Electronic Supplementary Material 2 (ESM2)

Cryptographic Simulation and Data Generation 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 (30 October 2025)

Purpose Note:

This supplementary material provides pseudocode, implementation details, parameter configurations, and the reference Python source files for all symmetric and hybrid cryptographic algorithm simulations, simulated data scaling, and data extraction processes discussed in the main manuscript. It is intended for review purposes and will be formatted in accordance with Springer's ESM journal standards after acceptance.

Contents Summary

File	Description
ESM2.docx	Main supplementary document containing pseudocode, implementation details, and supporting explanations
project_root.zip	Contains the Python sources for the main data generation module and simulation required submodules including IoT crypto reference and required local cipher submodules. Core module (under Data_Simulation/) is: Data_Generation_Final.py (data simulation + scaling + extraction and the resulting output files are located in the Data_Simulation/Simulation Results. Submodules are those that available in ESM 1. 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
ESM2_Cryptographic_Simulation_and_Data_Generation_v0.1.zip	Data generated from the Data_Generation_Final.py scripting

Reader Guide

This document presents organized pseudocode listings (Algorithms B.1–B.15), parameter configurations, and accompanying explanatory notes related to the implementation of symmetric and hybrid cryptographic simulations, as well as data scaling and extraction scripting, as discussed in Section 4.2.2 of the main text. The pseudocode prioritizes clarity and reproducibility over precise runtime equivalency. The accompanying Python scripts in the zip file offer the reference implementation utilized for doing simulations of cryptographic operations, data scaling, and data extraction as described 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. Comprehensive information and explicit instructions for utilizing the Python scripts are available in the accompanying README.txt.

Table of Contents

Section 0 – Algorithm Relationships and Workflow Overview

This section clarifies the relationship between Algorithms B.1 and B.15 utilized in the cryptographic simulation, host-side performance measurement, IoT-side scaling, feasibility assessment, and result-export 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 simulation workflow described in this ESM proceeds through the following algorithmic stages:

- **Algorithm B.1** initializes all required cryptographic, system-monitoring, and simulation libraries, dependencies, helper functions, calibration routines, and project-specific modules.
- **Algorithm B.2** orchestrates the complete simulation workflow, iterating across data sizes and cryptographic algorithms, and calling host-side simulation modules and IoT scaling modules.
- **Algorithm B.3** performs host-side encryption simulations, measuring execution time, CPU time, RAM usage, CPU frequency, and generating cryptographic artifacts.It depends on calibration routines (B.4, B.5) and monitoring utilities (B.9, B.10).
- **Algorithm B.4** calibrates the number of inner-loop encryption iterations required for measurements to exceed timing resolution.
- **Algorithm B.5** determines the number of encryption repeats using statistical confidence bounds.
- **Algorithm B.6** executes host-side decryption simulations, producing timing, memory, CPU, ROM-usage, and correctness-verification metrics.It depends on decryption calibration routines (B.7, B.8) and monitoring utilities (B.9, B.10) and uses ROM estimation (B.12).
- **Algorithm B.7** calibrates the number of decryption iterations.
- **Algorithm B.8** determines reliable decryption repeat counts using statistical models.
- **Algorithm B.9** monitors peak RAM usage during encryption and decryption runs at microsecond intervals.
- **Algorithm B.10** monitors CPU frequency during cryptographic operations at microsecond intervals.
- **Algorithm B.11** estimates energy consumption from execution time and active current, used for IoT feasibility computation.
- **Algorithm B.12** computes ROM usage based on key, nonce, ciphertext, tag, and ECC-key materials.
- **Algorithm B.13** scales host-side encryption metrics to IoT device profiles, evaluates RAM, ROM, CPU, and battery feasibility, and appends encryption results.

- **Algorithm B.14** scales decryption metrics to IoT device profiles, evaluates feasibility, and appends decryption results.
- **Algorithm B.15** aggregates all rows generated for a given data size and exports them to CSV for downstream data analysis steps.

