Electronic Supplementary Material 1 (ESM1)

Cryptographic Scripting Pseudocode and Implementation Details for "Machine Learning–Based Cryptographic Selection System for IoT Devices Based on Computational Resources"

Authors: Qi Ming Ng, Julia Juremi and Nor Azlina Abd Rahman
Version: v0.1 (27 October 2025)

Purpose Note:

This supplemental material includes pseudocode, implementation details, parameter configurations, and the corresponding Python source files for all symmetric and hybrid cryptographic algorithms addressed in the main manuscript. This document is for review purposes and will be structured in accordance with the criteria of the Springer journal ESM upon acceptance.

Contents Summary

File	Description
ESM1.docx	Main supplementary document containing pseudocode, implementation details, and supporting explanations
project_root.zip	Contains the Python sources for the IoT crypto reference modules and required local cipher submodules. Core modules (under Data_Simulation/) are: encry_decry_sym.py (symmetric AEAD/LWC ciphers), encry_decry_hybrid.py (ECC + HKDF + symmetric hybrids), and consolidate.py (uniform encrypt/decrypt API). External cipher submodules are under Cipher_GitHub/Simon_Speck_Ciphers/Python/simonspeckciphers/ (Simon/Speck) and Cipher_GitHub/present/present_python/ (Present). Environment files in Data_Simulation/ include generate_env_info_auto.py (environment descriptor module) and its output appendix_environment_concise.txt, plus requirements.txt (pinned packages)
README.txt	Guide for scripts execution and interpretation

Reader Guide

This document contains organized pseudocode listings (Algorithms A.1–A.37), parameter configurations, and accompanying explanatory notes related to the implementation of symmetric and hybrid cryptographic scripting, which underpins the research discussed in Section 4 of the manuscript. The pseudocode prioritizes clarity and reproducibility over precise runtime equivalency. The accompanying Python scripts in the zip file offer the reference implementation utilized for executing cryptographic procedures, including encryption and decryption, as detailed in the manuscript. It is recommended to utilize the accompanying requirements.txt file or adhere to the execution environment outlined in the appendix for establishing the necessary environment before executing the attached Python scripts. The enclosed README.txt provides additional information and comprehensive instructions for utilizing the Python scripts.

Table of Contents

Section 0 – Algorithm Relationships and Workflow Overview

This section clarifies the relationship between Algorithms A.1 and A.37 utilized in the symmetric-cipher environment setup, helper-function definition, API dispatch, symmetric cryptographic operations, ECC module generation, and hybrid cryptographic scripting workflow. The section is segmented into three subsections: the first presents a narrative dependency mapping, the second features a workflow diagram illustrating the complete pipeline, and the final section contains a dependency mapping table for rapid reference.

Section 0.1 – Algorithm Dependency Mapping

This section shows how the cryptographic scripting workflow described in this ESM proceeds through the following algorithmic stages:

- **Algorithm A.1** initializes all required symmetric-cipher modules, including ChaCha20-Poly1305, Ascon-128/128a, Speck, Simon, Present, and TWINE, as well as conversion helpers, randomness utilities, and block/stream cipher back-ends.
- **Algorithm A.2** initializes ECC modules, including P-256 key-pair generation, ECDH key exchange, and HKDF-SHA256, and sets up hybrid encryption dependencies used in A.24–A.37.
- **Algorithm A.3** configures the unified encryption/decryption API, which dispatches calls to either symmetric-only or hybrid-cipher routines.
- **Algorithm A.4** executes a standardized encryption workflow by mapping the selected algorithm to the appropriate symmetric or hybrid routine.
- **Algorithm A.5** manages decryption in the same unified manner, calling the correct symmetric or hybrid decryption script.
- **Algorithm A.6** serves as a utility converting byte-length parameters to bit-length values for algorithms requiring bit-oriented inputs.
- **Algorithm A.7** generates ASCII-formatted keys for TWINE-128 encryption and decryption, used in A.20–A.21 and hybrid versions A.36–A.37.
- **Algorithms A.8–A.21** implement symmetric encryption and decryption routines, including ChaCha20-Poly1305, Ascon-128/128a, Speck-128, Simon-128, Present-128, and TWINE-128.
- **Algorithm A.22** generates ECC P-256 key pairs and raw public/private key byte encodings.
- **Algorithm A.23** performs ECC key exchange using ECDH and derives session keys via HKDF-SHA256 for hybrid cryptography.
- **Algorithms A.24–A.37** implement hybrid cryptographic operations by combining ECC key establishment with each symmetric algorithm from A.8–A.21, producing consolidated hybrid encryption and decryption routines.

Section 0.2 – Workflow Diagram for Algorithm A.1-A.37

This section presents the sequential cryptographic-environment initialization, API dispatching, symmetric and hybrid encryption/decryption scripting workflow executed by Algorithms A.1 through A.37. The complete pipeline is illustrated in Fig. 1.

