed to step 19);
5. The slave server must verify the validity of the DHPH value. If it is invalid, proceed to step 19);
6. Generate a random DHY from the slave server;
7. Calculate the DH public key DHPM from the slave server;
8. The slave server must check if the DHPM value is valid. If it is invalid, proceed to step 19);
9. Generate verification value RM from the slave server;
10. Sending protocol data from the slave server;
11. The slave server waits for the master client to send back a response message;
12. The slave server must check whether the RH of the response from the master client is correct. If not correct, proceed to step 19);
13. Calculate AKM from the slave server;

14. Request AKH from the slave server to the master client;
15. The slave server should receive AKH within 5 seconds. Otherwise, proceed to step 19);
16. The slave server must check if the received response contains a valid AKH. If the received AKH is invalid, proceed to step 19);
17. The slave server must check whether the received AKH is consistent with AKM. If there is inconsistency, proceed to step 19);
18. The authentication of the slave server has been successfully completed;
19. Send error message (optional);
20. Authentication failed.

The authentication process of the master client is briefly described as follows:

1. Session is successfully opened;
2. The master client receives a certificate chain from the slave server;
3. The master client must check if the received certificate is valid;
4. The master client generates a random DHX;
5. The master client calculates the DH public key DHPH;
6. The master client must check whether the calculated DHPH is valid;
7. The master client sends DHPH and master client certificate chain to the slave server;
8. The master client is waiting for a response message from the slave server;
9. The master client must verify that the DHPM received from the slave server is valid;
10. The master client must verify whether the received slave server checksum is valid;

11. The master client calculates the verification value RH;
12. The master client sends RH to the slave server;
13. The master client calculates AKH;
14. Authentication successful;
15. Any error will result in authentication failure;
16. The master client receives a request from the slave server;
17. The master client responds with AKH value;
18. Authentication completed.

Note: The slave server can repeatedly send requests to AKH until the master client responds.

The calculations involved in the certificate based authentication process are shown in Figure 15, and described as below.

```

+-----+                                     +-----+
|Slave server|<----->|Master client|
+-----+                                     +-----+

-----1) Send the certificate chain----->
      2) Verify certificate by client locally---
      3) Generate random number DHX by client locally--
          4) Calculate DPHH by client locally---
<-----5) Send DPHH and certificate chain-----
----6) Verify certificate by server locally
----7) Generate random number DHY by server locally
----8) Calculate DHPM by server locally
----9) Calculate DHSK and RMby server locally
-----10) Send DHPM and RM----->
      11) Check parameters by client locally---
      12) Calculate DHSK and RH by client locally--
<-----13) Send RH-----
--14) Check parameters and calculate AKM by server locally
      15) Calculate AKH by client locally--
-----16) Request AKH----->
<-----17) Responds AKH-----
--18) Compare AKM=AKH by server locally

```

Figure 15: Calculation steps involved in the authentication process

1. The slave server sends its public key certificate chain to the master client
2. The master client verifies the validity of the two received certificates from the slave servers.
3. The master client uses an appropriate PRNG to generate a random DH exponent x (DHX).
4. The master client calculates its own DH public key (DHPH).
5. The master client sends the DHPH, master client device certificate, and master client device brand certificate to the slave server.
6. The slave server verifies the validity of the two certificates received from the master client.
7. The slave server generates a random DH exponent y (DHY) using an appropriate PRNG.
8. The slave server computes its own DH public key (DHPM) .
9. The slave server calculates the verification value RM as follows: a) Firstly, use the inverse element MDQ-1 of its private key to calculate $xG = E_M(MDQ-1, DHPH)$; b) Calculate and store $DHSK = E_M(y, xG)$; c) Calculate $RM = SM3(xG || DHPM || DHSK || SERVER_ID)$; Note: $E.M(*, *)$ represents scalar multiplication on an elliptic curve, where the first parameter is an element in a finite field and the second parameter is a point on the elliptic curve.
10. The slave server sends DHPM and RM to the master client.
11. The master client verifies the received parameters as follows: a) Firstly, use the inverse element HDQ-1 of the private key to calculate $yG = E_M(HDQ-1, DHPM)$; b) Calculate and store $DHSK = E_M(x, yG)$; c) Calculate $RM' = SM3(xG || DHPM || DHSK || SERVER_ID)$; d) Verify whether the received RM is consistent with the calculated RM '. Note: $E.M(*, *)$ represents scalar multiplication on an elliptic curve, where the first parameter is an element in a finite field and the second parameter is a point on the elliptic curve.
12. The master client uses the data calculated in steps 4) and 11) to calculate the verification value RH: $RH = SM3(yG || DHPH || DHSK || CLIENT_ID)$.

13. The master client sends RH to the slave server.
14. The slave server verifies the received parameters using the data calculated in steps 7) and 9): a) Calculate $RH' = SM3(yG || DPH || DPHM || CLIENT_ID)$; b) Verify whether the received RH is consistent with the calculated RH '. The slave server calculates and stores the authentication key (AKM).
15. The master client calculates and stores the authentication key (AKH).
16. The slave server start to enter the second stage of the authentication process: request the current authentication key AKH from the master client through the response APDU.
17. The master client uses the corresponding APDU to send AKH to the slave server.
18. The slave server compares the received AKH with its own calculated AKM. If they are different, it will result in a system error.

The DH public keys (DPH and DHPM) are temporary data that should be cleared and deleted after the authentication protocol is completed.

The DH public key is calculated as follows:

$$DPH = E_M(x, MDP), \quad DHPM = E_M(y, HDP)$$

Note that in the above equation:

The values x (DHX) and y (DHY) are generated by the built-in PRNG of the device. The values of x (DHX) and y (DHY) should be kept secret locally and cleared and deleted after the authentication protocol ends. In order to generate secure protocol parameters, after the master client calculates DPH, the following checks must be performed:

DPH is neither an infinite point nor a generator G, and the value of $E-M(EC-N, DPH)$ is an infinite point.

Similarly, after calculating DHPM from the station server, the following checks must be performed:

DHPM is neither an infinite point nor a generator G, and the value of $E-M(EC-N, DHPM)$ is an infinite point.

The authentication keys (AKH and AKM) can be used in the calculation process of SAC keys and CK. The authentication key is generated when the master client and slave server are first bound and stored in NVM, which can be used for repeated authentication in the future. The calculation process is as follows:

$$AKM = AKH = SM3(SERVER_ID || CLIENT_ID || DHSK)$$

The relevant parameter columns are shown in Figure 16.

Parameter	No. of bits	description
DHSK	512	The DH secret key shared during the authentication process.
CLIENT_ID	64	Included in public key certificate of the master client device.
SERVER_ID	64	Included in public key certificate of the slaver serve device.

Figure 16: Parameters in Authentication Key Calculation

Authentication context and re-authentication: Authentication context is a secure data object stored in NVM, and its storage format is not strictly defined in this specification. However, in the implementation of actual products, it should be given sufficient security to ensure that it is not changed by attackers. The authentication context is used to store the results of public key authentication between both parties of the device, and is used to perform the re authentication process. The information that should be included in the authentication context of the slave server and master client is shown in Figure 17.

+-----+-----+	
Slave server	Master client
+-----+-----+	
CLIENT_ID	SERVER_ID
DHSK	DHSK
AKM	AKM
Encryption mode	Encryption mode
+-----+-----+	

Figure 17: Information in Authentication Context

After the master client and slave server are first bound, the device should associate the DHSK and authentication key negotiated during the public key authentication process with an authentication context. The master client should support at least 247 authentication contexts, and the slave server should support at least 1 authentication context.

After establishing a session, the master client and slave server can first perform an authentication key verification step to check if there is an existing binding relationship between the two devices. If there is, there is no need to perform a public key authentication protocol. If the slave server has a valid authentication context, it should request the master client to send an AKH and then check if the received AKH matches the AKM in the local authentication context. If there is inconsistency, the slave server shall attempt up to 246 more times (as the master client may have 247 authentication contexts). When there are no other available authentication contexts for the master client, the master client should send a zero value as a response to the AKH request from the slave server, and the slave server should start executing the public key authentication protocol. If the master client has the authentication context of the slave, it should first send the AKH in the context of the slave. If the AKH is inconsistent with the AKM of the slave server, the slave server should attempt to request AKH again, and the master client should send other AKHs in other authentication contexts. So, re-powering on the slave server may not necessarily execute the public key authentication protocol. But when the slave server accesses an unbound master client, or when the master client accesses an unbound slave server, the identity authentication protocol related to the public key certificate is required to be executed.