Section 0.2 – Workflow Diagram for Algorithm B.1-B.15

This section presents the sequential cryptographic simulation, host-measurement, IoT scaling, feasibility evaluation, and results-export workflow executed by Algorithms B.1 through B.15. The complete pipeline is illustrated in Fig. 1.

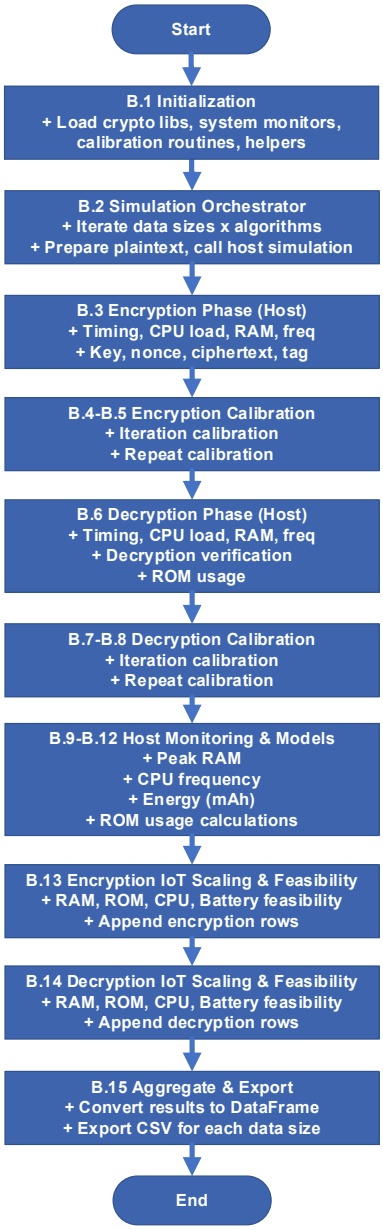

Fig. 1 Workflow diagram for the cryptographic simulation, host-measurement, IoT scaling, feasibility evaluation, and results-export pipeline (Algorithms B.1–B.15)

Section 0.3 – Algorithm Dependency Table

This section summarizes the dependencies among the algorithms and outlines their primary roles within the cryptographic simulation and IoT-scaling workflow as shown in Table 1.

Table 1 Algorithm dependencies and functional roles

Algorithm	Depends On	Provides Output For	Functional Role
B.1	—	B.2–B.15	Initialize simulation environment, crypto libs, monitoring, calibration tools
B.2	B.1, B.3, B.6, B.13–B.15	—	Main orchestrator: iterate data sizes × algorithms, call host sim + scaling + export
B.3	B.4, B.5, B.9, B.10	B.13	Host-side encryption: timings, RAM/CPU/freq metrics, ciphertext artifacts
B.4	—	B.3	Calibrate encryption iterations
B.5	B.4	B.3	Calibrate encryption repeats using statistical confidence
B.6	B.7, B.8, B.9, B.10, B.12	B.14	Host-side decryption: timings, RAM/CPU/freq, ROM, verification
B.7	—	B.6	Calibrate decryption iterations
B.8	B.7	B.6	Calibrate decryption repeats
B.9	—	B.3, B.6	Monitor peak RAM usage
B.10	—	B.3, B.6	Monitor CPU frequency
B.11	—	B.13, B.14	Estimate energy consumption
B.12	—	B.6, B.13, B.14	Compute ROM usage
B.13	B.3, B.11, B.12	B.2	IoT-side scaling & feasibility (encryption)
B.14	B.6, B.11, B.12	B.2	IoT-side scaling & feasibility (decryption)
B.15	B.2	—	Aggregate simulation rows and export to CSV

Section 1 – System Initialization

This section outlines the necessary libraries and dependencies for establishing the cryptographic simulation environment on the host machine, enabling the results to be scaled according to the specifications of the IoT device and subsequently retrieved for data analysis. Algorithm B.1 specifies the libraries and dependencies necessary for simulating cryptographic operations; data scaling and extraction are positioned at the

beginning of the module titled ‘Data_Generation_Final.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:**
Imports loaded; helpers defined; environment ready for simulation