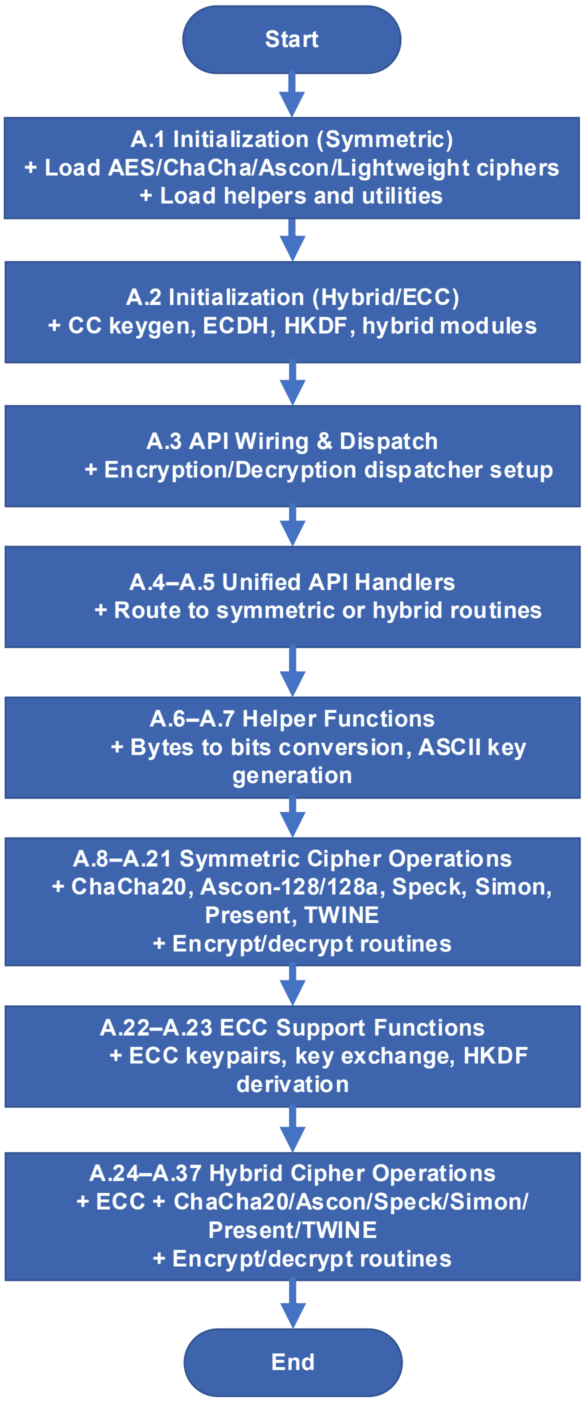
Start
A.1 Initialization (Symmetric)
+ Load AES/ChaCha/Ascon/Lightweight ciphers
+ Load helpers and utilities
A.2 Initialization (Hybrid/ECC)
+ CC keygen, ECDH, HKDF, hybrid modules
A.3 API Wiring & Dispatch
+ Encryption/Decryption dispatcher setup
A.4–A.5 Unified API Handlers
+ Route to symmetric or hybrid routines
A.6–A.7 Helper Functions
+ Bytes to bits conversion, ASCII key generation
A.8–A.21 Symmetric Cipher Operations
+ ChaCha20, Ascon-128/128a, Speck, Simon, Present, TWINE
+ Encrypt/decrypt routines
A.22–A.23 ECC Support Functions
+ ECC keypairs, key exchange, HKDF derivation
A.24–A.37 Hybrid Cipher Operations
+ ECC + ChaCha20/Ascon/Speck/Simon/Present/TWINE
+ Encrypt/decrypt routines
End

Fig. 1 Workflow diagram for the cryptographic simulation, host-measurement, IoT scaling, feasibility evaluation, and results-export pipeline (Algorithms A.1–A.37)

Section 0.3 – Algorithm Dependency Table

This section summarizes the dependencies among the algorithms and outlines their primary roles within the symmetric-cipher, ECC, and hybrid-cipher scripting workflow as shown in Table 1.

Table 1 Algorithm dependencies and functional roles

Algorithm	Depends On	Provides Output For	Functional Role
A.1	—	A.3–A.21	Initialize symmetric ciphers, utilities, AEAD/LWC modules
A.2	—	A.3, A.22–A.37	Initialize ECC, HKDF, hybrid encryption infrastructure
A.3	A.1, A.2	A.4–A.5	Unified encryption/decryption dispatch API
A.4	A.3, A.8–A.37	—	Standardized encryption dispatcher
A.5	A.3, A.8–A.37	—	Standardized decryption dispatcher
A.6	—	A.8–A.37	Bytes→bits conversion helper
A.7	—	A.20–A.21, A.36–A.37	ASCII key generator for TWINE
A.8–A.9	A.1, A.6	A.4–A.5	ChaCha20-Poly1305 symmetric operations
A.10–A.13	A.1, A.6	A.4–A.5	Ascon-128 and Ascon-128a routines
A.14–A.15	A.1, A.6	A.4–A.5	Speck-128 routines
A.16–A.17	A.1, A.6	A.4–A.5	Simon-128 routines
A.18–A.19	A.1, A.6	A.4–A.5	Present-128 routines
A.20–A.21	A.1, A.6, A.7	A.4–A.5	TWINE-128 routines
A.22	A.2	A.23, A.24–A.37	ECC key-pair generation
A.23	A.22, A.2	A.24–A.37	ECC key exchange + HKDF session key
A.24–A.25	A.22–A.23, A.8–A.9	A.4–A.5	Hybrid ChaCha20
A.26–A.29	A.22–A.23, A.10–A.13	A.4–A.5	Hybrid Ascon
A.30–A.31	A.22–A.23, A.14–A.15	A.4–A.5	Hybrid Speck
A.32–A.33	A.22–A.23, A.16–A.17	A.4–A.5	Hybrid Simon
A.34–A.35	A.22–A.23, A.18–A.19	A.4–A.5	Hybrid Present
A.36–A.37	A.22–A.23, A.20–A.21	A.4–A.5	Hybrid TWINE

