



Research article

Reimplementation and performance characterization of the PE-AONT schema for efficient security and retrieval of confidential data at the edge

Roberto Vergallo ^a, Simone Casciaro ^{b,*}, Luca Mainetti ^b

^a *Pegaso Digital University, Department of Information Science and Technology, Piazza Trieste e Trento, 48, Naples, 80132, Campania, Italy*

^b *University of Salento, Department of Innovation Engineering, Via per Monteroni, 165, Lecce, 73100, Apulia, Italy*

ARTICLE INFO

Keywords:

AONT
PE-AONT
Fragmentation
Dispersal storage systems
Internet of things
Edge computing
Cybersecurity

ABSTRACT

PE-AONT (Partial Encryption All or Nothing Transform) represents the most recent iteration (2022) of an algorithm designed to fragment and disperse data over different independent servers or storage sites. It combines encryption and non-encryption techniques to achieve exposure protection and faster processing. However, these claims have yet to be confirmed, partly due to the lack of publicly available source code. In this work, we present the first publicly available Java reimplementation of PE-AONT, built on Docker containers for reproducibility, and evaluate it on an IoT network comprising five Raspberry Pi nodes. In addition to providing evidence of markedly different performance characteristics in our setup (26.11 – 37.92 MB/s vs. the claimed 220 – 310 MB/s), we extend the scheme by adding a previously missing data reconstruction service and an exhaustive performance characterization across a range of configurations. We release the full PE-AONT implementation in the open-science spirit.

1. Introduction

Computer systems comprising numerous nodes are vulnerable to data loss and tampering, especially when they rely on devices with limited hardware capabilities. In this context, Dispersal Storage Systems (DSS) have attracted significant interest, since they offer an effective strategy for mitigating risks by enhancing the overall confidentiality, integrity, and availability of the system.

Among these, the PE-AONT (Partial Encryption All-Or-Nothing Transform) schema proposed by Kapusta & Memmi [1] aims to strike a balance between security and performance. Previous approaches typically apply full encryption combining with AONT [2] to the entire dataset. In contrast, PE-AONT applies partial encryption, yielding faster processing ranging rates from 220 MB/s to 310 MB/s depending on the parameters used. However, several open issues arise when analyzing Kapusta & Memmi's paper, in particular: (i) the lack of detail concerning the experimentation, and (ii) concerning the reconstruction of distributed messages; (iii) the absence of corroborating evidence for the claimed performance, particularly on resource-constrained devices. Moreover, the absence of open-source implementations limits the adoption by industry. The currently available repositories (<https://github.com/nsg21/aont>, <https://pkg.go.dev/github.com/1AD8BE38F271/vpncore/enc/aont>, and <https://github.com/declanmalone/aont>) are outdated (2014, 2016, 2021) and lack dispersal storage, partial encryption, and a production-grade level of maturity.

* Corresponding author.

E-mail addresses: roberto.vergallo@unipegaso.it (R. Vergallo), simone.casciaro@unisalento.it (S. Casciaro), luca.mainetti@unisalento.it (L. Mainetti).

<https://doi.org/10.1016/j.iot.2026.102048>

Received 26 January 2026; Received in revised form 18 June 2026; Accepted 29 July 2026

Available online 30 July 2026

2542-6605/© 2026 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

The aim of this work is the reimplementaion and performance characterization of the PE-AONT schema. We implement it within a Java-based architecture and deploy it on a set of Raspberry Pi nodes. Our experiments confirm that PE-AONT can effectively secure data within the distributed system, and that the throughput trend reported by Kapusta & Memmi is faithfully reproduced on our edge testbed. However, our setup exhibits substantially lower throughput, the magnitude of which depends on the parameters used. We release the full source code on GitHub in an open-science spirit [Rep].

The remainder of this paper is organized as follows. Section 2 introduces the PE-AONT scheme and situates our work with respect to the state of the art. Section 3 details the experimental setup of our testbed and the threat model. Section 4 presents the results and compares them with findings reported in the literature. Finally, Section 5 concludes the paper and identifies open research challenges.

2. Distributed storage system

In this section, we first introduce the AONT family and the PE-AONT scheme, and then position our contribution with respect to related approaches in transformation-based algorithms, lightweight cryptography, and trusted-execution storage.

2.1. AONT and PE-AONT schemes

AONT, originally proposed in 1997, can be viewed as an $(s + 1, s + 1)$ threshold scheme. In this setting, input data consisting of s words of size w_A^1 is transformed into $s + 1$ output words such that none of the original words can be recovered unless all $s + 1$ transformed words are available, or the attacker is capable of breaking the underlying encryption.

In recent years, several efforts have been made to improve the efficiency of AONTs, aiming to preserve strong security guarantees while reducing computational and storage overhead. For example, AONT-LT [4] (2014) is a variant of AONT-RS [5] (2011) that replaces the Reed-Solomon (RS) encoder with a Luby Transform (LT) encoder [6]. Other notable developments include a 2015 proposal that combines the AONT transformation with chaos-based encryption [7], image-encryption [8], and the 2019 AONT-NZZD scheme [9], which incorporates a New ZigZag-Decodable code into the AONT framework.

The most recent advancement, proposed in 2022, is the PE-AONT. Unlike previous approaches that combine the AONT with full encryption, error-correcting codes or chaos-based encryption, PE-AONT integrates the AONT transformation with partial encryption and dispersal storage. In this model, only a subset of the data fragments is encrypted prior to dispersion. The Advanced Encryption Standard (AES) is adopted as the encryption algorithm.

In PE-AONT, data are organized into a $k \times f$ matrix C , where the k rows correspond to the k blocks and the f columns correspond to the f fragments within a block (k is an even number with a minimum value of k of 4):

$$C = \begin{pmatrix} \text{IV} & C_{0,1} & C_{0,2} & \cdots & C_{0,f-1} \\ C_{1,0} & C_{1,1} & C_{1,2} & \cdots & C_{1,f-1} \\ C_{2,0} & C_{2,1} & C_{2,2} & \cdots & C_{2,f-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ C_{k-1,0} & C_{k-1,1} & C_{k-1,2} & \cdots & C_{k-1,f-1} \end{pmatrix}$$

The total number of cells is therefore $k \cdot f$, where $C_{0,0}$ is reserved for the Initial Vector (IV) required by the encryption algorithm, according to Kapusta & Memmi's paper. Partial encryption is applied to only $e < k$ blocks. Once the partially encrypted blocks are obtained, the AONT transformation is applied, followed by the distribution phase. When $e = k - 1$, the protection level is high; when $e \leq k - 2$, the protection level is lower in exchange for better performance.

The general pseudocode of our implementation is shown in Algorithms 1 and 2. Each row of C is treated as the atomic unit of distribution: the row $(C_{i,0}, C_{i,1}, \dots, C_{i,f-1})$ is encapsulated in a single object C_i , and the block as a whole is sent to a single peer storage node. The block-to-node assignment is deterministic and follows the round-robin rule $\text{node}(i) = p_i \bmod p$, where p is the number of physical nodes in the cluster. The encryption policy is fixed by lines 3–9 of Algorithm 1. After the encryption pass, lines 10–15 of Algorithm 1 apply the AONT XOR pass to the whole matrix column by column regardless of whether it carried plaintext or ciphertext immediately before.

The transformation is self-inverse, that is, it is its own inverse. After having collected the blocks $C_0, C_1, \dots, C_{k-1}$, the entry point runs Algorithm 2 on the resulting blocks. The inverse AONT pass of lines 1–6 of Algorithm 2 undoes the column-wise XOR introduced by lines 10–15 of Algorithm 1. At line 7, the Initial Vector is recovered from $C_{0,0}$. The partial decryption loop of lines 8–14 then applies $\text{AES-CBC}_{K, \text{IV}}^{-1}$ to the same set of cells that the Algorithm 1 had encrypted, restoring their plaintext. Finally, line 15 concatenates the cells of C in row-major order (skipping $C_{0,0}$, which carries the IV), removes the per-fragment padding introduced by line 1 of Algorithm 1, and returns the reconstructed plaintext.

2.2. Relation to prior work

Applying a data-transformation algorithm prior to data distribution in a DSS can improve confidentiality by preventing compromised nodes from accessing to original information. Chou & Kliever [10] encode data into multiple shares so that the reconstruction phase requires a sufficient number of storage servers. Nabawy et al. [11] combine data fragmentation with multi-cloud distribution to mitigate the risk of insider attacks and unauthorized access. Transformation-based protection has also been applied to sensitive data templates. Sardar et al. [12] combine a non-invertible FaceHashing transformation with cryptographic operations to reduce the risk of template reconstruction after a partial data leak.

Algorithm 1 PE-AONT forward transform.