3: **Procedure** *SetupAndLoadDependencies*()
4: Import time, gc, os, threading, statistics, math
5: Import psutil and pandas
6: Import Data_Simulation.consolidate module for encryption/decryption wrappers
7: Register helper function for estimating crypto energy (Algorithm B.11)
8: Register helper function for ROM usage simulation (Algorithm B.12)
9: Register host RAM peak monitoring routine (Algorithm B.9)
10: Register host CPU frequency monitoring routine (Algorithm B.10)
11: Register encryption iteration calibration routine (Algorithm B.4)
12: Register encryption repeat calibration routine (Algorithm B.5)
13: Register decryption iteration calibration routine (Algorithm B.7)
14: Register decryption repeat calibration routine (Algorithm B.8)
15: Configure main orchestrator function for simulation (Algorithm B.2)
16: If module is executed as main → invoke EncryptionSimulation()
17: **endProcedure**

Algorithm B.1 SetupAndLoadDependencies

Section 2 – Cryptographic Operations Simulation

This section presents the pseudocode utilized to simulate cryptographic operations for sixteen symmetric and hybrid algorithms, incorporating various combinations of IoT device specifications on the host machine. All the pseudocode listings in this part are contained within the module titled ‘Data_Generation_Final.py’, immediately following the system initialization scripts. Algorithm B.2 delineates the comprehensive simulation scripting employed in this research, which includes encryption and decryption simulation, data scaling, and data extraction. Algorithms B.3 and B.6 delineate the comprehensive procedures for simulating encryption and decryption, as well as measuring and calculating performance and resource consumption, encompassing encryption and decryption duration, CPU execution time, CPU frequency, and RAM and ROM usage. Algorithms B.4 and B.5 served as the pseudocodes for iterating and repeating the cryptographic encryption to provide dependable and precise measurement, whilst Algorithms B.7 and B.8 were employed

for cryptographic decryption. Algorithms B.9–B.12 served as pseudocode for helper functions utilized in measuring and calculating performance and resource utilization. Specifically, Algorithms B.9 and B.10 were employed for RAM and CPU frequency measurement at a fixed interval of 10 μs, whereas Algorithms B.11 and B.12 were designated for calculating battery consumption and ROM usage, respectively.

1: **Inputs:**
—

2: **Output:**
CSV results for all data sizes, algorithms, and IoT device profiles

3: **Procedure** *EncryptionSimulation*()
4: Define experiment ranges for data size, battery capacity, RAM, ROM, CPU power, core count, active current, threat level, and data sensitivity
5: For each data_size_kb in configured range:
6: Initialize results list
7: Construct plaintext of size data_size_kb × 1024 bytes
8: Obtain handle to current process for metric sampling
9: For each algorithm in algorithm list:
10: Execute host encryption phase → collect encryption metrics
11: (Algorithm B.3)
12: Scale encryption metrics to IoT device grid and check feasibility
13: (Algorithm B.13)
14: Execute host decryption phase → collect decryption metrics
15: (Algorithm B.6)
16: Scale decryption metrics to IoT device grid and check feasibility
17: (Algorithm B.14)
18: Export accumulated results for the current data size to CSV
19: (Algorithm B.15)
20: Print simulation completion message
21: **endProcedure**