Section 1 – System Initialization

This section specifies the necessary libraries and dependencies required to provide the cryptographic environment for simulating encryption and decryption processes of symmetric and hybrid encryption algorithms. Algorithms A.1 and A.2 outline the libraries and dependencies required for symmetric and hybrid cryptography, respectively, and these are located at the beginning of the modules titled "encry_decry_sym.py" and "encry_decry_hybrid.py". All library dependencies and version specifications utilized in this work are documented in the supplementary ESM files—requirements.txt and appendix_environment_concise.txt—with an accompanying human-readable summary presented in Appendix A.

```
1: Inputs:
      —

2: Output:
      Symmetric cipher modules loaded; helper utilities available

3: Procedure LoadSymmetricDependencies()
4:       Import cryptographic primitives for block and stream cipher operations
5:       Enable backend provider required for cipher execution
6:       Load AEAD cipher modules including ChaCha20-Poly1305
7:       Activate Ascon-128/128a sponge-based AEAD routines
8:       Load lightweight block-cipher modules (Speck, Simon, Present, Twine)
9:       Import utility modules for randomness, timestamps, and OS operations
10:      Register helper conversion procedure for bytes-to-bits
11:      Define symmetric encryption and decryption interfaces
12:      Return handles to all loaded modules
13: endProcedure
```

Algorithm A.1 LoadSymmetricDependencies

```
1: Inputs:
      —

2: Output:
      ECC and hybrid primitives loaded; key exchange utilities available

3: Procedure LoadHybridDependencies()
4:       Import ECC primitives for key generation and exchange
5:       Import ECC key interface classes for private and public keys
6:       Load HKDF and hashing tools required for key derivation
7:       Enable block-cipher and AEAD modules used in hybrid encryption
8:       Load lightweight block ciphers (Speck, Simon, Present, Twine)
9:       Import system utilities for randomness and timestamps
```

```
10:     Register helper conversion procedure for bytes-to-bits
11:     Define ECC key-pair generation procedure
12:     Define key-exchange procedure combining ECDH and HKDF-SHA256
13:     Define hybrid encryption and decryption routines for ECC-based algorithms
14:     Return handles to all loaded modules
15: endProcedure
```

Algorithm A.2 LoadHybridDependencies

Section 2 – API Setup

This section delineates the API initialization and its role in coordinating the selection of one cryptographic algorithm from a total of sixteen symmetric and hybrid options. This API is implemented in a distinct module titled “consolidate.py”, encompassing the system startup segment detailed in Algorithm A.3, along with the basic functions for activating symmetric and hybrid cryptographic algorithms as illustrated in Algorithm A.4 and Algorithm A.5, respectively. All library dependencies and version specifications utilized in this work are documented in the supplementary ESM files—requirements.txt and appendix_environment_concise.txt—with an accompanying human-readable summary presented in Appendix A.

```
1: Inputs:
    Algorithm identifier and plaintext (or ciphertext bundle for decryption)

2: Output:
    Standardized encryption/decryption result tuples

3: Procedure WireEncryptionInterface()
4:     Import symmetric and hybrid cryptographic modules
5:     Define encryption dispatcher
6:         If symmetric → call appropriate symmetric encrypt function
7:         If hybrid → generate ECC keys and derive session key
8:         Return standardized encryption tuple
9:     Define decryption dispatcher
10:        If symmetric → call symmetric decrypt function
11:        If hybrid → call hybrid decrypt function
12:    Return both dispatcher functions
13: endProcedure
```

Algorithm A.3 WireEncryptionInterface

```
1: Inputs:
    Algorithm name, plaintext
```

```
2: Output:
      Ciphertext, key, iv/nonce, tag, aad, ECC key materials (if hybrid)

3: Procedure ConsolidateEncryption(algo, plaintext)
4:       If algorithm is symmetric → call symmetric encryption
5:       Else (hybrid algorithm):
6:             Generate ECC key pairs
7:             Derive session key using key-exchange
8:             Call hybrid encryption with derived key
9:       Return encryption outputs
10: endProcedure
```

Algorithm A.4 ConsolidateEncryption

```
1: Inputs:
      Algorithm name, ciphertext, key, iv/nonce, tag, aad

2: Output:
      Decrypted plaintext

3: Procedure ConsolidateDecryption(algo, ciphertext, key, iv, tag, aad)
4:       If algorithm is symmetric → call symmetric decryption
5:       Else if algorithm is hybrid → call hybrid decryption
6:       Else → return None
7: endProcedure
```