5.2.2.3. Establishing secure authentication channel Mechanism

After authentication is completed, the protocol will establish a secure authentication channel (SAC), which is mainly used for content key establishment and transmission. Figure 18 is a schematic diagram of its function.

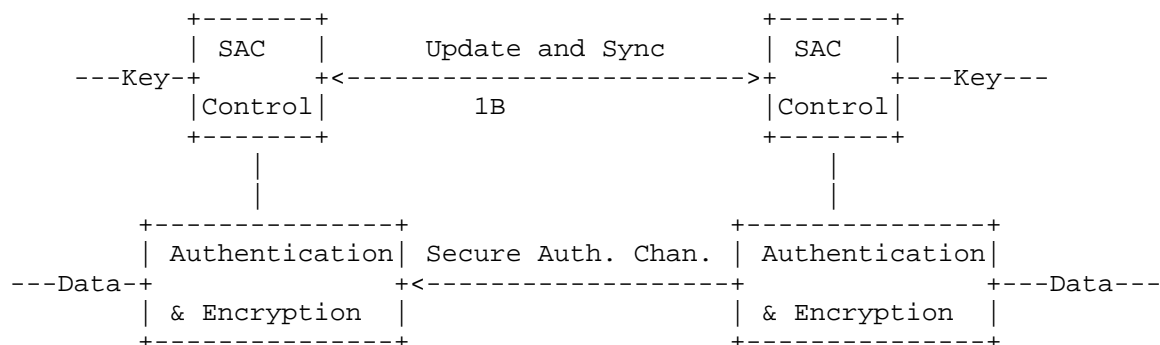


Figure 18: Schematic diagram of SAC function

Figure 19 shows the workflow of SAC channel establishment, which is described as follows.

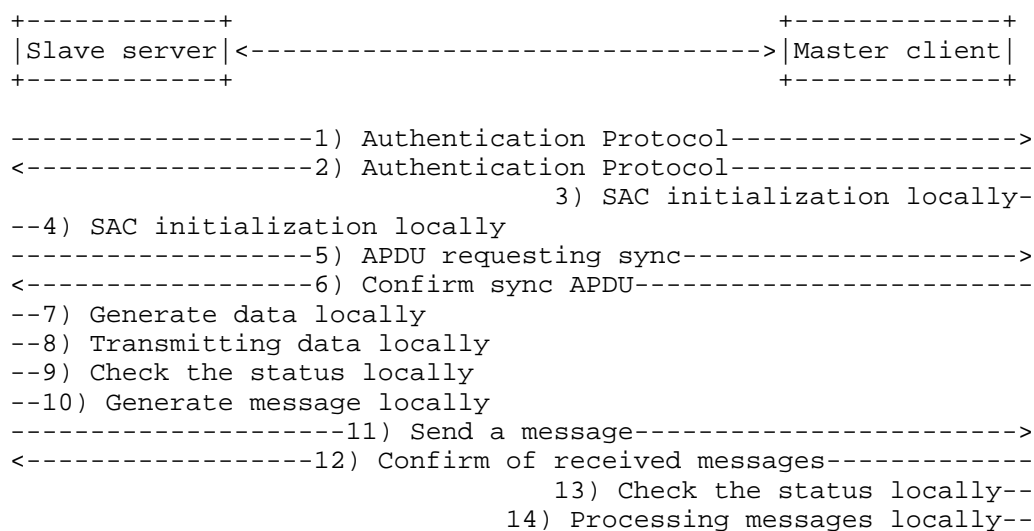


Figure 19: The workflow of SAC channel establishment

1. 1),2) Authentication protocol. The slave server and master client must have successfully executed the mutual authentication protocol.
2. 3),4) SAC initialization. Both the slave server and the master client need to first perform SAC initialization to generate key data and set the initial state of SAC.

3. 5),6) SAC synchronization request and confirmation. If the slave server has correctly initialized SAC, the slave server should send an APDU requesting synchronization to the master client. After the master client successfully responds, both parties can start using SAC.
4. 7)-11) Generate and send SAC messages. Generate SAC messages by adding message headers and authentication domains to the payload and encrypting them (optional).
5. 12)-14) Receive and verify SAC messages. Validate the received SAC message and proceed with further processing under the condition that the message is valid. Note: SAC can be used for bidirectional sending and receiving of messages;

Figure 20 shows the SAC initialization process, which is described as follows.

```

+-----+                                     +-----+
|Slave server|<----->|Master client|
+-----+                                     +-----+

--1) SAC initialization conditions are met
--2) Generate Ns_S locally
-----3) ss_data_req (Ns_M+SERVER_ID)----->
--4) Generate Ns_H locally
<-----5) ss_data_cnf (Ns_H+CLIENT_ID)-----
--6) Export SAK and SEK, reset SAC status locally
      7) Export SAK and SEK, reset SAC status---
-----8) ss_sync_req()----->
<-----9)confirm ss_sync_cnf()-----

```

Figure 20: The workflow of SAC channel establishment

1. When the slave server detects that the conditions are met, it initiates the SAC initialization process.
2. The slave server generates a random number Ns_S, which is used as part of the data involved in SAC key calculation.
3. The slave server sends the ss_data_req APDU to the master client, including the following parameters: 1) Ns_M; 2) SERVER_ID.
4. The master client generates a random number Ns_H, which is used as part of the data for SAC key calculation.

5. After receiving the `ss_data_req` APDU from the slave server, the master client should respond to the slave server with the `ss_data_cnf` APDU, which includes the following parameters: 1) `Ns_H`; 2) `CLIENT_ID`.
6. The slave server checks whether the received `CLIENT_ID` matches the pre-stored one. If they match, start calculating SAK and SEK and set or reset SAC status. If they do not match, it will cause a system error. Note: The pre-stored `SERVER_ID/CLIENT_ID` is in the "authentication context".
7. The master client checks whether the received `SERVER_ID` matches the pre-stored one. If they match, start calculating SAK and SEK and set or reset SAC status. If they do not match, it will cause a system error. Note: The pre-stored `SERVER_ID/CLIENT_ID` is in the "authentication context".
8. The slave server sends `ss_stync_req` APDU to the master client to indicate that the master client SAK needs to be refreshed. When the slave server has completed initialization, it will send a synchronization request indicating that the slave server is ready to use SAC.
9. The master client sends `ss_sync_cnf` APDU to the slave server as a confirmation response, indicating that the master client is also ready to use SAC. Failure to respond to `ss_sync_cnf` messages will result in a system error.

The SAC initialization protocol is initiated by the slave server and passively responded by the master client. This protocol is triggered by the following conditions:

1. Power on restart; After system start-up or restart and there is a valid AKM in NVM.
2. Certification; When there is no valid AKM in NVM, the system will enter the re authentication protocol, and after the authentication protocol is completed, trigger the SAC initialization protocol.
3. Message counter overflow.

APDU definition used during SAC initialization process:

The definitions of `ss_open_req` and `ss_open_cnf` APDU are the same as those in the authentication mechanism in the previous section.

- 1) `ss_sync_req` APDU

Used for SAC key calculation, the slave server sends the APDU after the key calculation is completed to indicate that it is ready to use the newly calculated key. The syntax of `ss_sync_req` APDU is shown in Figure 21.

syntax	No. of bits	Mnemonic
<pre>ss_sync_req() { ss_sync_req_tag length_field()=0 }</pre>	24	uimsbf

Figure 21: `ss_sync_req` APDU syntax

`ss_sync_req_tag:0x9F 90 05.`

2) `ss_sync_cnf` APDU

This APDU is the response of the master client to the `ss_sync_req` APDU, indicating that the master client has completed the key calculation. The syntax of the `ss_sync_cnf` APDU is shown in Figure 22.

syntax	No. of bits	Mnemonic
<pre>ss_sync_cnf() { ss_sync_cnf_tag length_field()=0 status_field }</pre>	24	uimsbf
	8	uimsbf

Figure 22: `ss_sync_cnf` APDU syntax

`ss_sync_cnf_tag: 0x9F 90 06.`

`status_field`: This byte returns the status of the master client. Figure 23 lists possible values.

status_field	value
OK	00
No resource	01
Busy	02
Authentication failure	03
Reserved	04-FF

Figure 23: Possible values of status_field

The APDU format of the SAC initialization protocol, which includes specific data types, is shown as below:

When the slave server sends SERVER_ID and a random number, ss_data_req APDU is sent with following Parameters in Figure 24.

content		
send_datatype_nbr=2		
i	datatype_id	datatype_len
0	2(SERVER_ID)	64 bits
1	16(Ns_M)	64 bits
request_datatype_nbr=2		
i	datatype_id	
0	1(CLIENT_ID)	
1	15(Ns_H)	

Figure 24: ss_data_req APDU content

When the master client responds with CLIENT_ID and a random number, ss_data_cnf APDU is sent with following Parameters in Figure 25.

content			
send _datatype_nbr=2			
i	datatype_id	datatype_len	
0	1(CLIENT_ID)	64 bits	
1	15(Ns_H)	64 bits	

Figure 25: ss_data_cnf APDU content

Note: During the secure channel establishment phase, APDU messages will encapsulate the Modbus serial link data unit field with a success code of 0, which is much smaller than 252 bytes (the maximum data field length of a Modbus RTU frame) of APDU. One Modbus serial link message frame can complete the interaction of APDU.