Algorithm B.2 EncryptionSimulation

1: **Inputs:**
algo, plaintext, process

2: **Output:**
ciphertext, key, iv_nonce, tag, aad,
recv_priv, recv_pub, send_priv, send_pub,
host_freq_avg_mhz_encrypt, iterations_encrypt, repeats_encrypt,
duration_encrypt, cpu_time_encrypt, mem_before_encrypt,
peak_mem_during_encrypt, parallelism_encrypt,
host_cpu_usage_encrypt, host_cpu_core_encrypt

```
3: Procedure EncryptionPhaseHost(algo, plaintext, process)
4:       iterations_encrypt ← calibrate encryption iterations using target time of 1
           second
5:       repeats_encrypt ← calibrate number of repeats using statistical bounds
6:       Invoke garbage collector; record baseline RAM usage
7:       Start RAM peak monitor thread
8:       Start CPU frequency monitor thread
9:       Initialize wall-clock and CPU-time tracking arrays
10:        For each repeat r in repeats_encrypt:
11:              Record CPU start time and wall-clock start
12:              For each iteration i in iterations_encrypt:
13:                    Perform encryption using consolidated API
14:              Record wall-clock end and CPU end time
15:              Append elapsed values to timing arrays
16:      Stop monitor threads and join
17:      Compute peak RAM and average CPU frequency
18:      Compute median encryption duration per iteration
19:      Compute median CPU time per iteration
20:      Compute host CPU usage percentage
21:      Determine parallelism and effective CPU core usage
22:      Return all collected encryption metrics
23: endProcedure
```

Algorithm B.3 EncryptionPhaseHost

```
1: Inputs:
     algo, plaintext, target_time

2: Output:
     iterations_encrypt

3: Procedure CalibrateIterationsEncryption(algo, plaintext, target_time)
4:       iterations ← 1
5:       Loop:
6:            Record start time
7:            Repeat encryption for current iteration count
8:            Compute elapsed time
9:            If elapsed ≥ target_time → break
10:             Else if elapsed == 0 → multiply iterations by 10
11:             Else scale iterations proportionally to target_time
12:      Return iterations
13: endProcedure
```

Algorithm B.4 CalibrateIterationsEncryption

```
1: Inputs:
      iterations_encrypt, algo, plaintext,
      target_rel_err, conf_level, pilot_runs, min_iters, max_iters

2: Output:
      m

3: Procedure CalibrateRepeatsEncryption(…)
4:       Initialize sample array
5:       For r in pilot_runs:
6:             Record CPU time before and after running encryption iterations
7:             Append CPU delta to samples
8:       Compute mean and standard deviation
9:       Determine z-value from confidence level
10:      Compute required repeats using normal-approximation formula
11:      Clamp repeats between min_iters and max_iters
12:      Return m
13: endProcedure
```

Algorithm B.5 CalibrateRepeatsEncryption

```
1: Inputs:
      algo, ciphertext, key, iv_nonce, tag, aad, plaintext, process

2: Output:
      decrypted, host_freq_avg_mhz_decrypt, iterations_decrypt,
      repeats_decrypt, duration_decrypt, cpu_time_decrypt,
      mem_before_decrypt, peak_mem_during_decrypt,
      parallelism_decrypt, host_cpu_usage_decrypt, host_cpu_core_decrypt,
      rom_used_mb_decrypt, ram_usage_decrypt, decryption_verification

3: Procedure DecryptionPhaseHost(…)
4:       iterations_decrypt ← calibrate decryption iterations with target time of 1
5:       repeats_decrypt ← calibrate decryption repeats using statistical model
6:       Invoke garbage collector; record baseline RAM
7:       Start RAM peak monitor thread
8:       Start CPU frequency monitor thread
9:       Initialize timing arrays
10:      For each repeat r in repeats_decrypt:
11:            Record CPU and wall start times
12:            Perform decryption for each iteration
13:            Store last decrypted output
14:            Record times and append to arrays
15:      Stop threads and join
16:      Compute peak RAM and average CPU frequency
17:      Compute median decryption time per iteration
18:      Compute CPU time per iteration
19:      Verify correctness of decrypted output
```

```
20:       Compute host CPU usage
21:       Determine parallelism behavior
22:       Compute ROM usage for decryption
23:       Compute RAM usage delta
24:       Return all outputs
25: endProcedure
```