Require: plaintext M , parameters k (even), f , e ($e < k$), key K , IV

Ensure: blocks $C_0, C_1, \dots, C_{k-1}$, with $C_i = (C_{i,0}, \dots, C_{i,f-1})$

- 1: Split M into $l = k \cdot f - 1$ equal-sized fragments; pad to a common length
- 2: Place IV at $C_{0,0}$ and the l fragments at $C_{i,j}$ for $(i, j) \neq (0, 0)$ in row-major order
- 3: **for** $i \leftarrow 0$ to $e - 1$ **do**
- 4: **for** $j \leftarrow 0$ to $f - 1$ **do**
- 5: **if** $(i, j) \neq (0, 0)$ **then**
- 6: $C_{i,j} \leftarrow \text{AES-CBC}_{K, \text{IV}}(C_{i,j})$
- 7: **end if**
- 8: **end for**
- 9: **end for**
- 10: **for** $j \leftarrow 0$ to $f - 1$ **do**
- 11: $\text{sum} \leftarrow \bigoplus_{i=0}^{k-1} C_{i,j}$
- 12: **for** $i \leftarrow 0$ to $k - 1$ **do**
- 13: $C_{i,j} \leftarrow \text{sum} \oplus C_{i,j}$
- 14: **end for**
- 15: **end for**
- 16: **for** $i \leftarrow 0$ to $k - 1$ **do**
- 17: $C_i \leftarrow (C_{i,0}, C_{i,1}, \dots, C_{i,f-1})$ ▷ row i encapsulated as block C_i
- 18: **end for**
- 19: **return** blocks $C_0, C_1, \dots, C_{k-1}$

Algorithm 2 PE-AONT inverse transform.

Require: blocks $C_0, C_1, \dots, C_{k-1}$ with $C_i = (C_{i,0}, \dots, C_{i,f-1})$, viewed as the rows of a $k \times f$ matrix C (with k even); parameters f , e ($e < k$), key K

Ensure: plaintext M

- 1: **for** $j \leftarrow 0$ to $f - 1$ **do**
- 2: $\text{sum} \leftarrow \bigoplus_{i=0}^{k-1} C_{i,j}$
- 3: **for** $i \leftarrow 0$ to $k - 1$ **do**
- 4: $C_{i,j} \leftarrow \text{sum} \oplus C_{i,j}$
- 5: **end for**
- 6: **end for**
- 7: $\text{IV} \leftarrow C_{0,0}$
- 8: **for** $i \leftarrow 0$ to $e - 1$ **do**
- 9: **for** $j \leftarrow 0$ to $f - 1$ **do**
- 10: **if** $(i, j) \neq (0, 0)$ **then**
- 11: $C_{i,j} \leftarrow \text{AES-CBC}_{K, \text{IV}}^{-1}(C_{i,j})$
- 12: **end if**
- 13: **end for**
- 14: **end for**
- 15: Concatenate $C_{i,j}$ for $(i, j) \neq (0, 0)$ in row-major order, remove padding, return M

This work intersects three lines of research. First, post-2022 developments have combined AONT with residual number systems, attribute-based encryption, and proxy re-encryption [13,14], showing how AONT can be leveraged to achieve confidentiality and security in the cloud-storage domain. Second, lightweight cryptography has become increasingly relevant in IoT and Edge systems [15,16]; partial-encryption algorithms such as PE-AONT can offer resource-saving benefits compared with full-encryption systems. Third, confidential computing solutions primarily ensure data security through hardware components such as Trusted Execution Environments (TEEs) and attestation mechanisms [17]. PE-AONT does not necessarily require trusted hardware, which may be unavailable owing to computational constraints or prohibitive production costs. Data confidentiality is instead inherent in the All-Or-Nothing transformation itself.

3. Experimental setup

We designed a three-stage pipeline. The first stage specified the high-level workflow within the distributed system. The second stage defines and evaluates the threat model for the architecture introduced in the first stage. The third stage defines the metrics

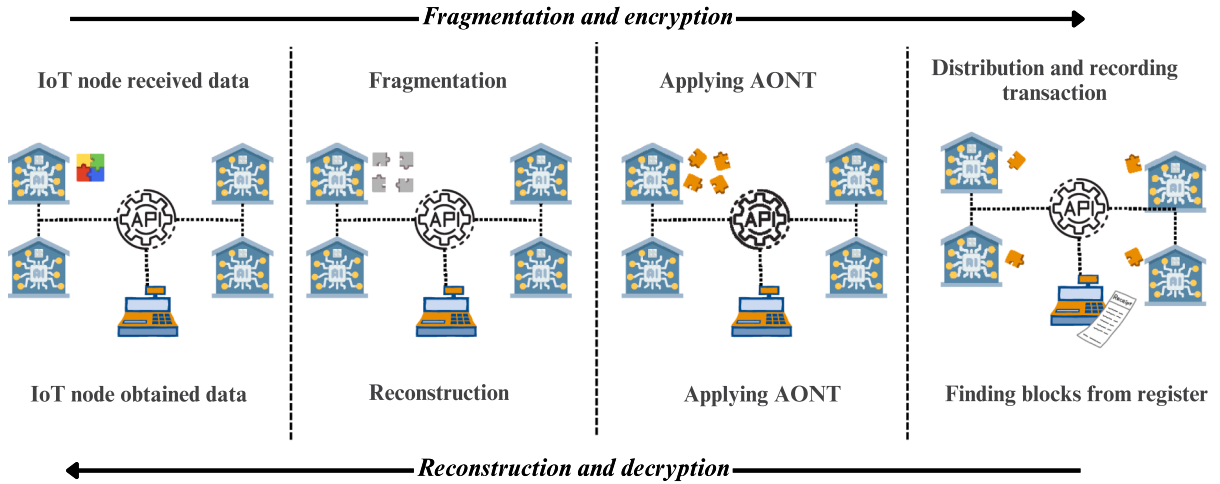


Fig. 1. Information workflow for a DSS based on the PE-AONT schema.

of interest and develops the APIs required for their collection. This section also reports the main implementation details in order to support reproducibility and enable an independent experimental evaluation of the proposed system.

The following subsections describe each stage in detail.

3.1. API-based information workflow

After the user uploads a data object via any node, the workflow for storing data through PE-AONT proceeds as follows: (i) Fragmentation; (ii) Partial encryption; (iii) Application of the AONT transformation; and (iv) Registration of fragment metadata into a “registry node”; (v) Distribution of fragments across the nodes. Conversely, retrieving the original data from any node includes: (i) Querying the registry node for fragment locations; (ii) Retrieving the fragment blocks; (iii) Reapplying the AONT transformation; (iv) Partial decryption; and (v) Concatenation of the original fragments.

Fig. 1 depicts the information workflow. Specifically, during the data ingestion phase (left-to-right arrow), PE-AONT generates the fragments that are distributed across the nodes. In the data retrieval phase (right-to-left arrow), upon request, the system collects all fragments, reapplies the AONT transformation, decrypts the necessary parts, and reconstructs the original data. As a further enhancement over the original PE-AONT implementation, we introduce an additional node that functions as a registry for the blocks. It stores metadata mapping each original object to its distributed fragments, thereby enabling the reconstruction process.

To interact with the PE-AONT based DSS within a RESTful architecture, each node exposes at least two REST APIs: (i) a POST endpoint, accepting either JSON or Multipart Form Data payloads, for data fragmentation, partial encryption, AONT transformation, and fragment distribution; and (ii) a GET endpoint, requiring an OBJECT identifier to retrieve and reconstruct a stored data item. This involves block retrieval, AONT application, partial decryption, and fragment reassembly. The OBJECT identifier is chosen by the client invoking the POST, and is retained for future use. This detail represents a novel contribution relative to Kapusta & Memmi’s design, which provides no reconstruction service.

3.2. Threat model

We assume that each node executes the PE-AONT algorithm and holds the IV and the key K . All communications take place on an isolated Wi-Fi network in which all nodes know each other’s IP addresses. The generation and distribution of the IV and the key K among the nodes, as well as the secure building of the network, are outside the scope of this study.

Based on the model of Kapusta & Memmi, we consider the following adversaries: (i) a passive network adversary that observes all traffic but cannot inject or modify the fragments; (ii) an honest-but-curious adversary that compromises one or more storage nodes (the attacker reads the fragments stored there but does not deviate from the protocol); and (iii) a compromise of the registry node. The security goals of our implementation are summarised as follows.

Against (i), the system ensures plaintext confidentiality. The transmitted data have already passed through the PE-AONT and partial encryption operations, so the original plaintext is unreadable by All-Or-Nothing property. Against (ii), the system provides indistinguishability under partial fragment exposure. An adversary that recovers strictly fewer than $k \cdot f$ fragments learns nothing about the data, by definition of the All-Or-Nothing property, which ensures that fragments cannot be distinguished under partial data leakage. Even an adversary that holds $k \cdot f$ fragments cannot read the $e \cdot f - 1$ encrypted fragments without the key K .