Algorithm A.5 ConsolidateDecryption

Section 3 – Helper Functions

This section presents the self-defined helper function designed to minimize repetitive and lengthy code lines resulting from their frequent occurrence in the sixteen symmetric and hybrid cryptographic method scripts. Algorithm A.6 was established in both modules, ‘encry_decry_sym.py’ and ‘encry_decry_hybrid.py’, to standardize the scripting of cryptographic algorithms, as the majority of cryptographic libraries are structured in terms of bits. Conversely, Algorithm A.7 was exclusively delineated in ‘encry_decry_sym.py’ to produce a random ASCII key specifically necessary for the TWINE128 encryption procedure. This function was added to the TWINE128 encryption script to make it easier to combine ECC and TWINE128. It changes the existing binary-derived key into a format that can be printed in ASCII.

```
1: Inputs:
      Number of bytes
```

```
2: Output:
      Equivalent number of bits

3: Procedure BytesToBits(numbytes)
4:       Compute numbytes × 8
5:       Return result
6: endProcedure
```

Algorithm A.6 BytesToBits

```
1: Inputs:
      Length (default = 16)

2: Output:
      ASCII key string

3: Procedure GenerateAsciiKey(length)
4:       Randomly select ASCII letters/digits of specified length
5:       Return generated key
6: endProcedure
```

Algorithm A.7 GenerateAsciiKey

Section 4 – Symmetric Cryptographic Operations

This part presents the pseudocode for the remaining fifteen candidate symmetric cryptographic algorithms, excluding AES-GCM, which is already detailed in the main text, as documented in the 'encry_decry_sym.py' module. The chosen key size for all candidate symmetric cryptographic algorithms was either 128 or 256 bits, and the maximum block size for each technique was decided based on the logic outlined in Section 4.1.1 of the main text. The supplementary components utilized to guarantee data authenticity and integrity in AES-GCM are the initialization vector (IV), authentication tag, and additional authenticated data (AAD). Dworkin [1] has proposed utilizing 96 bits (12 bytes) and 128 bits (16 bytes) for the initialization vector (IV) and authentication tag of AES-GCM, emphasizing that the IV must be unique for each encryption, even while employing the same key. To diminish complexity and minimize processing cost, AAD was omitted from AES-GCM, as it is not mandatory and just enhances the confidentiality and validity of the ciphertext and IV, rather than serving as the primary protective mechanism. ChaCha20-Poly1305, ASCON-128, and ASCON-128a employ an authentication scheme identical to that of AES-GCM, with nonce values of 96 bits (12 bytes) and 128 bits (16 bytes) serving the same purpose as the IV and tag [2, 3]. Although AAD was not mandatory for ChaCha20-Poly1305, ASCON-128, and ASCON-128a as it is for AES-GCM, AAD was illustrated for these three algorithms by the combination of a fictional device name and timestamp. The cryptographic parameter configurations were applied in Algorithms A.8 through A.13, respectively. The residual cryptographic algorithms that do not require the values of IV, nonce, authentication tag, and ADD were assigned placeholder values for each unnecessary parameter to streamline the integration of all algorithms within the same API module, as illustrated in Algorithm A.14 through Algorithm A.21.

```
1: Inputs:
      Plaintext string

2: Output:
      Ciphertext_with_tag, key (32B), nonce (12B), tag (16B), aad

3: Procedure ChaCha20Encrypt(plaintext)
4:        Generate a random 32-byte key and a 12-byte nonce
5:        Encode plaintext into bytes
6:        Construct AAD using device identifier and timestamp
7:        Initialize ChaCha20-Poly1305 with generated key
8:        Encrypt using (nonce, plaintext_bytes, aad) → ciphertext_with_tag
9:        Extract final 16 bytes of ciphertext as tag
10:       Return ciphertext_with_tag, key, nonce, tag, aad
11: endProcedure
```

Algorithm A.8 ChaCha20Encrypt

```
1: Inputs:
      Ciphertext_with_tag, key, nonce, aad

2: Output:
      Decrypted plaintext string or "[authentication failed]"

3: Procedure ChaCha20Decrypt(ciphertext_with_tag, key, nonce, aad)
4:        Initialize ChaCha20-Poly1305 with provided key
5:        Attempt decryption using (nonce, ciphertext_with_tag, aad) → result_bytes
6:        If result_bytes is None → return "[authentication failed]"
7:        Decode result_bytes into plaintext string
8:        Return decrypted plaintext
9: endProcedure
```

Algorithm A.9 ChaCha20Decrypt

```
1: Inputs:
      Plaintext string
```

```
2: Output:
      Ciphertext_with_tag, key (16B), nonce (16B), tag (16B), aad

3: Procedure Ascon128Encrypt(plaintext)
4:      Generate a random 16-byte key and 16-byte nonce
5:      Encode plaintext into bytes
6:      Construct AAD using device identifier and timestamp
7:      Encrypt using Ascon-128 → ciphertext_with_tag
8:      Extract final 16 bytes of ciphertext as tag
9:      Return ciphertext_with_tag, key, nonce, tag, aad
10: endProcedure
```

