HEliOS: Huffman Coding Based Lightweight Encryption Scheme for Data Transmission

Mazharul Islam[1], Novia Nurain[2], Mohammad Kaykobad[3], Sriram Chellappan[4], A. B. M. Alim Al Islam[5]
Bangladesh University of Engineering and Technology, Dhaka, Bangladesh[1, 2, 3, 5] University of South Florida, FL, USA[4]
{mazharul, novia}@cse.uiu.ac.bd[1, 2], {kaykobad, alim_razi}@cse.buet.ac.bd[3, 5], sriramc@usf.edu[4]

Abstract

Demand for fast data sharing among smart devices is rapidly increasing. This trend creates challenges towards ensuring essential security for online shared data while maintaining the resource usage at a reasonable level. Existing research studies attempt to leverage compression based encryption for enabling such secure and fast data transmission replacing the traditional resource-heavy encryption schemes. Current compression-based encryption methods mainly focus on error insensitive digital data formats and prone to be vulnerable to different attacks. Therefore, in this paper, we propose and implement a new **H**uffman compression based **E**ncryption scheme using **li**ghtweight dynamic **O**rder **S**tatistic tree (**HEliOS**) for digital data transmission. The core idea of HEliOS involves around finding a secure encoding method based on a novel notion of Huffman coding, which compresses the given digital data using a small sized "secret" (called as *secret_intelligence* in our study). HEliOS does this in such a way that, without the possession of the secret intelligence, an attacker will not be able to decode the encoded compressed data. Hence, by encrypting only the small-sized intelligence, we can secure the whole compressed data. Moreover, our rigorous real experimental evaluation for downloading and uploading digital data to and from a personal cloud storage Dropbox server validates efficacy and lightweight nature of HEliOS.

CCS Concepts

• **Security and privacy → Software and application security** ; *Web application security*.

Keywords

compression, encryption, Huffman encoding, order statistic trees.

ACM Reference Format:
Mazharul Islam[1], Novia Nurain[2], Mohammad Kaykobad[3], Sriram Chellappan[4], A. B. M. Alim Al Islam[5]. 2019. **HEliOS**: Huffman Coding Based Lightweight Encryption Scheme for Data Transmission. In *Proceedings of 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous).* ACM, Houston, TX, USA, 10 pages. https://doi.org/10.1145/3360774.3360829

1 Introduction

Recent proliferation in data connectivity along with low pricing of hardware devices have enabled smart devices with many applications. These applications tend to stay coherently synced to a cloud server, and enable sharing a diverse range of multimedia and textual data covering audio, video, emails, sensor data (from environment and infrastructure such as rail lines), health data (such as heart rate, body movements, sleep-wake time and body temperature) etc. Most digital data today are highly sensitive as well as confidential. These applications demand fast and energy efficient security mechanisms to protect their data.

In order to implement security mechanisms for shared data, we generally need to apply encryption to the transmitted data. However, encryption is often a resource-expensive task, which consumes a lot of CPU cycles and valuable resources. In this realm, existing studies [1, 5, 25] have come up with lightweight cryptographic schemes to ensure secure communication without demanding substantial time and resource usage to secure the data before transmission. However, the level of security gets sacrificed with these lightweight schemes in most of the cases. Moreover, with the growing demand of fast connectivity for these devices, surprisingly manufacturers have taken a retroactive step by trading off security to meet the demands of fast connectivity. Do et al., [11] have recently presented a case study on Samsung Gear Live smart-watch showing that recent smart devices store and transmit a wide range of unsecured sensitive data, which are vulnerable to attacks and exfiltration.