Against (iii), the system ensures the confidentiality of the plaintext, but does not guarantee its availability. A registry node compromise leaks only fragment locations inside the network, so neither plaintext nor ciphertext is exposed. The registry node stores

only an ordered list of fragment identifiers and their storage locations. However, the unavailability of the registry node causes the reconstruction process to fail, since we do not provide registry node redundancy or a copy of registry table on the compute nodes.

The current implementation does not provide cryptographic integrity or authenticated availability. A single node that returns a corrupted fragment, or fails to return one, causes an incorrect reconstruction or a failure reconstruction pipeline.

A detection mechanism and a recovery policy sit on top of the PE-AONT algorithm and are out of scope of this study. Our implementation evaluates the feasibility of running the PE-AONT algorithm on a network of constrained devices such as Raspberry Pi, focusing on confidentiality under partial fragment exposure and on the performance impact of fragmentation, partial encryption, and distribution overhead.

3.3. PE-AONT evaluation

Following the experimental methodology proposed by Lopez & Alhulayil [18], this subsection provides a detailed description of the implementation choices and platform constraints. The system consists of several components, which are defined, explained, and isolated in order to obtain fine-grained results.

To validate the performance of the PE-AONT schema, we implemented an experimental IoT architecture consisting of five nodes – four computational nodes and one registry node –, each of which is a Raspberry Pi 4 device. The choice of four nodes aligns with the experimental setup proposed by Kapusta & Memmi: the minimum value of k is 4. In our implementation, we modified the fragment-to-node placement policy by adding a round-robin distribution. This allows us to simulate a system with a larger number of nodes than is physically available. The e fragments to be encrypted are selected deterministically in our implementation: the first $e \cdot f - 1$ fragments in the original split order are encrypted.

Their implementation relied on PCs running Windows 7, equipped with Intel Core i7 processors and 8 GB of RAM. In contrast, our Raspberry Pi devices run the Raspberry Pi OS 64-bit operating system and are equipped with a quad-core Cortex-A72 (ARM v8) 64-bit SoC 1.5GHz and 4 GB of RAM. In their work, Java 1.8 was selected as the programming language. We used instead a more recent version (OpenJDK 17) to support modern deployments. The reference paper and our implementation both use the AES algorithm with a 128-bit key. However, while Kapusta & Memmi rely on AES-NI hardware acceleration available on Intel CPUs, our Raspberry Pi does not expose the ARM Crypto Extensions owing to architectural constraints. To prevent potential thermal throttling, passive heat sinks were installed on every Raspberry Pi. A Tapo P125M smart plug was employed to track power consumption.

We adopted Docker to favour portability and reproducibility across heterogeneous environments. Five containers – one per node – were instantiated and connected via Wi-Fi (802.11n). Each container runs the Java archive (JAR) implementing PE-AONT and is co-located via Docker Compose with a MongoDB instance, which is used for local fragment storage on each node. In Kapusta & Memmi's work, no information is disclosed regarding network connectivity. In our setup, wireless connectivity was chosen over Ethernet to mimic real deployment conditions of IoT nodes. The Wi-Fi network is isolated from the Internet to avoid external interference. For consistency, all nodes were physically located in close proximity, and the average Wi-Fi signal-to-noise ratio (SNR) was approximately 42 dB.

To measure the impact of each component of the architecture and algorithm, we built a Python script with six modes:

- **Core-mode** measures fragmentation, partial encryption and AONT transformation. The reported throughput is network-free and storage-free by construction (no network latency, HTTP serialization/deserialization cost, database I/O, containerization overhead);
- **Full-encryption** is the Core-mode baseline with $e = k$ (all fragments are encrypted). It includes AES and AONT operations and serves as the baseline for measuring the speedup achievable through partial encryption;
- **Full-mode** performs the same operations as Core-mode but additionally accounts for all the other architectural components (database I/O, network latency, HTTP serialization/deserialization, containerization overhead);
- **Full-encryption-full** is the Full-mode baseline with $e = k$ (all fragments are encrypted). It represents the baseline for measuring the speedup achievable through partial encryption on the complete pipeline;
- **Breakdown** is a fine-grain decomposition of Core-mode into phase timings (split, AES, assembly, AONT);
- **AES-baseline** is a pure AES-128 loop on the payload (no fragmentation, no AONT, no storage, no network) using `-XX:+UseAES` and `-XX:+UseAESIntrinsics` enabled at Java Virtual Machine (JVM) start-up. It establishes the platform's intrinsic AES ceiling.

All modes were run on a single node, `Node01`, which represented the entry point during the benchmarks. The JVM log files were analyzed to quantify resource usage on the Raspberry Pi devices, Garbage Collection (GC) pauses, and the JIT effect.

In our experiments, we extend the methodology adopted by Kapusta & Memmi to better characterize the performance of PE-AONT. For each mode, we run a sweep benchmark using $e \in [2, 3, \dots, 7]$ and $k \in [4, 6, 8]$. As reported by Kapusta & Memmi, the minimum value of k is 4. The selection of the parameters e and k therefore requires a thorough analysis of both the security requirements to be met and the performance criteria.

To reduce the impact of network fluctuations and local background processes, each configuration was cleanly installed, warmed up to properly initialize the JVM before measurements, and then tested with 50 iterations for each payload size; the results were then averaged over the iterations.

A plain text random data sample was used by Kapusta & Memmi for each measurement; the size of such plaintext is, however, not disclosed. We instead conducted a ramping test including two kinds of payloads: (i) a plain text of varying size from *1kB* to *10MB*, and (ii) an executable file, also ranging from *1kB* to *10MB*. The executable file prints a simple "Hello PE-AONT!" message to the

console when launched. The executable file is executed as an integrity check after the reconstruction. We include this payload type because it may inspire future research.

Captured performance metrics (processing time and throughput, along with input size) are appended to JSON Lines (JSONL) files. A Python script is then used to parse the results and generate the performance plots. We release both the benchmark scripts and the corresponding log files in the project repository.

3.4. Implementation details for reproducibility

To promote independent experimental evaluation and describe our runtime environment, this subsection reports the technical specifications of our implementation:

- **Application Layer.** SpringBoot 3.4.4 with embedded Tomcat 10.1. Inter-node communication uses Spring's `RestTemplate` on a isolated Wi-Fi network. The maximum multipart/form-data payload size in SpringBoot is 10 MB.
- **Storage.** Each node runs an independent MongoDB 4.0 instance. The `Block` collection on Raspberry Pi nodes stores documents of the form `{_id: String, f: int, block: BinData []}`, indexed only on default `_id`. The `Register` collection on the registry node stores `{_id: ObjectId, object: String, fragments: [{nodeId, fragmentId}]}`. The maximum document size in MongoDB is 16 MB.
- **JVM configuration.** OpenJDK 17 with the G1 garbage collector and the default C1 + C2 JIT. By default, the JVM uses an initial heap of 64 MB and a maximum heap of 1 GB, corresponding respectively 1/64 and 1/4 of the physical RAM on the Raspberry Pi 4. The `docker-compose` development configuration enables flags `-XX:+UseAES -XX:+UseAESIntrinsics` explicitly.
- **Fragment-to-node placement policy.** The placement of fragments is deterministic and hardcoded. Let p denote the number of physical compute nodes available (in our setup $p = 4$, matching the Kapusta & Memmi's setup). Row i of the matrix C is sent to the node $i \bmod p$ at the static IP assigned to that node. When $k > p$, the same storage node receives more than one fragment. The maximum number of fragments stored on a single node is $k - 2$ when $e = k - 1$ as suggested by Kapusta & Memmi. This placement policy allows us to simulate a system with a larger number of nodes than the physical devices available. If the number of fragments to be distributed exceeds the number of physical devices, a round-robin distribution is applied across the available nodes.
- **Payload encoding.** JSON data remains in text from end to end: the client sends a JSON whose payload is a UTF-8 string, and the server stores it as `String` in MongoDB. Multipart/form-data are converted to a JSON envelope, using Base64 encoding, before begin passed to the fragmentation algorithm. This choice expands the payload length by approximately 33%.
- **Failure handling.** If any of the k fragment fetches during reconstruction fails or overrides Spring's default `RestTemplate` timeouts, the operation aborts and returns HTTP 500. No partial reconstruction is attempted, consistent with the All-Or-Nothing semantics. We have chosen to implement the system as safe-fail by design.

4. Results and discussions

4.1. Performance characterization

Fig. 2 plots the system throughput aggregated over all PE-AONT configurations (e, k) and payload type (JSON and multipart/form-data) for `Core-mode` and `Full-mode`. The throughput exhibits an approximately linear increase with the payload size (the x scale of Fig. 2 is not linear, while the y scale is logarithmic). However, as the parameter e increases, the performance decreases due to the higher computational effort required for block encryption, while the overall security level improves.