Algorithm A.10 Ascon128Encrypt

```
1: Inputs:
      Ciphertext_with_tag, key, nonce, aad

2: Output:
      Decrypted plaintext string or “[authentication failed]”

3: Procedure Ascon128Decrypt(ciphertext_with_tag, key, nonce, aad)
4:      Attempt Ascon-128 decryption → result_bytes
5:      If result_bytes is None → return “[authentication failed]”
6:      Decode result_bytes to plaintext string
7:      Return decrypted plaintext
8: endProcedure
```

Algorithm A.11 Ascon128Decrypt

```
1: Inputs:
      Plaintext string

2: Output:
      Ciphertext_with_tag, key (16B), nonce (16B), tag (16B), aad

3: Procedure Ascon128aEncrypt(plaintext)
4:      Generate a random 16-byte key and 16-byte nonce
5:      Encode plaintext into bytes
6:      Construct AAD using device identifier and timestamp
7:      Encrypt using Ascon-128a → ciphertext_with_tag
8:      Extract final 16 bytes as authentication tag
9:      Return ciphertext_with_tag, key, nonce, tag, aad
10: endProcedure
```

Algorithm A.12 Ascon128aEncrypt

```
1: Inputs:
      Ciphertext_with_tag, key, nonce, aad

2: Output:
      Decrypted plaintext string or “[authentication failed]”

3: Procedure Ascon128aDecrypt(ciphertext_with_tag, key, nonce, aad)
4:       Attempt Ascon-128a decryption → result_bytes
5:       If result_bytes is None → return “[authentication failed]”
6:       Decode result_bytes into plaintext string
7:       Return decrypted plaintext
8: endProcedure
```

Algorithm A.13 Ascon128aDecrypt

```
1: Inputs:
      Plaintext string

2: Output:
      Ciphertext, key (16B), iv = b"None", tag = b"None", aad = b"None"

3: Procedure Speck128Encrypt(plaintext)
4:       Generate a random 16-byte key
5:       Initialize Speck-128 cipher with the key
6:       Encode plaintext into bytes
7:       Apply PKCS padding to produce 16-byte blocks
8:       Split padded plaintext into 16-byte blocks
9:       Encrypt each block to produce ciphertext
10:      Return ciphertext, key, b"None", b"None", b"None"
11: endProcedure
```

Algorithm A.14 Speck128Encrypt

```
1: Inputs:
      Ciphertext bytes, key (16B)

2: Output:
      Decrypted plaintext string

3: Procedure Speck128Decrypt(ciphertext, key)
4:       Initialize Speck-128 cipher using the provided key
5:       Split ciphertext into 16-byte blocks
6:       Decrypt each block to obtain padded plaintext
```

```
7:      Remove PKCS padding
8:      Decode plaintext bytes into string
9:      Return decrypted plaintext
10: endProcedure
```

Algorithm A.15 Speck128Decrypt

```
1: Inputs:
    Plaintext string

2: Output:
    Ciphertext, key (16B), iv = b"None", tag = b"None", aad = b"None"

3: Procedure Simon128Encrypt(plaintext)
4:      Generate a random 16-byte key
5:      Initialize Simon-128 cipher with the key
6:      Encode plaintext into bytes
7:      Apply PKCS padding (16-byte block size)
8:      Split padded plaintext into 16-byte blocks
9:      Encrypt each block to produce ciphertext
10:     Return ciphertext, key, b"None", b"None", b"None"
11: endProcedure
```

Algorithm A.16 Simon128Encrypt

```
1: Inputs:
    Ciphertext bytes, key (16B)

2: Output:
    Decrypted plaintext string

3: Procedure Simon128Decrypt(ciphertext, key)
4:      Initialize Simon-128 cipher using the provided key
5:      Split ciphertext into 16-byte blocks
6:      Decrypt each block to obtain padded plaintext
7:      Remove PKCS padding
8:      Decode plaintext bytes into a string
9:      Return decrypted plaintext
10: endProcedure
```

Algorithm A.17 Simon128Decrypt

```
1: Inputs:
    Plaintext string
```

```
2: Output:
     Ciphertext, key (16B), iv = b"None", tag = b"None", aad = b"None"

3: Procedure Present128Encrypt(plaintext)
4:       Generate a random 16-byte key
5:       Convert key to hexadecimal string form
6:       Initialize Present-128 cipher using converted key
7:       Encode plaintext into bytes
8:       Apply PKCS padding (8-byte block size)
9:       Split padded plaintext into 8-byte blocks
10:        Encrypt each block to obtain ciphertext
11:        Return ciphertext, key, b"None", b"None", b"None"
12: endProcedure
```

Algorithm A.18 Present128Encrypt

```
1: Inputs:
     Ciphertext bytes, key (16B)

2: Output:
     Decrypted plaintext string

3: Procedure Present128Decrypt(ciphertext, key)
4:       Convert key to hexadecimal and then to integer
5:       Initialize Present-128 cipher using key_int
6:       Split ciphertext into 8-byte blocks
7:       Decrypt each block to obtain padded plaintext
8:       Remove PKCS padding
9:       Decode plaintext bytes into a string
10:        Return decrypted plaintext
11: endProcedure
```