The operation process of the master client during SAC initialization is shown as below.

1. Master client receives “Ns_M” .
2. Master client sends “Ns_H” .
3. Master clientcalculates SAK and SEK.
4. Master client receives synchronization request.
5. Master client confirms synchronization and starts using the new SAC key

The operation process of the slave server during SAC initialization is shown as below.Note: If the condition in step 6 (or step 9) is not met more than 3 times, it will result in a system error.

1. Start SAC initialization.
2. Set the SAC key update timer to 0 seconds on the slave server.
3. Slave server sends 'Ns_M'.
4. Slave server receives 'Ns_H'.
5. Slave server calculates SAK and SEK.
6. SAC key update timer is not more than 9s? If not, goto step 1).

7. If yes, Slave server sends sync request.
8. Slave server received sync confirmation within 10s?
9. If not, Key update timer is not more than 10s? If not, goto step 1).
10. If yes, Wait for 1 second, goto step 7).
11. Slave server allows SAC operations.
12. Set message counter msg_counter = 1 on the slave server
13. Any messages need to be sent? If not, goto step 13).
14. If yes, msg_counter is overflow? If yes, goto step 1).
15. If not, send a message from the slave server and add msg_counter + 1, goto step 13).

Calculating the authentication key SAK and encryption key SEK for SAC requires two steps:

1. Calculation of key seed Ks.
2. Export SAK and SEK.

The key seed Ks has a length of 256 bits. The slave server and master client should calculate the value of the key seed, and the master client must use the following method to calculate Ks_client:

$$Ks_client = SM3(DHSK || AKH || Ns_H || Ns_M)$$

The slave server must calculate Ks_server using the following method:

$$Ks_server = SM3(DHSK || AKM || Ns_H || Ns_M)$$

Under normal circumstances, Ks_server = Ks_client, this two keys herein are named Ks, with the parameters shown in Figure 26

parameter	No. of bits
DHSK	512
AKH/AKM	256
Ns_H	64
Ns_M	64

Figure 26: Parameters in SAC Key Calculation

The SAK and SEK exports are calculated as follows:

$SEK = MSB_{128}(K_s)$, $SAK = LSB_{128}(K_s)$

The SAC message format and semantic description are as follows:

The data messages transmitted as SAC payloads should be encapsulated in SAC messages, with the encapsulation format shown in Figure 27.

Messg. counter	Header	Payload	MAC
+<----overhead----->+<-----Encryption (optional)---->+			

Figure 27: SAC packaging payload

The specific definition of SAC message syntax is shown in Figure 28.

syntax	No. of bits	Mnemonic
message() { message_counter /* Message header start */ protocol_version authentication_cipher_flag payload_encryption_flag encryption_cipher_flag reserved_for_future_use length_payload /* Message header end */ /* Message body start */ if (payload_encryption_flag == MSG_FLAG_TRUE) { encrypted_payload }elif (payload_encryption_flag == MSG_FLAG_FALSE) { payload authentication } /* Message body end */ }	 32 4 3 1 3 5 16 length_payload * 8+128 length_payload * 8 128	 uimsbf uimsbf uimsbf bslbf uimsbf bslbf uimsbf bslbf bslbf bslbf

Figure 28: SAC message syntax

Encoding and semantics of fields in SAC messages:

1. message_comounter: Data messages require a count value to identify the number of messages that have been transmitted.
2. protocol version: In this section, set to 0x0.
3. authentication_cipher_flag: This parameter specifies the algorithm that generates the authentication code, currently only 0x0 is valid, means AES-128-XCBC-MAC, other values are reserved.
4. payload_decryption-flag: This parameter indicates whether the payload is encrypted. '1' indicates encryption, '0' indicates no encryption. Devices that comply with this section are explained as "1" and other unsupported values are ignored.
5. encryption_cipher_flag: This parameter indicates the encryption algorithm, currently only 0x0 is valid, means AES-128, CBC mode, other values are reserved.

6. `length_payload`: This parameter represents the number of bytes in the payload, including padding data but not the number of bytes in the authentication code.
7. `encrypted_payload`: This field consists of encrypted data, including ciphertext for message payload, padding data, and authentication code.
8. `payload`: This field is composed of unencrypted data.
9. `authentication`: This field carries the authentication code for the data, which can be encrypted according to the indication of `"payload_encryption_flag"`.

The data message submitted for SAC transmission when sending SAC messages should be processed according to the following steps:

1. Check if `'message_counter'` overflows and update `'message_counter'`.
2. Calculate the message authentication code.
3. If encryption is required, concatenate the authentication code and payload to encrypt the message (optional).
4. Construct SAC messages.
5. Transmitting messages.

If any step of the above process fails, the message and status (such as key set, counter, etc.) should be destroyed and an error generated.

The authentication code for SAC data messages is calculated as follows:

```
authentication =  
MAC{SAK}(length(header_hi)||i||header_hi||payload_pi)
```

Where:

1. MAC - The algorithm is identified by `"authentication_cipher_flag"`;
2. SAK - is a 128 bit SAC authentication key;

Authentication code calculation is specific to the entire message, with parameters shown in Figure 29.

Parameter	No. of bits	Mnemonic
length_hi	8	uimsbf
i	32	uimsbf
header_hi	length_hi * 8	bslbf
payload_pi	length_payload*8	bslbf

Figure 29: Parameters in MAC Calculation

1. i: This field is the 'message_counter' in the data message.
2. length_hi: This field is the number of bytes in the message header.
3. header_hi: This field represents the message header.
4. d) payload_pi: This domain carries the message payload. The original unencrypted payload.

If SAC messages need to be encrypted, the following method should be used:

$$\text{encrypted_payload}_i = E\{\text{SEK}, \text{SIV}\}(\text{payload_pi} || \text{authentication_ai})$$

Where:

1. E - The encryption algorithm indicated by encryption_cipher_flag;
2. SEK -128 bit encryption key;
3. SIV - a fixed constant with a value of 0xB27097DEAF305D8A94C871D89525C7A0;
4. authentication_ai - This value is the authentication code MAC.

If the length of payload_pi is not an integer multiple of 128 bits, it should be padded with a "1" first, followed by several "0s" so that the padded result is an integer multiple of 128 bits. If the payload is not encrypted, it will not be filled.

SAC message reception should be processed according to the following steps:

1. Firstly, check if the message contains the correct "message_counter" and "protocol-version";
2. If 'payload_encryption-flag'=1, decrypt the message payload;
3. Calculate the message authentication code and verify the integrity of the message.

If any step of the above process fails, the message and status (such as the key set, counter, etc.) should be destroyed and an error should be generated.

Both the slave server and the master client should maintain a message counter (message_counter) locally to identify the number of messages that have been transmitted. The initial value of this counter is 0x01 (not 0x00), and every time a SAC message is sent or received, the counter value is incremented by 1. In the current counter value state, if a message needs to be sent, assign the current value of the counter to the "message_counter" field in the SAC message being sent. After the message is sent, increment the counter value by 1. In the current counter value state, if a message is received, the value of the "message_counter" field in the received SAC message should be the same as the current value of the counter. After receiving the message, the counter value needs to be incremented by 1. The receiving device should not receive messages with incorrect counter values. Incorrect counter values will result in a 'message sequence error'. When the value of the message sending counter of the slave server overflows (reaching 232), the slave server should stop using the current SAC key and start negotiating a new SAC key.