Table 1 provides a more detailed statistical analysis of the results obtained. `Full-mode`'s performances is approximately $\sim 4\%$ of the `Core-mode` throughput with a 10 MB payload (2.01 vs 46.25 MB/s) and $\sim 3\%$ with a 500 kB payload (1.41 vs 47.76 MB/s), highlighting the crucial role of the architectural setup and overhead. The Δ Convergence column reports the percentage difference between the means of the first and second halves of the measurements. `Core-mode` values remain below 3% for every payload size, whereas `Full-mode` values are subject to component overhead (MongoDB and network) that impacts the measurements at smaller payloads.

Please note that the results presented in this paper have been aggregated across the various configurations to aid readability. A detailed analysis, breakdown by configuration, is publicly available in the GitHub repository [Rep].

Figs. 3 and 4 show the system throughput isolated for all PE-AONT configurations (e, k) and payload size for `Core-mode` and `Full-mode`, aggregated over payload type (JSON and multipart/form-data). The best throughput performance is 72.72 MB/s, achieved with $k = 8; e = 2$ and a payload size of 500 kB for `Core-mode` (Fig. 3), and 2.26 MB/s, achieved with $k = 6; e = 2$ and a payload size of 10 MB for `Full-mode` (Fig. 4). A direct comparison with the performance achieved by Kapusta & Memmi – although the authors do not specify the payload size or quantify the obtained throughput – is shown in Table 2. Kapusta & Memmi measured throughput only for the pairs $(k = 4, e = 2), (k = 6, e = 4), (k = 4, e = 3), (k = 6, e = 5)$. The ranking of these configurations in our experiments matches the ranking reported by the authors.

The absolute difference in throughput between our results and those of Kapusta & Memmi can be attributed to hardware differences, the limited information available on the reference setup, and the missing information on the payload size. We attribute this gap to two platform differences. First, the Cortex-A72 of the Raspberry Pi 4 runs at 1.5 GHz clock, while Kapusta & Memmi report a Intel Core i7 with a 2.7 GHz clock. The CPU-clock difference alone contributes for 2 times factor to any compute-bound operation. Second, the SoC architecture of the Cortex-A72 does not expose the ARM Crypto Extensions, whereas Kapusta & Memmi used AES-NI hardware acceleration. These differences led to a 1:8 ratio between our results and those of Kapusta & Memmi.

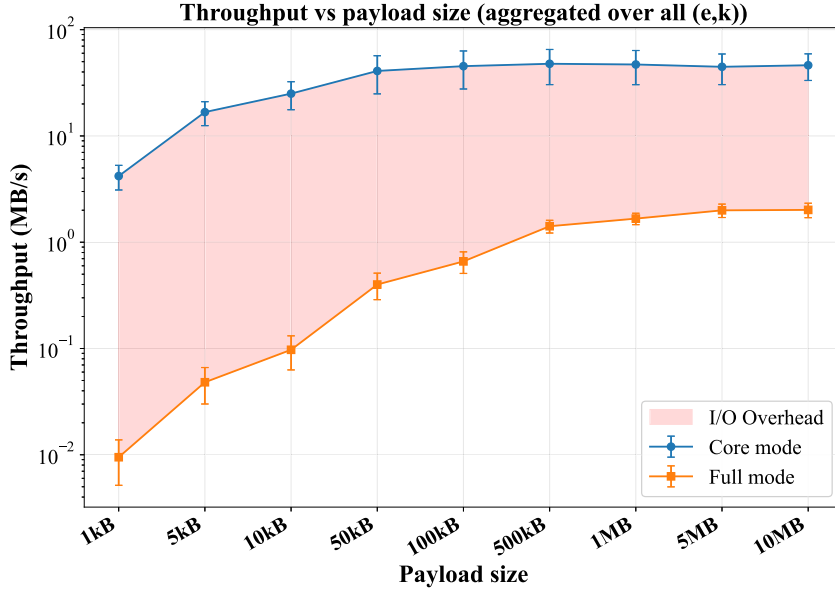


Fig. 2. System throughput (MB/s) versus payload size for the Core-mode and Full-mode, aggregated over all PE-AONT configurations (k, e) and payload type (JSON and multipart/form-data). Notice that the x scale is not linear, while the y scale is logarithmic. The red shaded region shows the gap between Core-mode (fragmentation, partial encryption and AONT transformation) and Full-mode (Core-mode, database I/O, network latency, HTTP serialization/deserialization, containerization overhead).

Table 1

System throughput (MB/s) versus payload size for the core-mode and full-mode, aggregated and averaged over all PE-AONT configurations (k, e) and payload type (JSON and multipart/form-data). The table shows: payload size and mode (core-mode or full-mode); mean throughput and standard deviation σ in MB/s, minimum and maximum throughput; 99th and 50th percentile of the throughput distribution; Δ Convergence%. Low percentage values of Δ Convergence% indicate a steady-state system.

Payload (Mode)	Thr- $\pm\sigma$ (MB/s)	Min	Max	p99	p50	Δ Conv%
1kB (Core-mode)	4.19 $\pm$ 1.09	0.58	5.71	5.49	4.55	2.91
5kB (Core-mode)	16.75 $\pm$ 4.25	1.38	25.63	25.21	17.05	2.59
10kB (Core-mode)	25.01 $\pm$ 7.36	6.18	42.76	42.23	24.94	2.89
50kB (Core-mode)	40.88 $\pm$ 15.99	6.35	85.47	82.27	39.56	1.69
100kB (Core-mode)	45.35 $\pm$ 17.70	6.84	98.00	93.48	44.21	0.22
500kB (Core-mode)	47.76 $\pm$ 17.35	12.42	101.85	100.24	44.04	0.76
1MB (Core-mode)	47.02 $\pm$ 16.65	11.48	103.58	100.35	43.80	0.88
5MB (Core-mode)	44.73 $\pm$ 14.31	23.69	99.99	95.77	40.84	1.28
10MB (Core-mode)	46.25 $\pm$ 12.98	13.93	89.69	82.39	43.14	0.49
1kB (Full-mode)	0.01 $\pm$ 0.00	0.00	0.02	0.02	0.01	15.45
5kB (Full-mode)	0.05 $\pm$ 0.02	0.02	0.11	0.09	0.04	13.51
10kB (Full-mode)	0.10 $\pm$ 0.03	0.03	0.19	0.18	0.09	10.66
50kB (Full-mode)	0.40 $\pm$ 0.11	0.05	0.66	0.63	0.39	4.06
100kB (Full-mode)	0.66 $\pm$ 0.15	0.22	0.98	0.94	0.66	7.23
500kB (Full-mode)	1.41 $\pm$ 0.19	0.45	1.86	1.82	1.42	0.96
1MB (Full-mode)	1.67 $\pm$ 0.20	0.30	2.14	2.08	1.69	1.01
5MB (Full-mode)	2.00 $\pm$ 0.29	0.18	2.45	2.39	2.04	2.58
10MB (Full-mode)	2.01 $\pm$ 0.31	0.26	2.53	2.47	2.07	0.06

Although the binary file (multipart/form-data payload) exhibits lower entropy than the random text (JSON payload), the two payload types produce comparable throughput. This behavior can be attributed to the PE-AONT design: the computational cost is dominated by bitwise operations (AES and XOR) whose runtime is independent of the input entropy.

The choice between JSON and multipart/form-data impacts the throughput (Table 3). The throughput metric is defined as follows:

$$\text{Throughput} = \frac{\text{input size (bytes)}}{\text{run time (ms)}} \quad (1)$$

The multipart request handler Base64 encodes the incoming bytes and wraps them in a JSON envelope before calling the fragmentation routine, so the cryptographic core processes $\sim 4N/3$ bytes on the multipart path against N bytes on the JSON path. The