In this regard, designing compression based encryption schemes can appear as a promising approach solution [14, 16, 20, 24, 26, 29, 30, 37, 38]. Here, by combining compression and encryption techniques, we can achieve security, computational and energy savings in two steps as shown on Fig. 1. Firstly, we can apply encoding to compress to the data using a small-sized *secret_intelligence*. We introduce the *secret_intelligence* in such a way that it becomes impossible (or extremely difficult) for an attacker to decode the compressed encoded data without knowing the *secret_intelligence*. Secondly, we secure the encoded compressed data simply by encrypting the small-sized *secret_intelligence*, which is used in the encoding mechanism. From a cryptographer's point of view, we can relate the secret_intelligence to the notion of key used in traditional encryption standards. Just like without the key we can not decrypt the encrypted data in traditional encryption standards, without the *secret_intelligence* we cannot decompress (nor restore) the compressed data in our case. Therefore, by securing only the *secret_intelligence* that is required for decompressing the data, we can make the whole compressed data as meaningless as encrypted data would appear to attackers. Existing encoding schemes that can be used to compress the data in this regard are either error insensitive (e.g., Compressive Sensing [4]), vulnerable to attacks (e.g., Multiple Huffman Tree [37] and Swapped Huffman Tree [20]), or suffer from poor scalability (e.g., Chaotic Huffman Tree [16]).

To this extent, in this paper, we propose and implement a new **H**uffman compression based **E**ncryption scheme using **li**ghtweight dynamic **O**rder **S**tatistics trees (**HEliOS**) for data transmission. HEliOS builds a dynamic order statistic tree using a secret_intelligence and uses this dynamic order statistic tree to encode and compress plain text data. For secured and fast data transmission, HEliOS encrypts only the secret_intelligence using receiver's public key and sends the encoded compressed data along with it. The receiver, then, can first decrypt the secret_intelligence with his/her private key to build the same dynamic order statistic tree using the secret_intelligence. Subsequently, the receiver can decode the encoded data using the built dynamic order statistic tree. Form an attacker's point of view, without the possession of the receiver's private key, the attacker can not figure out the secret intelligence and build the order statistic tree. Without the order statistic tree, the encoded data remains non-decodable and safe from the attacker.

Based on our study, we make the following key contributions in this paper:

- We present a novel compression based encryption scheme HEliOS, which can be used as a new encryption standard for data sharing. HEliOS can meet the demand of fast online file sharing in a secured way.
- We present theoretical analysis on security and performance of HEliOS through several lemmas and their proofs.
- We implement HEliOS in two different real experimental scenarios. One of them covers mobile and laptop clients connected to a personal cloud storage Dropbox server. The other one covers sharing files in a wired P2P network comprising several machines. Our experimental results demonstrate that HEliOS achieves significantly faster transmission and reception of data in a secured manner for a range of digital benchmark data formats (i.e., audio, video, emails, ebooks, and office files) compared to other traditional methods. Additionally, HEliOS when implemented on mobile devices demonstrates energy efficiency.

2 Background on Compression Based Encryption Techniques

In this section, we will discuss two types of existing compression based encryption schemes, namely Compressive Sensing and Huffman encoding which combine compression and encryption.

2.1 Compressive Sensing Based Encryptions

Compressive Sensing (CS) [4] has attracted a lot of attention among researchers in the search for energy efficient secure encryption schemes. CS uses a sensing matrix to compress the data and without the presence of the sensing matrix the attacker can not decompress the data with low error. Hence, this sensing matrix can be easily used as the secret intelligence. Given vector $x \in R^n$, we can compute its representation $\theta \in R^n$ in a basis $\Psi \in R^{n \times n}$ by solving the Eqn $x = \Psi\theta$. We can say x is $m - compressible$ if in θ there are m elements with significant coefficients and $n - m$ elements with almost zero magnitude. CS is applicable to $m - compressible$ data only when the value of m is small enough. We encrypt x by using

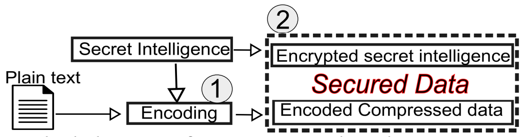


Figure 1: Block diagram of a compression based encryption scheme.