Algorithm B.6 DecryptionPhaseHost

```
1: Inputs:
     algo, ciphertext, key, iv_nonce, tag, aad, target_time

2: Output:
     iterations_decrypt

3: Procedure CalibrateIterationsDecryption(…)
4:       iterations ← 1
5:       Loop:
6:            Record start time
7:            Perform decryption iterations
8:            Compute elapsed time
9:            If elapsed ≥ target_time → break
10:             Else if elapsed == 0 → multiply iterations by 10
11:             Else scale proportionally
12:      Return iterations
13: endProcedure
```

Algorithm B.7 CalibrateIterationsDecryption

```
1: Inputs:
     iterations_decrypt, algo, ciphertext, key, iv_nonce, tag, aad,
     target_rel_err, conf_level, pilot_runs, min_iters, max_iters

2: Output:
     n

3: Procedure CalibrateRepeatsDecryption(…)
4:       Initialize sample list
5:       For each pilot run:
6:            Record CPU before and after running decryption iterations
7:            Append CPU delta to samples
8:       Compute statistical parameters
9:       Determine z-value from confidence level
10:      Compute required n
11:      Clamp to [min_iters, max_iters]
```

12: Return n
13: **endProcedure**

Algorithm B.8 CalibrateRepeatsDecryption

1: **Inputs:**
process, interval

2: **Output:**
running, peak, thread

3: **Procedure** *MonitorPeakRss*(process, interval)
4: Initialize peak tracker and running flag
5: Define thread loop that samples RSS and updates peak
6: Start thread
7: Return running flag, peak reference, and thread
8: **endProcedure**

Algorithm B.9 MonitorPeakRss

1: **Inputs:**
interval

2: **Output:**
running, cpu_freq_samples, thread

3: **Procedure** *MonitorHostCpuStats*(interval)
4: Initialize sample list and running flag
5: Define thread loop to sample current CPU frequency
6: Start thread
7: Return running flag, samples list, and thread
8: **endProcedure**

Algorithm B.10 MonitorHostCpuStats

1: **Inputs:**
time_s, active_current_ma

2: **Output:**
energy_mah

```
3: Procedure EstimateCryptoEnergy(time_s, active_current_ma)
4:       Return (active_current_ma × time_s) ÷ 3600
5: endProcedure
```

Algorithm B.11 EstimateCryptoEnergy

```
1: Inputs:
     crypto_type, key, iv_nonce, ciphertext, tag,
     receiver_private_key_bytes, receiver_public_key_bytes,
     sender_private_key_bytes, sender_public_key_bytes

2: Output:
     total_bytes

3: Procedure SimulateRomUsage(…)
4:       If crypto_type = "enc":
5:             Return total size of key, iv/nonce, ciphertext, tag,
                 receiver public key, sender private key
6:       Else:
7:             Return size of key, iv/nonce, ciphertext, tag,
                 receiver private key, sender public key
8: endProcedure
```

Algorithm B.12 SimulateRomUsage

Section 3 – IoT Scaling and Feasibility Check

This section presents the pseudocode utilized for data scaling, resource consumption feasibility assessments, and data exporter. Algorithms B.13 and B.14 present the pseudocodes utilized for initially scaling the encryption and decryption simulation outcomes of the host machine, followed by employing the scaled results to assess resource consumption feasibility in accordance with the corresponding specifications of IoT device pairs. Algorithm B.15 defines the pseudocode for exporting all requisite outcomes in CSV format, essential for the next data analysis step.

```
1: Inputs:
     Encryption host metrics, ECC key objects, IoT ranges, algo,
     data_size_kb, results

2: Output:
     Encrypted rows appended to results

3: Procedure EncryptionIotScalingAndFeasibility(…)
4:       crypto_type ← "enc"
5:       Compute ROM usage using SimulateRomUsage
6:       Convert ROM bytes to MB
```

```
7:      Compute RAM usage delta
8:      For each IoT device profile:
9:          Compute RAM feasibility
10:         Compute ROM feasibility
11:         If parallel: scale by CPU_power × core_count
12:         Else: scale by CPU_power alone
13:         Compute energy consumption
14:         Compute remaining battery
15:         Compute battery feasibility
16:         For each threat × sensitivity pair:
17:             Append encryption row to results
18: endProcedure
```