The decryption process of SAC data messages is as follows:

`payload_pi || authentication_ai = D{ SEK, SIV}(encrypted_payloadi)`

Where:

1. D - the algorithm indicated by "encryption_cipher_flag";
2. SEK -128 bit key;
3. SIV - a fixed constant with a value of 0xB27097DEAF305D8A94C871D89525C7A0;

Among them, 'authentication_ai' should be separated from 'payload_pi'. If 'payload_pi' is not an integer multiple of the 128 bit packet, then the message is obtained by padding a '1' and several '0s'. If the payload is not encrypted, there is no need to fill in any data.

The SAC data message contains an authentication domain, and its verification process is as follows:

```
authentication_ai' =
MAC{SAK}(length(header_hi)||i||header_hi||payload pi)
```

Where:

1. MAC - The algorithm is identified by "authentication_cipher_flag";
2. SAK -128 bit SAC authentication key.

If the calculated 'authentication_ai' is not equal to the 'authentication_ai' exported from the message, the received message must be deleted and a 'message validation error' generated.

Note: The design of SAC channels can be used for various purposes, mainly for negotiating and updating content keys. The way SAC messages are encapsulated into Modbus serial link data unit fields with function code 0 in Figure 30. The packaging steps is given as follows Since SAC is mainly used to synchronize and update CK content key data in this specification, as detailed in the next chapter, one Modbus serial link data unit can be sent completely in one message.

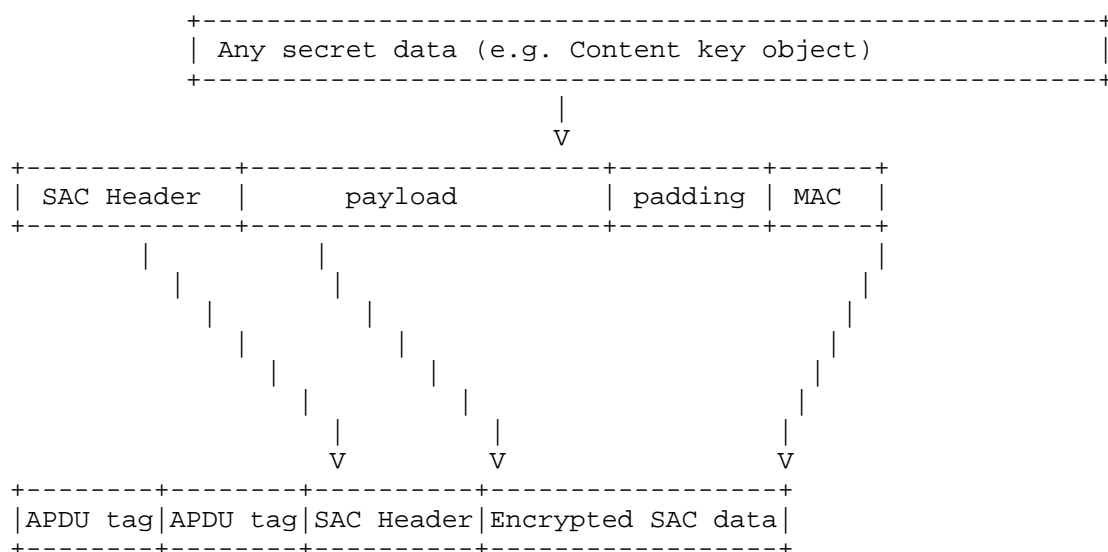


Figure 30: SAC message encapsulation

1. The system collects data objects to form SAC payloads

2. System calculates message authentication code
3. The SAC payload and authentication code are encrypted. Add SAC header, APDU tag, and APDU length field before the encrypted SAC data

5.2.2.4. Content Key Update Mechanism

The process of the content key CK update protocol is shown in Figure 31, and the its description is shown as follows

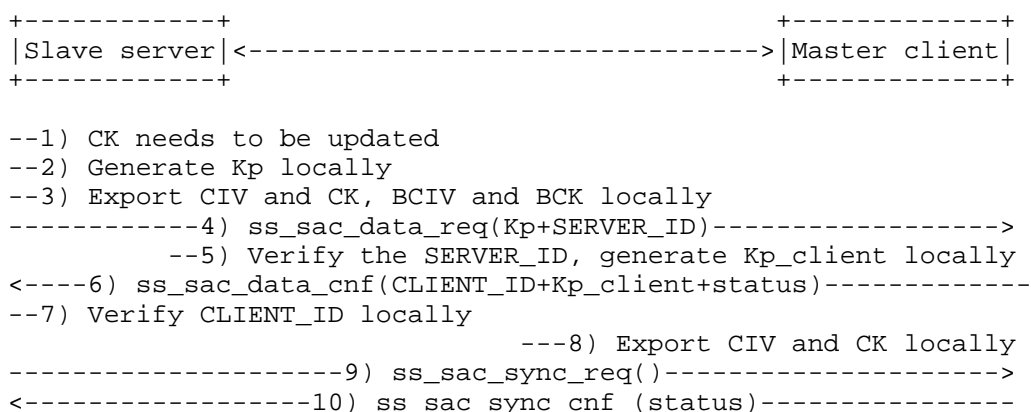


Figure 31: The CK Content Key Update Process

1. When the slave server detects the need for CK update, the slave server begins the CK update process.
2. The slave server generates a 256 bit nonce and calculates Kp=SM3 (nonce).
3. The slave server can immediately start calculating CIV and CK.
4. The slave server sends an ss_sac_data_req APDU containing the following parameters: 1) Kp; 2) SERVER_ID;
5. The master client checks that the received SERVER_ID matches the one in "authentication context". If they are consistent, the master client starts calculating CIV and CK, BCIV and BCK; If there is inconsistency, it will result in a system error. Note: The pre stored SERVER_ID/CLIENT_ID is in the "authentication context".

6. The master client generates a 256 bit nonce and calculates $Kp_client = SM3(\text{nonce})$. Respond to ab ss_sac_data_cnf APDU containing CLIENT_ID and Kp_client. Failure to respond to this message will result in a system error.
7. The slave server checks that the received CLIENT_ID matches the one in "authentication context". If they are consistent, the slave server starts calculating CIV and CK; If there is inconsistency, it will result in a system error. Note: The pre stored SERVER_ID/CLIENT_ID is in the "authentication context".
8. The master client calculates CIV and CK, while the slave server calculates BCIV and BCK.
9. The slave server sends an ss_sac_sync_req APDU to the master client to indicate content key updates. When the slave server completes the initialization of the encryption process, it sends a synchronization request to the master client, informing them that the slave server is ready to use the new CK and BCK.
10. The master client responds with an ss_stac_sync_cnf APDU and confirms to the slave server that the master client is also ready to use the new CK and BCK. Failure to respond to this message will result in a system error.

The update of content key CK is mainly initiated by the slave server, and the master client only passively responds. This process is optional, and the content key CK update protocol is triggered by any of the following conditions:

1. After successful completion of authentication and SAC initialization.
2. Periodic triggering (maximum key lifecycle parameter).
3. The group counter has reached its upper limit.
4. Every time it restarts.

APDU definition used in the content key update protocol process:

- 1) ss_sac_data_req APDU

The slave server uses this APDU to send protocol specific data to the master client, while requesting a response. The syntax of `ss_sac_data_req` APDU is shown in Table 25. Unlike `ss_data_req` APDU, the data contained in this message is authenticated and encrypted. SAC uses the input data in Figure 32 as the payload in the SAC message.

syntax	No. of bits	Mnemonic
<pre>ss_sac_data_req() { ss_sac_data_req_tag length_field()=0 sac_message() }</pre>	24	uimbsbf

Figure 32: `ss_stac_data_req` APDU syntax

`ss_sac_data_req_tag`: 0x9F 90 07.

`sac_message`: The format of this message was defined in the previous section, `payload_encryption-flag` should be 1.

The definition of the payload of SAC messages is in Figure 33.

syntax	No. of bits	Mnemonic
<code>ss_system_id_bitmask</code>	8	bslbf
<code>send_datatype_nbr</code>	8	uimbsbf
for (i=0; i<send_datatype_nbr; i++) {		
<code>datatype_id</code>	8	uimbsbf
<code>datatype_length</code>	16	uimbsbf
<code>data_type</code>	8*datatype_length	bslbf
}		
<code>request_datatype_nbr</code>	8	uimbsbf
for (i=0; i<request_datatype_nbr; i++) {		
<code>datatype_id</code>	8	uimbsbf
}		

Figure 33: `ss_sac_data_req` Payload syntax

`ss_system_id_bitmask`: Same as above.