the Eqn. $y = \Phi x + z$ which is to multiply x with the projection matrix $\Phi \in R^{m \times n}$ under the condition that x is $m - compressible$ in some basis Ψ. While decompressing, we can get an estimated $\hat{x}$ from $\hat{x} = \Psi\hat{\theta}$ where we find the optimized $\hat{\theta}$ subjected to minimizing $||y - \Phi\Psi\hat{\theta}||$. The reconstruction error $||\hat{x} - x||$ depends on how large m is in Φ. and which basis Ψ is chosen. These two terms are expressed in terms of sensing matrix $A = \Psi\Phi$. One key point to note here is that CS is lossy compression sacrificing a small amount of reconstruction error while decompressing the compressed data on the receiver's side even with the correct sensing matrix A. As a result, CS as a means for encryption standard can only work for data that is not accuracy sensitive, but will fail when smart devices need to communicate on accuracy sensitive data (e.g., text, physiological signals etc.) where we cannot accept any error. Hence CS based encryption schemes proposed in [14, 24, 26, 29, 30, 38] are not suitable for error sensitive data.

2.2 Huffman Based Encryption Schemes

The classical Huffman compression coding [17] assigns prefix binary strings of 0's and 1's to represent each symbol in the compressed data. The assignment of binary strings is optimal in the sense the symbols with the highest frequencies, are assigned the binary string having lowest lengths and hence the plain text is compressed. Fig. 2 illustrates one such classical Huffman tree for the plain text *"aabcacbddbaaa"*. The symbol 'a' which has the frequency 6 is assigned the lowest length binary encoding string "0". In the same way, the next higher frequency symbols 'b', 'c', 'd' (with frequencies 3, 2, 2) are assigned encoding strings "10", "110", "1111" respectively. More specifically, Huffman compression keeps the

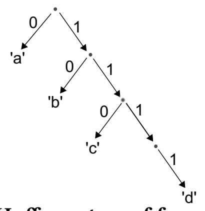


Figure 2: A classical Huffman tree of four symbols 'a', 'b', 'c', 'd'.

coding rules in a small-sized Huffman tree. Without the small sized Huffman tree, decompressing the compressed data completely is near to impossible (NP-Hard) [15]. Hence we can treat this classical Huffman tree as the secret_intelligence. This motivated the study in [21] to use Huffman compression to store a large textual compressed database on a CD-ROM securely. Moreover, existing research work which have tried to use this concept of the compressed data being breakable in the absence of Huffman tree are Multiple Huffman Tree (MHT) [37], Chaotic Huffman Tree (CHT) [16], Swapped Huffman Tree (SHT) [20] for the purpose of storing large multimedia data securely and in a resource efficient way. However, these schemes have classical vulnerabilities that are elaborated upon in the next section in detail.

2.3 Lightweight Attribute-based Encryption Schemes

There are other works that have tried to address the above problem using lightweight attribute based encryption schemes [1, 25, 33, 34]. However, for the purpose of end to end data transmission, attribute based encryption techniques inherently suffer from less flexibility, scalability, and generality [39].

3 Design Issues

In this section, we will discuss the challenges in designing a combined compression and encryption methods to use it as a comprehensive encryption standard. Following that, we will highlight why existing solutions are not suitable for secure, fast transmission of data files in a resource efficient way.

3.1 Challenges

An ideal compression based encryption scheme should have the following properties in order to use it as a comprehensive encryption standard.

- *Randomness property:* Making the compressed data as random as an encrypted data would appear to an attacker.
- *Sensitivity:* Making the compressed data sensitive to small changes in key and plain text.
- *Key space.* The key space should be large enough to resist brute-force attack.

3.2 Threat Models

We enumerate four classical types of attacks from the hardest to easiest as follows:

- *Cipher text only:* the attacker only can eavesdrop a large or small volume of cipher texts.
- *Known plain text:* the attacker process the plain texts and their corresponding cipher texts.
- *Chosen plain text:* In this case, we consider, the attacker has temporary access to the encryption machinery. Hence s/he can choose a plain text and construct the corresponding cipher text.
- *Chosen cipher text:* The attacker has temporary access to the decryption machinery. Hence s/he can choose a cipher text and construct the corresponding plain text.