Algorithm A.19 Present128Decrypt

```
1: Inputs:
     Plaintext string

2: Output:
     Ciphertext, key (ASCII 16 chars), iv = b"None", tag = b"None", aad = b"None"

3: Procedure Twine128Encrypt(plaintext)
4:       Generate a 16-character ASCII key
5:       Initialize Twine-128 cipher with ASCII key
6:       Encode plaintext into bytes
7:       Apply PKCS padding (16-byte block size)
8:       Split padded plaintext into 16-byte blocks
```

```
9:      Encrypt each block to generate ciphertext
10:     Return ciphertext, key, b"None", b"None", b"None"
11: endProcedure
```

Algorithm A.20 Twine128Encrypt

```
1: Inputs:
     Ciphertext bytes, ASCII key string

2: Output:
     Decrypted plaintext string

3: Procedure Twine128Decrypt(ciphertext, key)
4:      Initialize Twine-128 cipher using ASCII key
5:      Split ciphertext into 16-byte blocks
6:      Decrypt each block to obtain padded plaintext
7:      Remove PKCS padding
8:      Decode plaintext bytes into a string
9:      Return decrypted plaintext
10: endProcedure
```

Algorithm A.21 Twine128Decrypt

Section 5 – ECC Module Helper Functions

This section describes the helper functions employed to simulate ECC key pair generation and key exchange between the sender and receiver in the hybrid cryptographic algorithms contained within the encry_decry_hybrid.py module. Algorithm A.22 illustrates the construction of public and private keys, whereas Algorithm A.23 demonstrates the derivation of the shared session key utilizing the sender's public key and the receiver's private key via the ECC-based key exchange procedure.

```
1: Inputs:
     —

2: Output:
     Private_key, public_key, private_key_bytes (32B), public_key_bytes (65B)

3: Procedure GenerateEccKeyPair()
4:      Generate a P-256 private key
5:      Derive the associated public key
6:      Extract raw private scalar as a 32-byte sequence
7:      Extract public key coordinates (x, y) and prepend 0x04 byte
8:      Construct 65-byte uncompressed public key encoding
```

```
9:      Return private_key, public_key, private_key_bytes, public_key_bytes
10: endProcedure
```

Algorithm A.22 GenerateEccKeyPair

```
1: Inputs:
     Self_private_key, peer_public_key, extracted_key_size (integer)

2: Output:
     Derived_key (bytes)

3: Procedure PerformKeyExchange(self_private_key, peer_public_key,
     extracted_key_size)
4:      Compute shared_secret using ECDH with curve P-256
5:      Apply HKDF-SHA256 with length = extracted_key_size and info = b"session
          key"
6:      Derive output key from shared_secret
7:      Return derived_key
8: endProcedure
```

Algorithm A.23 PerformKeyExchange

Section 6 – Hybrid Cryptographic Operations

This section outlines the pseudocodes for hybrid cryptographic algorithms (excluding ECC + AES-GCM, which is already depicted in the main text) that are formulated by integrating candidate symmetric cryptographic algorithms with ECC, as documented in the 'encry_decry_hybrid.py' module. The scripting approach resembled symmetric cryptographic methods but included an additional key exchange and shared session key derivation step before the encryption and decryption stages. Algorithms A.24 to A.37 exhibit the remaining pseudocodes for ECC-based hybrid cryptographic operations, analogous to the ECC + AES-GCM pseudocode example; however, they differ in their cryptographic parameter setups.

```
1: Inputs:
     Plaintext, self_private_key, peer_public_key

2: Output:
     Ciphertext_with_tag, derived_key (32B), nonce (12B), tag (16B), aad

3: Procedure EccChaCha20Encrypt(plaintext, self_priv, peer_pub)
4:      Perform key exchange to derive a 32-byte symmetric key
5:      Generate a random 12-byte nonce
6:      Encode plaintext into bytes
```

```
7:      Construct AAD using timestamp and device identifier
8:      Initialize ChaCha20-Poly1305 with derived_key
9:      Encrypt using (nonce, plaintext_bytes, aad) → ciphertext_with_tag
10:     Extract final 16 bytes as authentication tag
11:     Return ciphertext_with_tag, derived_key, nonce, tag, aad
12: endProcedure
```

Algorithm A.24 EccChaCha20Encrypt

```
1: Inputs:
    Ciphertext_with_tag, derived_key (32B), nonce (12B), aad

2: Output:
    Decrypted plaintext string or "[authentication failed]"

3: Procedure EccChaCha20Decrypt(ciphertext_with_tag, derived_key, nonce, aad)
4:      Initialize ChaCha20-Poly1305 with derived_key
5:      Attempt decryption using (nonce, ciphertext_with_tag, aad) → result_bytes
6:      If result_bytes is None → return "[authentication failed]"
7:      Decode result_bytes to plaintext string
8:      Return decrypted plaintext
9: endProcedure
```

Algorithm A.25 EccChaCha20Decrypt

```
1: Inputs:
    Plaintext, self_private_key, peer_public_key

2: Output:
    Ciphertext_with_tag, derived_key (16B), nonce (16B), tag (16B), aad

3: Procedure EccAscon128Encrypt(plaintext, self_priv, peer_pub)
4:      Perform key exchange to derive a 16-byte key
5:      Generate a random 16-byte nonce
6:      Encode plaintext into bytes
7:      Construct AAD using device identifier and timestamp
8:      Encrypt using Ascon-128 → ciphertext_with_tag
9:      Extract final 16 bytes as tag
10:     Return ciphertext_with_tag, derived_key, nonce, tag, aad
11: endProcedure
```