`send_datatype_nbr`: This message contains the number of data categories.

`datatype_id`: See Appendix A.

`datatype_length`: The byte length of the data in the form of a data type.

`data_type`: The specific data represented by `datatype_id`.

`request_datatype_nbr`: The number of data categories that the primary client should include in its response.

`datatype_id`: See Appendix A.

2) `ss_sac_data_cnf` APDU

When data needs to be authenticated and encrypted, the master client should use this APDU to send protocol specific data to the slave server. The syntax of `ss_sac_data_cnf` APDU is shown in Figure 34. The input data in SAC encapsulation in Figure 35 is used as the payload in the SAC message.

syntax	No. of bits	Mnemonic
<pre>ss_sac_data_cnf() { ss_sac_data_cnf_tag length_field()=0 sac_message() }</pre>	24	uimsbf

Figure 34: `ss_sac_data_cnf` APDU syntax

syntax	No. of bits	Mnemonic
ss_system_id_bitmask	8	bslbf
send_datatype_nbr	8	uimsbf
for (i=0;i<send_datatype_nbr; i++) {		
datatype_id	8	uimsbf
datatype_length	16	uimsbf
data_type	8*datatype_length	bslbf
}		

Figure 35: ss_sac_data_cnf Payload syntax

ss_system_id_bitmask: Same as above.

send_datatype_nbr: This message contains the number of data categories.

datatype_id: See Appendix A.

datatype_length: The byte length of the data in the form of a data type.

data_type: The specific data represented by datatype_id.

3) ss_sac_sync_req APDU

This APDU is used in the CK calculation process. The slave server sends this message to indicate that it has completed the new CK calculation. The syntax of ss_sac_sync_req APDU is shown in Figure 36.

syntax	No. of bits	Mnemonic
ss_sac_sync_req() {		
ss_sac_sync_req_tag	24	uimsbf
length_field()=0		
sac_message()		
}		

Figure 36: ss_stac_sync_req APDU syntax

ss_sac_sync_req_tag: 0x9F 90 09.

sac_message: Same as above.payload_encryption_flag should be 1.The payload of the SAC message here is empty.

4) ss_sac_sync_cnf APDU

This APDU is used in the CK calculation process. The master client responds with this message to the ss_sac_sync_req APDU message sent from the slave server. The syntax of the ss_sac_sync_cnf APDU is shown in Figure 37.

syntax	No. of bits	Mnemonic
ss_sac_sync_cnf() { ss_sac_sync_cnf_tag length_field()=0 sac_message() }	24	uimsbf

Figure 37: ss_sac_sync_cnf APDU syntax

ss_sac_sync_req_tag: 0x9F 90 10.

sac_message: Same as above.payload_encryption_flag should be 1.The payload of this SAC message is a state domain. All possible values in the state domain are listed in Figure 38.

status_field	value
OK	00
No resource	01
Master client busy	02
Not required	03
Reserved	04-FF

Figure 38: State Fields in ss_sac_sync_cnf

When sending SERVER_ID and a random number from the slave server, ss_sac_data_req APDU is sent with following Parameters in Figure 39.

content		
send _datatype_nbr=2		
i	datatype_id	datatype_len
0	2(SERVER_ID)	64 bits
1	7(Kp)	256 bits
request_datatype_nbr=2		
i	datatype_id	
0	1(CLIENT_ID)	
1	20(Status_field)	

Figure 39: ss_sac_data_req APDU content

When the master client responds with CLIENT_ID and status information, ss_sac_data_cnf APDU is sent with following Parameters in Figure 40.

content		
send _datatype_nbr=3		
i	datatype_id	datatype_len
0	2(SERVER_ID)	64 bits
1	7(Kp_client)	256 bits
2	20(Status_field)	8 bits

Figure 40: ss_sac_data_cnf APDU content

The CK update operation on the slave server side is shown as below, and the broadcast content key BCK can be generated along with it.

1. CK update begins
2. Set CK update timer to 0s on the slave server
3. Send Kp

4. Calculate CIV and CK, BCIV and BCK.
5. Receive responses from the master client.
6. Is the initial key lifecycle? If not, goto step 12).
7. If yes, Timer is not more than 9s? If not, goto step 21).
8. If yes, Send sync request
9. sync confirm within the valid time? If yes, goto step 17)
10. Timer is not more than 10s? If not, goto step 21)
11. Wait for 1s, goto step 8).
12. Timer is more than 9s? if not, goto step 12)
13. Send sync request
14. Received sync confirmation? if yes, goto step 17)
15. Counter: $30 > t > 10$? if not, goto step 1)
16. Stop receiving data, goto step 14)
17. Slave server can receive data.
18. Set the block counter to 1.
19. Key lifetime exhausted? if yes, goto step 1)
20. block counter overflow? if yes, goto step 1) else got step 19)

Note: The key update timer is used to record the calculation time of the new CK.

Note: The maximum key lifecycle value is controlled by the administrator configuration system and is not within the scope specified in this specification. The decrease of this value is controlled by the slave server and only decreases when encrypting or decrypting content. For ease of processing, it is generally defined by the number of messages. So when CK is not in use, this value is not updated. Note: The default upper limit of the block counter is 232, and the message may be divided into multiple block frames. The CK block counter limit is the number of processed crypto blocks since CK refresh. The number of block limit defined here is configured by the system and is not within specified in this specification.

Note: The initial key lifecycle refers to the first key lifecycle after SAC initialization.

The CK update operation on the master client side is shown as below, and the broadcast content key BCK can be generated along with it.

1. Waiting for the key update from the slave server.
2. Receive K_p .
3. Confirm.
4. Calculate CIV and CK, BCK and CIV.
5. Receive sync request.
6. Confirm the sync request and start using the new CK.

The calculation process of content key CK and broadcast content key BCK should be completed as follows:

The encryption algorithm requires CK and CIV, and the calculation of these two values requires two steps: key seed calculation and CK and CIV export.

Step 1: Calculate the key seed.

$K_p = \text{SM3}(\text{nonce})$

Step 2: Export CK and CIV.

$\text{CK} = \text{MSB128}(\text{SM3}(K_p || \text{SERVER_ID}))$, $\text{CIV} = \text{LSB128}(\text{SM3}(K_p || \text{SERVER_ID}))$

Step 3: Export BCK and CIV for broadcast communication between the master client and all slave servers.

$\text{BCK} = \text{MSB128}(\text{SM3}(K_p_client || \text{CLIENT_ID}))$, $\text{BCIV} = \text{LSB128}(\text{SM3}(K_p_client || \text{CLIENT_ID}))$

Where:

SERVER_ID represents the ID of the slave server, while CLIENT_ID represents the ID of the master client. CIV remains unchanged throughout each key lifecycle. CIV and CK change simultaneously. The current CIV is reused at the beginning of encrypting (decrypting) each Modbus data frame. During broadcast communication, BCK and BCIV should be changed simultaneously. Generally, after power on, the BCK and BCIV of all slave servers can remain unchanged, or they can be

updated by the master station through the SAC content key update protocol. The process of updating BCK protocol is shown as follows separately.

1. The master client generates a 256 bit nonce and calculates $Kp_client = SM3(\text{nonce})$.
2. The master client can immediately start calculating BCIV and BCK..
3. The master client sends an `ss_sac_data_req` APDU containing the following parameters to the slave server: 1) `Kp_client`; 2) `CLIENT_ID`;
4. The slave server checks the consistency between the received `CLIENT_ID` and the "authentication context". If they are consistent, calculate BCIV and BCK from the slave server; If there is inconsistency, it will result in a system error. Note: The pre-stored `SERVER_ID/CLIENT_ID` is in the "authentication context".
5. The slave server responds with an `ss_sac_data_cnf` APDU containing the `SERVER_ID`. Failure to respond to messages will result in a system error.
6. The master client checks that the received `SERVER_ID` matches the 'authentication context'. If the `SERVER_ID` returned by the slave server is incorrect, a system error will be returned.
7. The master client sends an `ss_sac_sync_deq` APDU to indicate BCK updates. When the slave server completes the initialization of the encryption process, it sends a synchronization request to the master client, informing them that the slave server is ready to use the new CK and BCK.
8. The slave server responds with the `ss_sac_sync_cnf` APDU and confirms to the slave server that the master client is also ready to use the new CK and BCK. Failure to respond to this message will result in a system error.