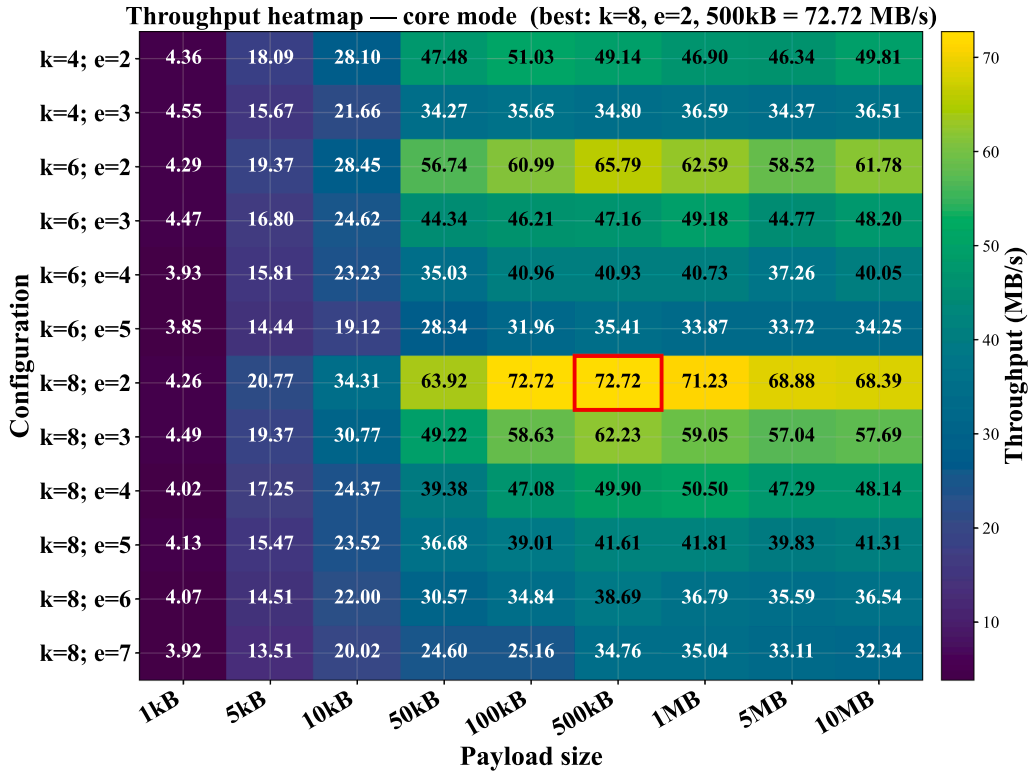


Fig. 3. System throughput (MB/s) versus payload size and configurations (k, e) for Core-mode aggregated and averaged over payload type (JSON and multipart/form-data). The cell with red border is the best overall configuration in terms of throughput.

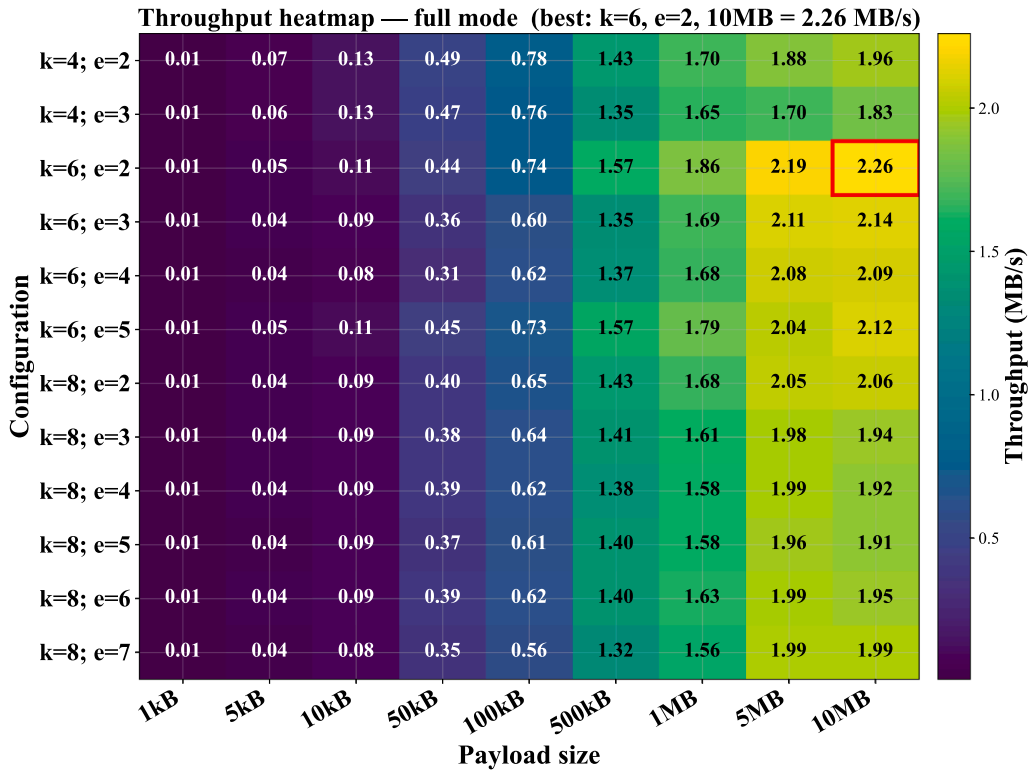


Fig. 4. System throughput (MB/s) versus payload size and configurations (k, e) for Full-mode aggregated and averaged over payload type (JSON and multipart/form-data). The cell with red border is the best overall configuration in terms of throughput.

Table 2

Direct comparison ranking between Kapusta & Memmi's paper and our Core-mode implementation on the same configuration (k, e) aggregated and averaged over payload size and type (JSON and multipart/form-data). Table shows: configuration (k, e); The throughput and standard deviation σ in (MB/s); Ranking for each configuration is the same in Kapusta & Memmi's paper and our results.

Config (k, e)	Thr. $\pm\sigma$ (MB/s)	Our rank	Kapusta & Memmi's rank
$k=4, e=2$	37.92 ± 18.51	1	1
$k=6, e=4$	30.88 ± 14.33	2	2
$k=4, e=3$	28.23 ± 12.68	3	3
$k=6, e=5$	26.11 ± 12.09	4	4

Table 3

Direct comparison of the per-request server-side throughput between the JSON and multipart/form-data payload types, aggregated and averaged over all (k, e) configurations. The table shows: mode, payload size, JSON and multipart/form-data throughput in MB/s and run time in ms.

Mode	Payload	JSON Thr. (MB/s)	Multipart Thr. (MB/s)	JSON time (ms)	Multipart time (ms)
Core-mode	1kB	3.96	4.43	0.29	0.32
	5kB	17.79	15.72	0.30	0.47
	10kB	26.94	23.09	0.41	0.63
	50kB	46.44	35.32	1.30	2.17
	100kB	50.88	39.82	2.38	3.84
	500kB	54.09	41.43	10.35	17.25
	1MB	53.60	40.45	21.09	35.91
	5MB	50.28	39.18	108.78	179.09
Full-mode	10MB	51.39	41.11	208.22	341.28
	1kB	0.01	0.01	176.58	110.41
	5kB	0.03	0.06	149.14	112.77
	10kB	0.07	0.12	144.85	113.01
	50kB	0.32	0.48	157.66	147.44
	100kB	0.55	0.77	184.86	173.33
	500kB	1.30	1.53	381.75	432.00
	1MB	1.57	1.77	654.04	763.33
	5MB	1.94	2.06	2632.31	3548.97
	10MB	1.93	2.10	5517.00	6523.22

Table 4

Core-mode cost decomposition at PE-AONT configuration $k=4, e=2$ over payload size aggregated and averaged over payload type (JSON and multipart/form-data). The table shows: payload size; mean phase time in milliseconds and its share of the core time: fragmentation, AES encryption of the e selected blocks, re-serialization, and the AONT XOR pass; total time.

Payload	Frag.	AES	Re-serialize	AONT XOR	Total (ms)
1kB	0.01 (5.2%)	0.05 (39.8%)	0.06 (49.6%)	0.01 (5.4%)	0.13
5kB	0.01 (4.8%)	0.10 (53.2%)	0.07 (35.8%)	0.01 (6.2%)	0.18
10kB	0.01 (5.2%)	0.17 (61.4%)	0.07 (26.3%)	0.02 (7.0%)	0.27
50kB	0.10 (9.5%)	0.72 (67.2%)	0.10 (9.8%)	0.14 (13.5%)	1.07
100kB	0.09 (5.1%)	1.40 (78.2%)	0.13 (7.4%)	0.17 (9.3%)	1.79
500kB	0.57 (6.6%)	6.73 (77.4%)	0.45 (5.1%)	0.95 (10.9%)	8.70
1MB	1.40 (7.3%)	15.06 (78.2%)	0.81 (4.2%)	1.99 (10.3%)	19.26
5MB	5.41 (6.0%)	67.72 (74.7%)	2.95 (3.3%)	14.63 (16.1%)	90.70
10MB	10.16 (6.0%)	129.24 (76.1%)	4.93 (2.9%)	25.45 (15.0%)	169.78

server-side throughput metric also uses the size of this inflated envelope as its input size numerator, so the multipart numerator is itself inflated by $\sim 33\%$.

In Core-mode, we measure fragmentation, partial encryption and AONT transformation. Here, the run time scales nearly linearly with the number of bytes actually processed, so the $\sim 33\%$ inflation introduced by the Base64 encoding on the multipart path translates into a comparable increase in run time. Since this increase in run time outweighs the corresponding increase in the numerator of Eq. (1), the throughput of multipart/form-data is approximately 30% lower than that of JSON.

In Full-mode, the denominator of Eq. (1) also includes database I/O, network latency, HTTP serialization/deserialization over the four nodes and containerization overhead. These components are largely independent of payload format and dominate the run time, so the $\sim 33\%$ inflation of the numerator is not matched by a proportional inflation of the denominator. As a result, handling multipart/form-data is about 10% faster than JSON in Full-mode.