Algorithm A.26 EccAscon128Encrypt

```
1: Inputs:
     Ciphertext_with_tag, derived_key (16B), nonce (16B), aad

2: Output:
     Decrypted plaintext string or “[authentication failed]”

3: Procedure EccAscon128Decrypt(ciphertext_with_tag, derived_key, nonce, aad)
4:       Attempt Ascon-128 decryption → result_bytes
5:       If result_bytes is None → return “[authentication failed]”
6:       Decode result_bytes to plaintext string
7:       Return decrypted plaintext
8: endProcedure
```

Algorithm A.27 EccAscon128Decrypt

```
1: Inputs:
     Plaintext, self_private_key, peer_public_key

2: Output:
     Ciphertext_with_tag, derived_key (16B), nonce (16B), tag (16B), aad

3: Procedure EccAscon128aEncrypt(plaintext, self_priv, peer_pub)
4:       Perform key exchange to derive a 16-byte key
5:       Generate a random 16-byte nonce
6:       Encode plaintext into bytes
7:       Construct AAD using device identifier and timestamp
8:       Encrypt using Ascon-128a → ciphertext_with_tag
9:       Extract final 16 bytes as tag
10:      Return ciphertext_with_tag, derived_key, nonce, tag, aad
11: endProcedure
```

Algorithm A.28 EccAscon128aEncrypt

```
1: Inputs:
     Ciphertext_with_tag, derived_key (16B), nonce (16B), aad

2: Output:
     Decrypted plaintext string or “[authentication failed]”

3: Procedure EccAscon128aDecrypt(ciphertext_with_tag, derived_key, nonce, aad)
4:       Attempt Ascon-128a decryption → result_bytes
5:       If result_bytes is None → return “[authentication failed]”
6:       Decode result_bytes to plaintext string
7:       Return decrypted plaintext
8: endProcedure
```

Algorithm A.29 EccAscon128aDecrypt

```
1: Inputs:
    Plaintext, self_private_key, peer_public_key

2: Output:
    Ciphertext, derived_key (16B), iv = b"None", tag = b"None", aad = b"None"

3: Procedure EccSpeck128Encrypt(plaintext, self_priv, peer_pub)
4:      Perform key exchange to derive a 16-byte key
5:      Initialize Speck-128 cipher using derived_key
6:      Encode plaintext into bytes
7:      Apply PKCS padding (16-byte blocks)
8:      Split padded plaintext into 16-byte blocks
9:      Encrypt blocks to form ciphertext
10:     Return ciphertext, derived_key, b"None", b"None", b"None"
11: endProcedure
```

Algorithm A.30 EccSpeck128Encrypt

```
1: Inputs:
    Ciphertext bytes, derived_key (16B)

2: Output:
    Decrypted plaintext string

3: Procedure EccSpeck128Decrypt(ciphertext, derived_key)
4:      Initialize Speck-128 cipher using derived_key
5:      Split ciphertext into 16-byte blocks
6:      Decrypt blocks to obtain padded plaintext
7:      Remove PKCS padding
8:      Decode plaintext bytes into a string
9:      Return decrypted plaintext
10: endProcedure
```

Algorithm A.31 EccSpeck128Decrypt

```
1: Inputs:
    Plaintext, self_private_key, peer_public_key

2: Output:
    Ciphertext, derived_key (16B), iv = b"None", tag = b"None", aad = b"None"
```

```
3: Procedure EccSimon128Encrypt(plaintext, self_priv, peer_pub)
4:      Perform key exchange to derive a 16-byte key
5:      Initialize Simon-128 cipher using derived_key
6:      Encode plaintext into bytes
7:      Apply PKCS padding (16-byte block size)
8:      Split padded plaintext into 16-byte blocks
9:      Encrypt blocks to produce ciphertext
10:     Return ciphertext, derived_key, b"None", b"None", b"None"
11: endProcedure
```

Algorithm A.32 EccSimon128Encrypt

```
1: Inputs:
    Ciphertext bytes, derived_key (16B)

2: Output:
    Decrypted plaintext string

3: Procedure EccSimon128Decrypt(ciphertext, derived_key)
4:      Initialize Simon-128 cipher with derived_key
5:      Split ciphertext into 16-byte blocks
6:      Decrypt blocks to obtain padded plaintext
7:      Remove PKCS padding
8:      Decode plaintext bytes into string
9:      Return decrypted plaintext
10: endProcedure
```

Algorithm A.33 EccSimon128Decrypt

```
1: Inputs:
    Plaintext, self_private_key, peer_public_key

2: Output:
    Ciphertext, derived_key (16B), iv = b"None", tag = b"None", aad = b"None"

3: Procedure EccPresent128Encrypt(plaintext, self_priv, peer_pub)
4:      Perform key exchange to derive a 16-byte key
5:      Convert derived_key to hexadecimal string
6:      Initialize Present-128 cipher using key_hex
7:      Encode plaintext into bytes
8:      Apply PKCS padding (8-byte block size)
9:      Split padded plaintext into 8-byte blocks
10:     Encrypt blocks to form ciphertext
11:     Return ciphertext, derived_key, b"None", b"None", b"None"
12: endProcedure
```