Algorithm B.13 EncryptionIotScalingAndFeasibility

```
1: Inputs:
     Decryption host metrics, IoT ranges, algo,
     data_size_kb, decryption_verification, results

2: Output:
     Decryption rows appended to results

3: Procedure DecryptionIotScalingAndFeasibility(…)
4:      For each IoT device profile:
5:          Compute RAM feasibility
6:          Compute ROM feasibility
7:          If parallel → scale by CPU_power × core_count
8:          Else → scale by CPU_power
9:          Compute energy consumption
10:         Compute remaining battery
11:         Compute battery feasibility
12:         For each threat × sensitivity pair:
13:             Append decryption row to results
14: endProcedure
```

Algorithm B.14 DecryptionIotScalingAndFeasibility

```
1: Inputs:
     results, data_size_kb, export_path_template

2: Output:
     CSV file written

3: Procedure AggregateAndExport(results, data_size_kb, path_tpl)
4:      Convert results list into DataFrame
5:      Display trailing records for sanity check
```

```
6:      Generate filename from template
7:      Export DataFrame to CSV
8:      Print export confirmation
9: endProcedure
```

Algorithm B.15 AggregateAndExport

Appendix A – Execution Environment

Scope

This appendix summarizes the execution environment used to perform the cryptographic simulations and benchmarking experiments described in the manuscript (Algorithms B.1– B.15).
The complete machine-readable metadata is provided in appendix_environment_concise.txt, automatically generated by generate_env_info_auto.py.
All Python package versions are pinned in requirements.txt to ensure reproducibility.

Table A.1 Runtime Environment Summary

Area	Description
Operating system	Windows 11 (10.0.26200, 64-bit, SP0)
CPU	Intel64 Family 6 Model 183 Stepping 1 (GenuineIntel)
Python	3.13.2 (build tags/v3.13.2:4f8bb39, Feb 4 2025 15:23:48)
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 17:13:50

Key Third-party Packages

- cryptography v45.0.4
- pandas v2.3.0
- psutil v7.0.0
  (Complete list available in requirements.txt.)

Benchmark and Simulation Tooling

- Data handling and analysis: pandas v2.3.0
- System metrics and performance tracking: psutil v7.0.0
- Standard-library modules used: gc, math, statistics, threading, time

Local Project Modules

Data_Simulation/ — contains the main simulation and evaluation scripts for cryptographic performance and benchmarking.

Standard Library / Built-in Modules Used

__future__, argparse, dataclasses, datetime, gc, importlib, math, os, pathlib, platform, re, site, ssl, statistics, subprocess, sys, sysconfig, threading, time, typing

Cryptographic Primitives Evaluated

AES-GCM (128/96/128), ChaCha20-Poly1305 (256/96/128), Ascon-128 (128/128/128), Ascon-128a (128/128/128), Speck-128/128, Simon-128/128, PRESENT-128, TWINE-128, and ECC (P-256) with HKDF-SHA256 for key exchange and derivation.
All parameters and security configurations followed standard NIST and IETF recommendations.

Randomness and Parameters

- Randomness source: os.urandom() for all keys, nonces, and IVs.
- Key agreement: ECDH using curve secp256r1 (NIST P-256).
- Key derivation: HKDF-SHA256 (128-bit derived keys for block/Ascon; 256-bit for ChaCha20-Poly1305).
- AEAD associated data: device identifier + timestamp (per implementation).
- Padding: PKCS or PKCS#7 for all block ciphers.

Reproducibility Notes

This summary provides a human-readable overview of the software and system environment.
For full replication, recreate the Python environment using requirements.txt and rerun generate_env_info_auto.py to regenerate appendix_environment_concise.txt. If discrepancies arise, the machine-readable environment file and pinned dependency list are authoritative.