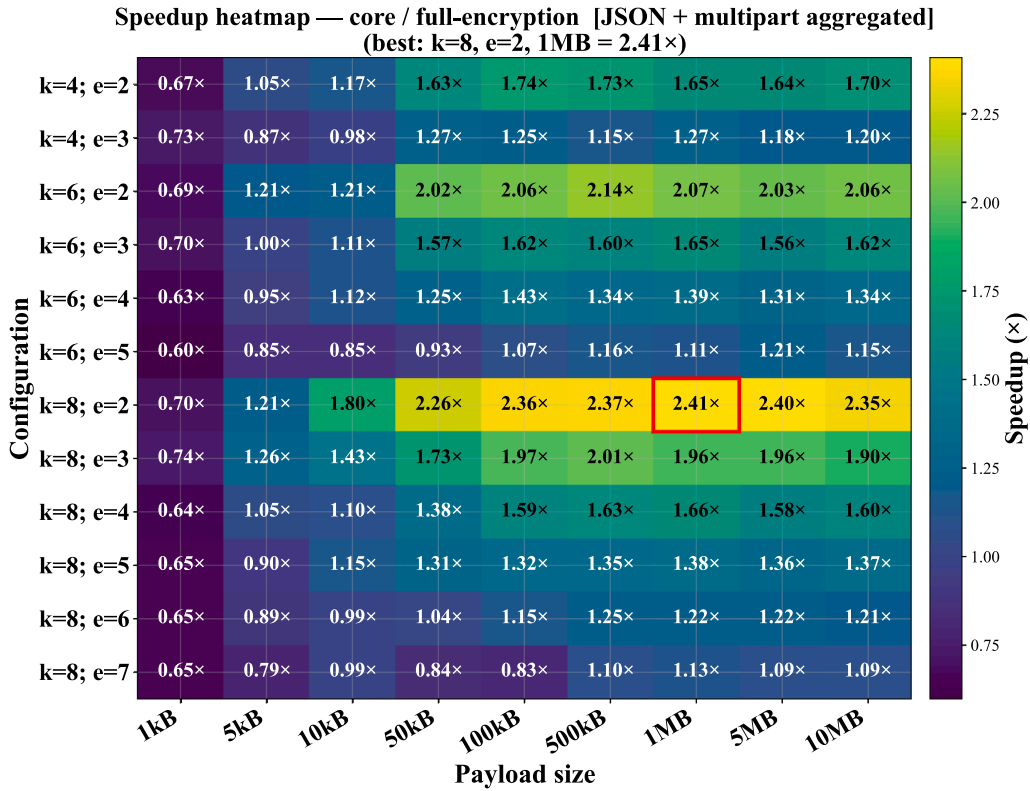


Fig. 5. System speedup over Full-encryption baseline in Core-mode versus payload size and configurations (k, e) aggregated and averaged over payload type (JSON and multipart/form-data). The cell with red border is the best overall configuration in terms of speedup.

This sign reversal therefore does not reflect an intrinsic difference in efficiency between the two formats, but rather the relative weight of the fixed numerator inflation against the variable denominator inflation, which in turn depends on which system component dominates the overall run time.

Partial encryption of fragments is expected to offer a throughput advantage over full encryption. To quantify this advantage, we compared the throughput of Core-mode and Full-mode with their respective baselines obtained from the Full-encryption and Full-encryption-full mode. These baselines are obtained by forcing the system to encrypt all k blocks ($e = k$). Figs. 5 and 6 plot the speedup for each payload size and configuration, respectively, for Core-mode and Full-mode. In both heatmaps, the speedup increases with the payload size but decreases as the number of fragments to be encrypted increases. This behavior is not surprising: as the number of fragments to be encrypted increases, so does the computational load. The Core-mode speedup remains below 1 at smaller payloads due to the fixed costs associated with fragmentation, re-serialization, and the XOR operation of the AONT. Table 4 confirms this aspect, showing the Core-mode cost decomposition of fragmentation, re-serialization, and the XOR operation. The table is based on the best configuration proposed by Kapusta & Memmi with $k = 4$ and $e = 2$: 39.8% of the Core-mode time is spent to AES operations; the remaining 60.2% is spent on other algorithm phases.

The benefit of partial encryption can be read directly by comparing Tables 1 and 5. Table 5 reports the results of AES-baseline, a pure AES-128-CBC loop through the JVM JCE provider (without fragmentation, AONT, storage or network) for each payload size, aggregated across all configurations (k, e). The Cortex-A72 architecture can be implemented with the optional ARM Cryptography Extensions, but these are not exposed by the SoC of Raspberry Pi 4. Therefore, the JVM cannot rely on hardware AES instructions, and AES execution is handled purely via software. The measurements shown in Table 5 can therefore be interpreted as the upper limit of the platform's software AES rather than as a limit of the hardware AES. We ran a cross-check with the command `openssl speed -evp aes-128-cbc`, which confirmed the results.

Focusing on the 10 MB payload, the mean throughput of the AES loop is 40.18 MB/s (Table 5, last row), against 46.25 MB/s for the corresponding PE-AONT Core-mode aggregated over all configurations (k, e) (Table 1, last row of the Core-mode). Partial encryption is on average $\sim 15\%$ faster than the AES-baseline at a 10 MB payload and the best configuration ($k = 8, e = 2$) peaks at 72.72 MB/s (almost twice the AES ceiling).

This advantage holds only for payloads of approximately 50 kB or larger, where the AES work skipped by partial encryption exceeds the fixed overhead of fragmentation, the AONT XOR pass on the entire payload, and reserialization.

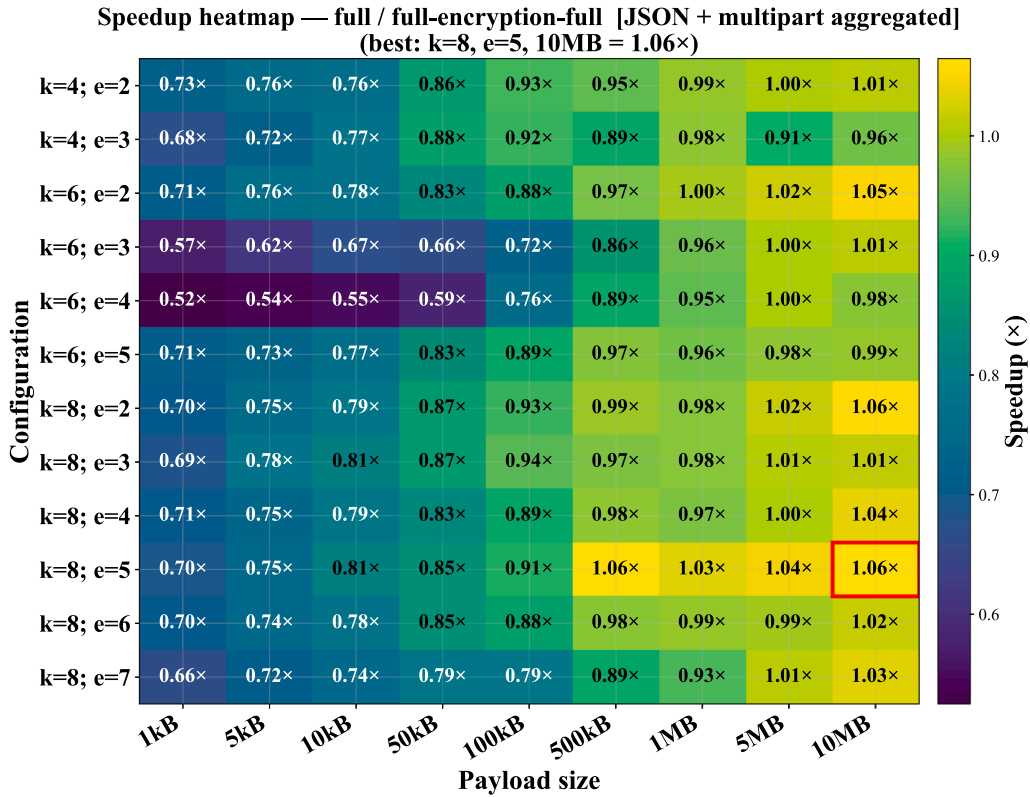


Fig. 6. System speedup over Full-encryption-full baseline in Full-mode versus payload size and configurations (k, e) aggregated and averaged over payload type (JSON and multipart/form-data). The cell with red border is the best overall configuration in terms of speedup.

Table 5

Intrinsic AES-128-CBC throughput ceiling of the raspberry Pi 4. Measured with a pure AES loop through the JVM JCE provider (no fragmentation, AONT, storage or network), with the JVM using `-XX:+UseAES -XX:+UseAESIntrinsics`. The table shows: payload size; mean throughput and standard deviation σ in MB/s, minimum and maximum throughput; 99th and 50th percentile of the throughput distribution; $\Delta\text{Conv}\%$. Low percentage values of $\Delta\text{Conv}\%$ indicate a steady-state system.