Algorithm A.34 EccPresent128Encrypt

```
1: Inputs:
      Ciphertext bytes, derived_key (16B)

2: Output:
      Decrypted plaintext string

3: Procedure EccPresent128Decrypt(ciphertext, derived_key)
4:        Convert derived_key to hexadecimal integer
5:        Initialize Present-128 cipher using key_int
6:        Split ciphertext into 8-byte blocks
7:        Decrypt blocks to obtain padded plaintext
8:        Remove PKCS padding
9:        Decode plaintext bytes into string
10:       Return decrypted plaintext
11: endProcedure
```

Algorithm A.35 EccPresent128Decrypt

```
1: Inputs:
      Plaintext, self_private_key, peer_public_key

2: Output:
      Ciphertext, derived_key (16B), iv = b"None", tag = b"None", aad = b"None"

3: Procedure EccTwine128Encrypt(plaintext, self_priv, peer_pub)
4:        Perform key exchange to derive a 16-byte key
5:        Convert derived_key into ASCII-compatible form
6:        Initialize Twine-128 cipher using ASCII key
7:        Encode plaintext into bytes
8:        Apply PKCS padding (16-byte block size)
9:        Split padded plaintext into 16-byte blocks
10:       Encrypt blocks to form ciphertext
11:       Return ciphertext, derived_key, b"None", b"None", b"None"
12: endProcedure
```

Algorithm A.36 EccTwine128Encrypt

```
1: Inputs:
      Ciphertext bytes, derived_key (ASCII-compatible 16B)

2: Output:
      Decrypted plaintext string
```

```
3: **Procedure** *EccTwine128Decrypt*(ciphertext, derived_key)
4:     Initialize Twine-128 cipher with ASCII-compatible derived_key
5:     Split ciphertext into 16-byte blocks
6:     Decrypt blocks to obtain padded plaintext
7:     Remove PKCS padding
8:     Decode plaintext bytes into string
9:     Return decrypted plaintext
10: **endProcedure**
```

Algorithm A.37 EccTwine128Decrypt

References

1. Dworkin M. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D. Gaithersburg (MD): National Institute of Standards and Technology; 2007 Nov. https://doi.org/10.6028/NIST.SP.800-38D.
2. Nir Y, Langley A. ChaCha20 and Poly1305 for IETF protocols. RFC 8439. Fremont (CA): Internet Engineering Task Force; 2018 Jun. https://doi.org/10.17487/RFC8439.
3. Turan MS, McKay KA, Chang D, Kang J, Kelsey J. Ascon-based lightweight cryptography standards for constrained devices. NIST Spec Publ. 2025;(800-232). https://doi.org/10.6028/NIST.SP.800-232.

Appendix A — Execution Environment

Scope

This appendix summarizes the execution environment used to generate the cryptographic simulations and pseudocode results (Algorithms A.1–A.37). The complete machine-readable metadata is provided in appendix_environment_concise.txt, generated automatically by generate_env_info_auto.py.
All package versions are pinned in requirements.txt for reproducibility.

Table A.1 Runtime Environment Summary

Area	Description
Operating system	Windows 11 (10.0.26200, 64-bit)
CPU	Intel64 Family 6 Model 183 (GenuineIntel)
Python	3.13.2 (build tags/v3.13.2:4f8bb39, Feb 4 2025)
Virtual environment	venv
OpenSSL (system)	3.0.15 (3 Sep 2024)
cryptography package	v45.0.4 (backend OpenSSL 3.5.0, 8 Apr 2025)
Environment snapshot	2025-10-26 17:13:35

Key Third-party Packages

- ascon v0.0.9
- cryptography v45.0.4
- xtwine v1.0.2
  (Complete list available in requirements.txt.)

Local Project Modules

Data_Simulation/ — encry_decry_sym.py, encry_decry_hybrid.py, consolidate.py
Cipher_GitHub/ — lightweight cipher submodules for Simon, Speck, and Present.

Cryptographic Primitives Evaluated

AES-GCM (128/96/128), ChaCha20-Poly1305 (256/96/128), Ascon-128 and Ascon-128a (128/128/128), Speck-128/128, Simon-128/128, PRESENT-128, TWINE-128, and ECC (P-256) with HKDF-SHA256 for hybrid key exchange.

Randomness and Parameters

- Randomness source: os.urandom() for all keys, IVs, and nonces.

- Key exchange: ECDH on curve secp256r1 (NIST P-256).
- Key derivation: HKDF-SHA256 (128-bit for block/Ascon, 256-bit for ChaCha20-Poly1305).
- AEAD associated data: device identifier + timestamp.
- Padding: PKCS or PKCS#7 for block ciphers.

Reproducibility Notes

This environment summary provides the human-readable overview only. For exact replication, recreate the environment using requirements.txt and rerun generate_env_info_auto.py to regenerate appendix_environment_concise.txt. In case of discrepancies, the machine-readable files should be considered authoritative.