The APDU format of the protocol that includes specific data types during the separate update process of BCK broadcast content keys is shown in Figure 41 below:

When the master client sends a `CLIENT_ID` and a random number to the slave server, `ss_sac_data_req` APDU is sent with following Parameters in Figure 41.

content		
send _datatype_nbr=2		
i	datatype_id	datatype_len
0	2(CLIENT_ID)	64 bits
1	7(Kp_client)	256 bits
request_datatype_nbr=2		
i	datatype_id	
0	1(SERVER_ID)	
1	20(Status_field)	

Figure 41: ss_sac_data_req APDU content

When the slave server responds with the SERVER_ID and status information, ss_sac_data_cnf APDU is sent with following Parameters in Figure 42.

content		
send _datatype_nbr=2		
i	datatype_id	datatype_len
0	1(SERVER_ID)	64 bits
1	20(Status_field)	8 bits

Figure 42: ss_sac_data_cnf APDU content

Note: During the content key update phase, APDU messages will encapsulate the Modbus serial link data unit field with function code 0, which is much smaller than 252 bytes (the maximum data field length of a Modbus RTU frame) of APDU. One Modbus serial link message frame can complete the interaction of APDU.

5.2.2.5. Content Encryption and Decryption Mechanism

After the authentication channel is established and the content keys BCK and CK are synchronized, when transmitting the encrypted Modbus serial link protocol, the encrypted content includes the original Modbus serial link protocol function code and data fields. The encryption mechanism for unicast and broadcast is the same. Taking unicast communication as an example, the content encryption and decryption methods are GCM Mode of symmetric cryptography algorithm such as AES GCM and SM4 GCM. The content encryption method has the following input parameters: CIV, CK, Mode, Modbus serial link protocol function code and data. Among them, CIV and CK are secret data shared by the master client and slave server after executing the content key update protocol. CIV is the CTR in the algorithm, CK is the content key, and Mode is the encryption and decryption mode agreed upon by both parties during the first binding. Currently, only AES GCM and SM4 GCM modes are supported (mode=aes_gcm or sm4_gcm, indicated by the "encryption_capability" field in the device's public key certificate). The plaintext groups P1, P2, P3, P4, ..., Pn are 128 bit slice groups of data from the function and data fields in the original Modbus PDU, and those that are less than 128 bits long are filled with 0. 128 bit integer, with additional message taken from the first 128 bits of SM3 ("Modbus"). during decryption, C1, C2, C3, C4, ..., Cn represent ciphertext, and the additional information takes the first 128 bits of SM3 ("Modbus"). The receiver should first calculate and generate a MAC' using a similar encryption mechanism, and then compare it with the received MAC value to see if they are same. If they are the same, the plaintext of the content should be decrypted. Otherwise, it should be considered that there is an abnormality in the current serial communication bus network, such as the possibility of a man in the middle attack.

The encrypted Modbus security protocol data field content should be transmitted in the following APDU format, with the encapsulation method shown in Figure 43. The implementation principle of ASCII transmission mode is similar, and not repeated here:

The data content of Modbus security protocol is sent through ss_data_snd APDU, and the syntax of ss_data_snd APDU is shown in Figure 44.

5.2.3.1. Key hierarchy structure

The password or pre-shared key storage hierarchy should include protocol public parameters, pre shared passwords or keys for the master client and slave server. The key data at this level is mainly used to execute authentication protocols. The pre-shared keys after device initialization cannot be changed or replaced, nor can they be obtained by unauthorized users. The definition of protection methods for these data is not within the scope of this specification.

At the authentication level, the master client and slave server mainly verify each other's identity through cryptographic protocols based on shared passwords or keys, and generate temporary content keys CK and an initial vector CIV for use by the content encryption layer. For the generated intermediate keys or state values, there should be a trusted execution environment to ensure security. The protection methods for these intermediate keys or state values are not within the scope of this specification. In password or pre-shared key based schemes, each party in the master-slave device combines its own password or pre shared key based value and private key with the other party's public key to obtain one or more identical keys. Other (public or private) information known to both the master and slave devices can also be used as key derivation parameters to participate in content key generation. If all parties use the same password or pre shared key and the same key derivation parameters, the protocol will ensure that all parties will receive the same content key, allowing both parties to export the shared key with only the password or pre shared key. It should be ensured that without using the password or pre shared key, neither party can successfully participate in the protocol and obtain the same content key.

The content encryption layer uses the content key CK and an initial vector CIV to encrypt and decrypt the media stream using AES/SM4. For the data encryption and decryption process of Modbus serial link communication protocol, there should be a trusted execution environment or hardware to ensure security. The definition of specific protection methods is not within the scope of this specification.

5.2.3.2. Authentication and Key Agreement Mechanism

This specification is based on the Modbus serial link security protocol using passwords or pre shared keys, and is based on the SM2 elliptic curve public key cryptography algorithm Part 3 Key Exchange Protocol standard developed by ISO/IEC 14888-3:2018 and China National Cryptography Administration. In this specification, the private key of the master and slave devices should be derived from a shared password or key. Assuming that the password or key shared by

the master/slave devices is represented as PW, the temporary private key of the device must be generated as follows:

The master/slave devices first respectively select a random number r and calculate the 32-bit byte string $PW_r = \text{SM3}(PW || r)$. By converting it into an element under the field F_q of ECC, the private key d_A is obtained, and the public key $P_A = [d_A] G = (x_A, y_A)$ is calculated. Assuming that the public and private key pairs of the master/slave devices are d_H/P_H , d_M/P_M , respectively.

The authentication and content key negotiation mechanism is shown as below. For master client, the raw data generated and configured by the host client are (ECC system parameters, Z_H , Z_M , d_H , P_H , P_M) and a 128bit random number r_{nonce} is selected to generate the broadcast key and encryption/decryption mode.

1. Generate random number $r_H \in [1, n-1]$.
2. Calculate $R_H = [r_H] G = (x_1, y_1)$.
3. Send R_H , mode to the slave server.
4. Calculate $X_1 = 2^w + (X_1 \text{ AND } (2^w - 1))$.
5. Calculate $t_H = (d_H + X_1 * r_H) \text{ Mod } n$.
6. Receive R_M , S_M from the slave server, does R_M satisfy the curve equation? If not, Content key negotiation failed.
7. Calculate $X_2 = 2^w + (x_2 \text{ AND } (2^w - 1))$.
8. Calculate ECC point $U = [h * t_H](P_M + [X_2]R_M) = (x_u, y_u)$.
9. $U = O$? If yes, Content key negotiation failed.
10. Calculate $K_H = \text{KDF}(PW || x_u || y_u || Z_H || Z_M, 256)$.
11. Calculate $S_1 = \text{SM3}(0x02 || PW || y_u || \text{Hash}(x_u || Z_H || Z_M || x_1 || y_1 || x_2 || y_2 || r_B))$.
12. $S_1 = S_M$? If not, Content key negotiation failed.
13. Calculate $S_H = \text{SM3}(0x03 || PW || y_u || \text{Hash}(x_u || Z_H || Z_M || x_1 || y_1 || x_2 || y_2 || r_B))$, $r_B = \text{MSB128}(K_H)r_{\text{nonce}}$, Sends S_H , r_B to Slave server.
14. Calculate $CK = \text{MSB128}(K_H)$, $CIV = \text{LSB128}(K_H)$.

15. Calculate $BCK = MSB_{128}(SM3(r_nonce || PW))$,
 $BCIV = LSB_{128}(SM3(r_nonce || PW))$.

For slave server, the raw data generated and configured are (elliptic curve system parameters, Z_H , Z_M , d_M , P_M , P_H).