Payload	AES Thr. $\pm\sigma$ (MB/s)	Min	Max	p99	p50	$\Delta\text{Conv}\%$
1kB	22.54 $\pm$ 4.80	5.32	26.55	26.07	24.34	0.32
5kB	34.61 $\pm$ 7.20	9.49	38.81	38.57	37.47	0.43
10kB	36.13 $\pm$ 8.88	1.17	41.19	41.02	40.14	2.79
50kB	38.45 $\pm$ 8.96	12.02	43.23	43.17	41.91	1.92
100kB	38.56 $\pm$ 8.84	6.45	43.53	43.41	42.03	2.94
500kB	40.65 $\pm$ 5.97	11.08	43.44	43.35	42.67	1.53
1MB	40.15 $\pm$ 6.42	16.26	43.33	43.19	42.53	0.34
5MB	38.86 $\pm$ 4.03	25.58	43.13	43.05	39.46	0.64
10MB	40.18 $\pm$ 2.42	29.99	43.16	43.06	40.76	0.27

4.2. Runtime JVM effect

Since the benchmarks are performed on a JVM-based implementation, this subsection describes the runtime effects of the JVM that may influence the measurements collected in a managed execution environment.

The JVM cold-start overhead is calculated using the first 5 iterations of each configuration in Full-mode with a 1 kB JSON payload. This is a design choice: our benchmark runs Full-mode as the first mode, with a 1 kB JSON payload.

Table 6 compares those five samples with the other 50 steady-state iterations of the same configuration. The first iterations are between $4.06\times$ ($k=8, e=7$) and $7.12\times$ ($k=6, e=2$) longer than steady-state ones, with a cold latency stable at around 1 s and post-warm-up ranging from 159 ms to 293 ms. These results suggest that the cost is dominated by the JVM itself – class-loading operations, compilation through C1 and C2, AES intrinsic operations – rather than by the choice of parameters (k, e) of the PE-AONT algorithm.

Table 6

JVM cold-start overhead in `full-mode` with a 1 kB JSON payload. Table shows: cold mean is the mean end-to-end latency of the 5 explicit warm-up invocations of each configuration (k, e); Warm mean $n = 5$ is the mean of the first 5 iterations that immediately follow the warm-up; warm mean $n = 50$ is the mean over all 50 steady-state iterations of the same configuration, representing the asymptotic latency once HotSpot has completed its tiered compilation; overhead is the ratio of cold mean to warm mean $n = 50$; a value above 1 measures how much slower the cold path is than the steady-state.

Config. (k, e)	Cold mean (ms)	Warm mean $n = 5$ (ms)	Warm mean $n = 50$ (ms)	Overhead (cold/warm)
$k = 4, e = 2$	1034.84	285.85	182.49	5.67
$k = 4, e = 3$	1007.01	197.87	159.85	6.30
$k = 6, e = 2$	1354.70	219.00	190.21	7.12
$k = 6, e = 3$	1049.61	246.72	199.89	5.25
$k = 6, e = 4$	1066.69	239.69	192.15	5.55
$k = 6, e = 5$	1114.58	229.80	188.19	5.92
$k = 8, e = 2$	1117.19	306.84	243.89	4.58
$k = 8, e = 3$	1186.54	313.38	250.24	4.74
$k = 8, e = 4$	1157.12	320.58	243.92	4.74
$k = 8, e = 5$	1209.13	294.24	235.10	5.14
$k = 8, e = 6$	1097.32	304.10	240.85	4.56
$k = 8, e = 7$	1189.23	342.23	293.05	4.06

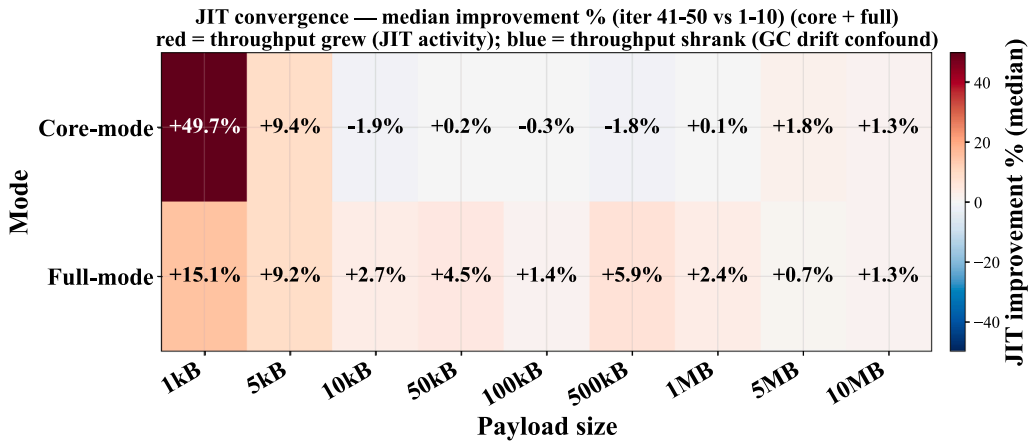


Fig. 7. Within-JVM JIT convergence for Core-mode and Full-mode versus payload size. For every cell the throughput shift between the first ten and the last ten of the 50 steady-state iterations is reported. Positive values indicate that throughput grew during the iteration sequence; values close to zero indicate steady state already from iteration 1. Large payload are affected by GC pressure (Table 7).

To isolate the effect of the JIT, we measured the difference in throughput between the first 10 iterations of each configuration and the last 10. The results are shown in Fig. 7. The median improvement with a payload of 1 kB is 49.7% for Core-mode and 15.1% for Full-mode. This effect rapidly decreases as the runner progresses through larger payload sizes: from 10 kB payload size, the median improvement is already within a $\pm 3\%$ band in both modes, meaning that Java HotSpot has reached steady state. However, this decrease is also due to the cumulative GC pressure. Table 7 reports the cumulative GC pauses per node and payload size. Node01 acts as the entry point during the benchmark and receives every incoming POST, performs the AONT transform on the entire payload, and writes the resulting fragments to MongoDB before distributing them to the storage nodes. Its cumulative pause grows from 435 ms for a 1 kB payload size to approximately 45 s for 10 MB payload.

4.3. System utilization on raspberry Pi 4

Using constrained devices such as the Raspberry Pi 4 can lead to hardware resource saturation, resulting in throughput bottlenecks. In Table 8, we show the per-node impact of our implementation during the benchmark in terms of memory usage, CPU utilization, power consumption, temperature, energy, and network consumption. Node01 recorded the highest resource consumption, as it acted as the entry point to the cluster during the testing phases. On Node01 the JVM peak heap is 783.24 MB, out of 950 MB upper bound that G1 reserves by default. MongoDB peaks at 1.5 GB on the same node, which dominates the memory footprint. The database

Table 7

GC pause activity per node and payload size. The first row is the entry-point Node01; the second is the arithmetic mean over Node02, Node03, and Node04 storage nodes; the third is the register node. Total pause is the cumulative stop-the-world time the JVM spent in G1 collections during the workload at that payload size.

Role	Payload	Total pause (ms)	Mean (ms)	Max (ms)
Node01	1kB	434.61	6.49	17.60
	5kB	377.33	6.29	19.64
	10kB	502.19	7.73	60.64
	50kB	772.05	7.35	20.44
	100kB	1049.29	5.89	21.50
	500kB	4096.65	4.43	161.77
	1MB	5903.90	4.89	399.95
	5MB	24166.23	7.32	1947.44
	10MB	44944.85	9.54	1742.39
storage (avg. of Node02–04)	1kB	66.74	8.70	18.57
	5kB	65.91	6.18	24.88
	10kB	91.99	7.88	26.51
	50kB	66.11	6.84	16.44
	100kB	106.45	8.63	22.50
	500kB	240.06	6.55	16.32
	1MB	419.84	6.96	21.51
	5MB	1515.88	6.17	21.65
	10MB	1158.53	6.98	74.82
Register node	1kB	62.13	10.36	15.59
	5kB	117.29	11.73	20.79
	10kB	166.12	12.78	28.92
	50kB	79.17	13.19	30.17
	100kB	57.56	9.59	17.75
	500kB	52.56	8.76	10.78
	1MB	93.59	10.40	17.03
	5MB	141.70	11.81	24.46
	10MB	164.30	11.74	23.33

Table 8

Per-node system utilization on a 4 GB raspberry Pi 4 during the benchmark.

Metric	Node01	Node02	Node03	Node04	Register
Heap peak (MB)	437.00	224.00	257.00	271.00	90.00
Heap committed (MB)	515.00	229.00	261.00	279.00	122.00
JVM RSS peak (MB)	783.24	411.03	445.95	462.26	307.72
JVM RSS mean (MB)	427.56	291.76	305.99	287.89	247.58
MongoDB RSS peak (MB)	1568.87	1416.01	1343.03	1358.59	99.91
Sys used peak (MB)	2637.59	2216.11	2239.87	2280.82	832.44
% used RAM (4 GB)	64.39%	54.10%	54.68%	55.68%	20.32%
Java CPU peak (%)	269.0	234.0	284.0	319.0	230.0
Java CPU mean (%)	17.8	6.2	5.9	6.1	5.4
Temp. peak (°C)	41.9	39.9	39.9	43.3	39.4
Temp. mean (°C)	36.7	34.7	34.6	35.3	35.0
Energy consumption (Wh)	3	3	3	3	3
Network traffic (GB)	228.00	29.49	23.67	23.67	0.57

is therefore the component that consumes the most system resources, but the four-core Cortex-A72 is far from saturated. The Java process uses between 2 and 3 of the 4 available cores. Total system RAM peaks between 55% and 65% for the computational nodes and 20% for the registry node. During the nearly 50 h of benchmarking, the mean temperature of each node remained stable between 35° and 36.7° C – approximately 37° C below the Pi4/s thermal-throttling threshold of 80° C. Energy consumption also remained stable at around 3 Wh throughout the duration of the benchmarks.

In an IoT environment, latency also plays a key role for the end user. [Table 9](#) shows client-side tail latency for Full-mode aggregated over all (k, e) configuration and payload types. For payloads typical of IoT applications (≤ 100 kB), the 99th percentile of client-observed latency stays below 530 ms. The residual outliers visible in the last column are consistent with the GC pause events reported in [Table 7](#).

Table 9

Client-side tail latency for Full-mode aggregated over all configurations (k, e) and payload type (JSON and multipart/form-data). The table shows: 50th, 99th and maximum percentile of total client time.

Payload	p50 (ms)	p99 (ms)	max (ms)
1kB	166.34	367.23	795.41
5kB	155.69	340.03	565.99
10kB	151.71	337.94	457.90
50kB	202.96	493.20	1436.10
100kB	253.92	528.06	690.36
500kB	663.16	1262.10	3167.88
1MB	1124.57	2021.37	3761.77
5MB	6810.53	19455.29	39310.13
10MB	13147.42	25861.58	42567.60

5. Conclusions

This paper focused on the PE-AONT scheme [1], designed to enhance data security within a Dispersal Storage System (DSS) by balancing security and processing speed. The technique is particularly suitable for deployment scenarios where resources – such as energy, computation, and memory – are constrained. To verify this assumption, a reimplementing study was conducted using a network of Raspberry Pi nodes. In addition to demonstrating the replicability of PE-AONT within an IoT setup, we observed throughput ranging from 26.11 to 37.92 MB/s (Table 2), a ratio of 1 : 8 compared to what Kapusta & Memmi originally claimed, with a peak of 72.72 MB/s for a different configuration ($k = 8, e = 2$). The observed performance gap can be explained by the absence of hardware support for the ARMv8 Crypto Extensions in the Raspberry Pi 4.

We make the resulting implementation publicly available to foster further research and industrial adoption. We note that our implementation has some limitations, mainly related to the management of the encryption key and the initial vector (IV), which must be securely shared among the nodes of the system. Another limitation concerns the registry node, which tracks the locations of the data fragments across the system. Since this study focuses on the performance characterization of PE-AONT, fault tolerance was not evaluated. This includes scenarios such as storage-node failures, distribution of the fragment-location registry across multiple computational nodes, and compromise of the registry node. The trade-off between security and performance was also not quantified. Furthermore, a more fine-grained analysis is required to isolate the effects of the input format (JSON vs multipart/form-data) on system performance. All these enhancements are left to future work.

Once inside the Wi-Fi network, an attacker would still need both the object identifier and the AES key to compromise the system and reconstruct the original data, which already provides a strong security layer. However, we suggest that future works should also explore more sophisticated techniques to further protect the registry node and reduce its exposure. Finally, centralizing the metadata that links the dispersed blocks within a single node introduces a potential Single Point Of Failure, which could be mitigated by implementing redundant backup nodes or adopting distributed registry mechanisms.

CRedit authorship contribution statement

Roberto Vergallo: Writing – review & editing, Writing – original draft, Software, Methodology, Conceptualization; **Simone Casciari:** Writing – review & editing, Writing – original draft, Visualization, Software, Investigation, Formal analysis, Data curation; **Luca Mainetti:** Writing – review & editing, Validation, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Simone Casciari reports financial support was provided by NextGenerationEU. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was co-financed by the European Union - NextGenerationEU.

Data availability

We have shared the repository associated with this manuscript.

References

- [1] K. Kapusta, G. Memmi, PE-AONT: partial encryption all or nothing transform, in: Proceedings of the International Conference on Security and Cryptography (SECRYPT), 2022, pp. 636–641. <https://doi.org/10.5220/0011300600003283>
- [2] R.L. Rivest, All-or-nothing encryption and the package transform, in: Fast Software Encryption: 4th International Workshop, FSE'97, Haifa, Israel, January 20–22, 1997, Proceedings, 1267 of *Lecture Notes in Computer Science*, Springer, 1997, pp. 210–218. <https://doi.org/10.1007/BFb0052359>
- [Rep] Our open-source PE-AONT code and detailed documentation, <https://github.com/softengunisalento/PE-AONT-SYSTEM-RP>.
- [4] M. Baldi, N. Maturo, E. Montali, F. Chiaraluce, AONT-LT: a data protection scheme for cloud and cooperative storage systems, in: Proc. Int. Conf. on High Performance Computing & Simulation (HPCS), 2014, pp. 566–571. <https://doi.org/10.1109/HPCSim.2014.6903736>
- [5] J.K. Resch, J.S. Plank, AONT-RS: blending security and performance in dispersed storage systems, in: Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11), USENIX Association, 2011.
- [6] M. Luby, LT codes, in: The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings., IEEE Computer Society, 2002, pp. 271.
- [7] A. Rickus, E. Pfluegel, N. Atkins, Chaos-based image encryption using an AONT mode of operation, in: Proc. Int. Conf. on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA), 2015, pp. 1–5. <https://doi.org/10.1109/CyberSA.2015.7166113>
- [8] J. Ye, X. Deng, A. Zhang, H. Yu, A novel image encryption algorithm based on improved Arnold transform and chaotic pulse-coupled neural network, *Entropy* 24 (8) (2022) 1103. <https://doi.org/10.3390/e24081103>
- [9] P. Xie, H. Li, H. Yu, Z. Chen, AONT-NZZD: a secure and efficient dispersal scheme in distributed storage systems, in: Proc. Int. Conf. on Big Data (Big Data), 2019, pp. 6232–6239. <https://doi.org/10.1109/BigData47090.2019.9005571>
- [10] R.A. Chou, J. Kliever, Secure distributed storage: optimal trade-off between storage rate and privacy leakage, *IEEE Trans. Inf. Theory* 70 (5) (2024) 3658–3668.
- [11] R. Nabawy, M. Hassanin, M.H. Ibrahim, M. Kaseb, Mixed fragmentation technique for securing structured data using multi-cloud environment (MFT-SSD), *Ad Hoc Netw.* 164 (2024) 103625.
- [12] A. Sardar, S. Umer, C. Pero, M. Nappi, A novel cancelable facehashing technique based on non-invertible transformation with encryption and decryption template, *IEEE Access* 8 (2020) 105263105277.
- [13] E. Baagyere, E. Batowise, P. Agbedemrab, G.E. Wenya, R. Turkson, An RNS-based all-or-nothing transform for CP-ABE in IoT distributed systems, *J. Cybersecur.* 11 (1) (2025) tyaf015.
- [14] T. Shi, Q. Zhao, N. Liu, An efficient attribute-based proxy re-encryption scheme supporting AONT-based encrypted deduplication, in: Proceedings of the 2024 7th International Conference on Artificial Intelligence and Pattern Recognition, 2024, pp. 847–852.
- [15] J. Kaur, A. Cintas Canto, M. Mozaffari Kermani, R. Azarderakhsh, A survey on the implementations, attacks, and countermeasures of the nist lightweight cryptography standard: ascon, *ACM Comput. Surv.* 58 (1) (2025) 1–16.
- [16] V.K. Vatsavayi, D. Bommireddy, Secure and efficient block cipher mode design for parallel processing and reliable security, *Cryptography* 10 (1) (2026) 13.
- [17] S.M. Zobaed, M.A. Salehi, Confidential computing across edge-to-cloud for machine learning: a survey study, *Softw. Pract. Exp.* 55 (5) (2025) 896–924.
- [18] M. López-Benítez, M. Alhulayil, On the computational efficiency of fading models: the fluctuating two-ray model for mmwave bands, *IEEE Open J. Commun. Soc.* 6 (2025) 8175–8189. <https://doi.org/10.1109/OJCOMS.2025.3608296>