1. Generate random number $r_M \in [1, n-1]$.
2. Calculate $R_M = [r_M] G = (x_2, y_2)$.
3. Take $X_2 = 2^w + (x_2 \text{ AND } (2^w - 1))$.
4. Calculate $t_M = (d_M + X_2 * r_M) \text{ mod } n$.
5. Receive R_H , mode from the slave server, Does R_H satisfy the curve equation? If not, Content key negotiation failed.
6. Calculate $X_1 = 2^w + (x_1 \text{ AND } (2^w - 1))$.
7. Calculate ECC point $V = [h * t_M](P_H + [X_1]R_H) = (x_v, y_v)$.
8. $V = O$? If yes, Content key negotiation failed.
9. Calculate $K_M = KDF(PW || x_v || y_v || Z_H || Z_M, 256)$.
10. Calculate $S_M = SM3(0x02 || PW || y_v || Hash(x_v || Z_H || Z_M || x_1 || y_1 || x_2 || y_2 || r_B))$.
11. Sends R_M , S_B to Master client.
12. Calculate $S_2 = SM3(0x03 || PW || y_v || Hash(x_v || Z_H || Z_M || x_1 || y_1 || x_2 || y_2 || r_B))$.
13. Receive S_H , r_B from the master client, $S_2 = S_H$? If not, Content key negotiation failed.
14. The CK confirmation from the Master client to the slave server has been successful, Calculate $CK = MSB_{128}(K_M)$, $CIV = LSB_{128}(K_M)$,
 $r_nonce = MSB_{128}(K_M)r_B$, $BCK = MSB_{128}(SM3(r_nonce || PW))$,
 $BCIV = LSB_{128}(SM3(r_nonce || PW))$.

The encryption algorithm mode is determined by an 8-bit mode and currently only supports AES GCM mode `aes_gcm` or SM4 GCM mode `sm4_gcm`.

The steps of Modbus serial link security protocol based on password or pre-shared key are shown in Figure 45 and described as below:

```

+-----+                                     +-----+
|Slave server|<----->|Master client|
+-----+                                     +-----+

-----1) ss_open_req()----->
<-----2) ss_open_cnf()-----
-----3) ss_data_req(R_H+mode)----->
<-----4) ss_data_cnf(R_M+S_M)-----
-----5) cc_data_req(S_H+r_B)----->
<-----6) ss_data_cnf()-----

```

Figure 45: Content key negotiation message exchange based on password or pre-shared key

1. The master client sends the ss_sk_open_req APDU to the slave server.
2. The slave server responds with ss_sk_open_cnf APDU.
3. The master client sends the ss_sk_data_req APDU to the slave server. The message contains: 1) The master client selects the ECC point (R_H) as needed; 2) Encryption and decryption mode; 3) The ID of the requested data type.
4. The slave server responds with ss_sk_data_cnf APDU. The message contains: 1) The slave server selects ECC points (R_M) as needed; 2) Message authentication code (S_M) of the slave server; Failure to respond to this message may result in a system error, possibly due to the master client being unable to verify data from the slave server.
5. The master client then sends the ss_sk_data_req APDU. The message contains: 1) The message authentication code (S_H) of the master client; 2) Random number (r_B) used to generate broadcast content keys; Failure to respond to this message may result in a system error, possibly due to the slave server being unable to verify data from the master client.
6. The slave server responds with ss_sk_data_cnf APDU. The inability to respond to this message may result in a system error, possibly due to the inability of the master client to verify data from the slave server.

The format definitions of ss_sk_open_req, ss_sk_open_cnf, ss_sk_data_req, and ss_sk_data_cnf APDU are the same as those of the previous ss_open_req, ss_open_cnf, ss_data_req, and ss_data_cnf APDU, respectively. But the tag types are different to distinguish different serial link content encryption mechanisms.

ss_sk_open_req(tag=0x9F 90 12), ss_sk_open_cnf(tag=0x9F 90 13), ss_sk_data_req(tag=0x9F 90 14), ss_sk_data_cnf(tag=0x9F 90 15), The APDU format for protocols based on passwords or pre-shared keys that include specific data types is shown as below:

When the master client sends a ECC random point, a random number, and encryption/decryption mode to the slave server, ss_sk_data_req APDU is sent with following Parameters in Figure 46.

content		
send_datatype_nbr=2		
i	datatype_id	datatype_len
0	22(R_H)	512 bits
1	24(mode)	8 bits
request_datatype_nbr=2		
i	datatype_id	
0	22(R_M)	
1	25(S_M)	

Figure 46: ss_sk_data_req APDU content

When sending response ECC random point and authentication code from the slave server, ss_sk_data_cnf APDU is sent with following Parameters in Figure 47.

content		
send_datatype_nbr=2		
i	datatype_id	datatype_len
0	22(R_M)	512 bits
1	25(S_M)	256 bits

Figure 47: ss_sk_data_cnf APDU content

When the master client sends a message authentication code,
 ss_sk_data_req APDU is sent with following Parameters in Figure 48.

content		
send _datatype_nbr=2		
i	datatype_id	datatype_len
0	9(S_H)	256 bits
1	23(r_B)	256 bits

Figure 48: ss_sk_data_req APDU content

5.2.3.3. Content Encryption and Decryption Mechanism

The content encryption and decryption mechanism used in this section is the same as that in previous section.

6. Appendix A

Parameter datatype id table exchanged in APDU. Table A.1 summarizes the parameters and data type ID number used in this draft recommendation.

Table A.1: System parameter numbers in secure communication protocol calculation

Keywords or variables	No, of bits	datatype id
CLIENT_ID	64	1
SERVER_ID	64	2
Client_Brand_Cert	variable	3
Server_Brand_Cert	variable	4
Reserved	--	5
Reserved	--	6
Kp	256	7
DHPH	512	8
DHPM	512	9

Client_dev_Cert	variable	10
Server_dev_Cert	variable	11
RM(Signature_A)	256	12
RH(Signature_B)	256	13
auth_nonce	256	14
Ns_H	64	15
Ns_M	64	16
AKH	256	17
AKM	256	18
Reserved	-	19
status_field	8	20
Reserved	-	21
R_H/R_M	512	22
r_B	256	23
mode	8	24
S_M/S_H	256	25

7. Appendix B

Pseudo-random number generator. The random numbers generated by the pseudo-random number generator PRNG in this draft recommendation are shown in Table B.1.

Table B.1: Parameters for Using Pseudo Random Number Generator

random number	No, of bits	description
DHX	256	DH exponent "x"
DHY	256	DH exponent "y"
Kp	64	Used to establish SAC
Ns_H	64	Used to establish SAC
Ns_M	64	Used to establish SAC
r_B	256	Used to generate BCK

This draft recommendation requires the use of FIPS-140-2 and a pseudo-random number generator defined by [SCTE 41:2004], section 4.6 and An AES based pseudo-random number generator by ANSI X 9.31 1316H [ANSI X 9.31:1998].

8. Appendix C

Trust Model and Certificate Format

1) Trust Model:

The Modbus serial link communication security protocol based on public key certificates adopts a tree shaped PKI structure, as shown in Figure C.1. The system has only one root of trust (ROT) as trust anchor, which issues public key certificates to industrial equipment manufacturers of different brands, and can also issue public key certificates to different business operators (factories). In addition, ROT also issues equipment certificates for different brands of industrial equipment manufacturers for their respective equipment brands. In actual operation, these brand certificates and equipment certificates can be issued by the original factory in the product at the time of leaving the factory, or they can be deployed and operated in the product by third-party agents or the most network user factory after leaving the factory.

Trust Model:

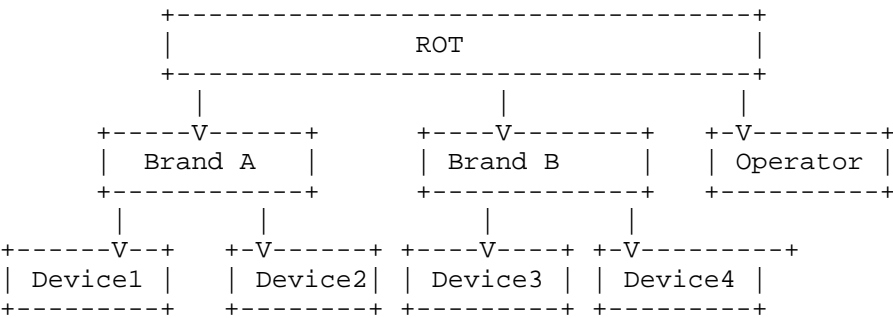


Figure C.1: Trust Model

1) Certificate format and content:

The Modbus serial link communication security protocol based on public key certificates has three types of public key certificates: ROT root certificate, manufacturer brand certificate, and device certificate. Each device stores the root certificate of ROT, device certificate, and the corresponding manufacturer’s brand certificate in its NVM. The root certificate is a self-signed certificate used to verify the manufacturer’s brand and device certificate. The manufacturer’s brand certificate is issued by ROT and is used to verify the equipment certificate. These three types of certificates have a unified field format, following the X.509 format. The basic and extended fields they contain are shown in Table C.1. They are distinguished by the "Certificate Type" field in the certificate.

Table C.1: Certificate Fields and Structure

syntax	No. of bits	Mnemonic
Modbus_certificate(){		
modbussec_ID	8	bslbf
certificate_type	8	bslbf
certificate_version	8	bslbf
encryption_capability	8	bslbf
holder_ID	64	bslbf
holder_public_key_x	256	uimsbf
holder_public_key_y	256	uimsbf
issuer_ID	64	bslbf
signed_date	32	bslbf
reserved	64	bslbf
signature_c	256	uimsbf
signature_d	256	uimsbf
}		

Modbussec-ID: This field represents the security system to which the certificate belongs, and this section specifies its value as 0xC2.

Certificate type: This field represents the type of certificate. See Table C.2.

Table C.2: Certificate Types

Certificate_type	The type of certificate represented
0x00	Root certificate
0x01	Operator certificate
0x02	Brand certificate
0x03	Device certificate
Others	Reserved

Certificate_version: This field represents the version of the certificate, and this section specifies that the version number for all types of certificates is 0x01.

Encryption_capability: This field represents the encryption modes supported by the device, and AES algorithm is used by default, as shown in Table C.3.

Table C.3: Encryption and Decryption Capability

encryption_capability	description
0x00	Reserved
0x01	For device supporting aes_gcm
0x02	For device supporting aes_gcm and sm4_gcm
Others	Reserved

Holder_ID: This field represents the ID of the certificate holder, which is SERVHER-ID or CLIENT-ID.

Holder_public_key_x: This field represents the horizontal axis value of the certificate holder's public key.

Holder_public_key_y: This field represents the y-axis value of the certificate holder's public key.

Issue_ID: This field represents the ID of the certificate issuer, which is the holder ID in the certificate issued by the CA that issued the certificate.

Signed_time: This field represents the date of issuance of the certificate. This time is composed of the lower 16 bits of MJD time and four 4-bit BCD digits. For example, on October 13, 1993, 12:45 was encoded as "0xC0791245".

Reserved: Set all fields to zero for future expansion.

Signature_c: This field represents the c value of the certificate issuer's digital signature on the certificate data.

Signature-d: This field represents the d value of the digital signature of the certificate issuer on the certificate data.

Note: The certificate signature scheme adopts ECSSA scheme in 10.2 of IEEE 1363-2000 standard, the signature algorithm adopts ECSP-DSA algorithm in 7.2.7 of IEEE 1363-2000 standard, the message encoding method adopts EMSA1 method in 12.1.1 of IEEE 1363-2000 standard, and the hash function in the encoding method adopts SHA-256. Alternatively, the SM2 and SM3 algorithms in the ISO/IEC 14888-3:2018 standard can be used.

Note: The verification algorithm corresponding to the above signature algorithm is the ECV-DSA algorithm or SM2 and SM3 algorithms in 7.2.8 of the IEEE 1363-2000 standard.

3) Certificate holder ID

In order to support different revocation granularity requirements in the future, when the certificate type is "device certificate", further definition of the holder_ID domain is required, as shown in Figure C.2.

1. Define the top 16 bits in the "holder-ID" field (64 bits) as the "device brand identifier";
2. Define the lower 32 bits of the "holder-ID" field (64 bits) as the "device serial number";
3. Define the middle 16 bits of the "holder-ID" field (64 bits) as the "device model identifier".

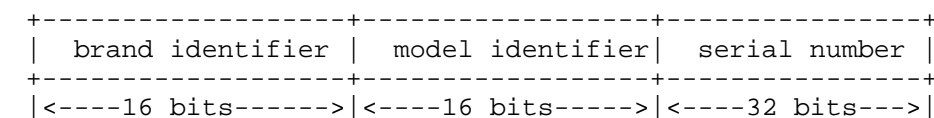


Figure C.2: Structure of Certificate Holder ID

In addition, this section requires that when the certificate type is "device certificate", the values of "device brand identifier" and "device model identifier" cannot be zero; The "device brand identifier" and "device model identifier" in the IDs of the other types of certificates are both set to zero values.

4) Elliptic curve parameters

The Modbus serial link communication security protocol based on public key certificates in this document uses the elliptic curve defined in the prime field according to the SM2 standard, where:

```
EC_p=0x8542D69E 4C044F18 E8B92435 BF6FF7DE 45728391 5C45517D 722EDB8B
08F1DFC3 EC_a=0x787968B4 FA32C3FD 2417842E 73BBFEFF 2F3C848B 6831D7E0
EC65228B 3937E498 EC_b=0x63E4C6D3 B23B0C84 9CF84241 484BFE48 F61D59A5
B16BA06E 6E12D1DA 27C5249A
```

The order of the above curve is:

```
EC_N=0x8542D69E 4C044F18 E8B92435 BF6FF7DD 29772063 0485628D 5AE74EE7
C32E79B7
```

The coordinates of the base point $EC_G=(G_x, G_y)$ on the curve:

```
Gx=0x421DEBD6 1B62EAB6 746434EB C3CC315E 32220B3B ADD50BDC 4C4E6C14
7FEDD43D Gy=0x0680512B CBB42C07 D47349D2 153B70C4 E5D7FDFC BFA36EA1
A85841B9 E46E09A2
```

9. IANA Considerations

This memo includes no request to IANA.

10. Security Considerations

This specification defines the reference native security protocol specification for the case with pure serial link communication, which (e.g., RS485-based Modbus) remains an unresolved challenge. Serial Modbus transmits data in plaintext, leaving it susceptible to interception, modification, and hardware-based side-channel attacks. There are currently no formal standards addressing security for Modbus over RS485, which is still widely used due to its simplicity and long-distance capabilities. Currently, for any EIA/TIA-485 multi-point system, whether it is a 2-wire or 4-wire configuration, the maximum length for cables with a maximum baud rate of 9600bit/s and AWG26 (or thicker) specifications can reach over 1000m. For RS485 Modbus, a sufficiently wide cable diameter should allow for a maximum length of over 1000m, and for RS485 Modbus using Category 5 cables, the maximum length can reach 600m. This lack of encryption

and authentication mechanisms poses risks, particularly in scenarios involving relays or bridged networks. While Modbus TCP has seen notable advancements in security through TLS-based solutions, serial Modbus communication requires urgent attention. Introducing lightweight encryption and authentication mechanisms for serial links could provide a practical way to enhance security and protect legacy systems without significant infrastructure changes. Therefore, it is urgent to improve and enhance the communication security of Modbus protocol under serial link mode, for reference by relevant institutions and organizations in various industries, in order to ensure the practical application security of various industrial control systems. With the introduction of the Modbus serial link security standard, the introduction of encryption and authentication mechanisms in the serial link communication channel of the Modbus protocol significantly improves its security, providing a relatively simple and direct upgrade path for existing devices that use Modbus extensively.

11. References

11.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

11.2. Informative References

- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/rfc/rfc5280>>.
- [IEC62443-3-3] "System Security Requirements and Security Levels", IEC 62443, 2013.
- [IEC62443-4-2] "Technical Security Requirements for IACS Components", IEC 62443, 2019.

[IEC62351-3]

"Power Systems Management and Associated Information
Exchange Data and Communications Security Part 3:
Communication Network and System Security for Profiles
Including TCP/IP", IEC 62351, 2014.

[IEC62351-4]

"Power Systems Management and Associated Information
Exchange Data and Communications Security Part 4:
Profiles Including MMS and Derivatives", IEC 62351, 2018.

[NISTSP800-82]

"Guide to Industrial Control Systems (ICS) Security",
NIST SP800, 2018.

[ENISA-ICS-Security-Guidelines]

"Good Practices for Security of Industrial Control
Systems", ENISA ICS-Security-Guidelines, 2018.

Authors' Addresses

Penghui Liu (editor)
Pengcheng Laboratory
No.2 Xingke 1 Street
Shenzhen
518055
China
Email: liuph@pcl.ac.cn

Rongwei Yang (editor)
Pengcheng Laboratory
No.2 Xingke 1 Street
Shenzhen
518055
China
Email: yangrw@pcl.ac.cn

Meiling Chen (editor)
China Mobile
No.32 Xuanwumen West Street
Beijing
100000
China
Email: chenmeiling@chinamobile.com

Yu Fu (editor)
China Unicom
No.1 Zhonghe Street
Beijing
100000
China
Email: fuy186@chinaunicom.cn

Weizhe Zhang (editor)
Pengcheng Laboratory
No.2 Xingke 1 Street
Shenzhen
518055
China
Email: zhangwzh@pcl.ac.cn