In each of the attacks, the goal of the attacker is to figure out any secret information (such as the secret intelligence or partial recovery of the encrypted data) [18]. The resistance of any compression based encryption scheme against these classical four types of attacks largely depends on how well these two properties (i.e., randomness and sensitivity) are maintained. Any compression based encryption scheme which does not preserve these two properties while encoding the data can be vulnerable against these four classical attacks. Therefore, the challenge for us is to design a scheme that introduces randomness, and to insert sensitivity in the encoded data during compression, while ensuring resource efficiency simultaneously.

3.3 Why not Existing Compression Based Encryption Schemes?

Although without the Huffman tree, an attacker can not decode the Huffman encoded data fully, with some luck on the attacker's side, *partial partial* recovery of the plain text is certainly possible exploiting prior knowledge of average frequency of the symbols by analyzing the existing natural texts [19]. For example, the probability of more frequent symbols (i.e., a, e, i, o, etc.) being assigned small length binary encoding strings is higher compared to others less frequent occurring symbols (i.e., z, j, q, x, etc.). Moreover, in this case, Huffman encoded data is not *sensitive* to small changes in the plain text. In addition to that, each symbol is represented by the same binary encoding string. As a result, the Huffman encoded data may not appear as *random* as it should be to an attacker. This lack of *sensitivity* and *randomness* makes the Huffman encoded data vulnerable even without the presence of classical Huffman tree.

Firstly, Multiple Huffman Tree (MHT) was proposed in [37] which keeps four public Huffman trees as well as a secret ordering (i.e., secret intelligence) to decide on which Huffman tree among the four should be used while encoding. Any adversary can not decode the encoded compressed data since the secret ordering is unknown. However, MHT fails to achieve the expected level of *randomness* required for securing the data. By exploiting this lack of randomness of MHT, Zhou et al., [40] proposed a known plain text attack of complexity $O(3^{|number\_of\_symbols\_in\_data|})$ and later proposed another simple *known plain text* attack where only 10 blocks of plain text is required to know the secret order and break MHT easily. The study in [18] proposed a more sophisticated *known plain text* attack of complexity 2^{32} and offered Chaotic Huffman Tree (CHT) as a solution. CHT tries to achieve the expected level of randomness by mutating the Huffman tree with the help of chaos theory. However, CHT mutates the Huffman tree using a newly generated chaotic sequence for each symbol of the data. As a result of this, mutation operation for a symbol in the data makes CHT substantially computational expensive. Swapped Huffman tree (SHT) was proposed to achieve the same level of randomness as CHT while still generating lower computational overhead. However, both SHT and CHT use 'mutation' operation to introduce randomness in the encoding process, which can reveal the secret key by a novel *known plain text* attack presented as follows.

The 'mutation' operation involves swapping the 0 and 1 labels of the two edges of an internal node if the corresponding bit in the secret key is 1. More specifically, if the secret key is $(k_1, k_2, k_3, \cdots, kn)$, then for each internal node s_i its edge labels will be swapped if k_i is 1. The mutated tree will now encode the symbols $(s1, s2, s3, \cdots, s_n)$ differently from $c_{initial} = (c_1, c_2, c_3, \cdots, c_n)$ to $c_{mutated} = (\overline{c_1}, \overline{c_2}, \overline{c_3}, \cdots, \overline{c_n})$. Moreover, since the mutation operation only swaps the two edges of a node, for each symbol s_i, $length(c_i) = length(\overline{c_i})$ is maintained always. This causes the compression ratio to remain optimal even after mutation applying operation on the classical Huffman tree. Fig. 3 shows a 'mutation' operation. Our key observation is that for any symbol (say s_i) if the attacker knows the binary encoding string before and after the mutation, that is both c_i and $\overline{c_i}$, from the value of $c_0 \oplus \overline{c_0}$, then the attacker can find out which internal nodes edge labels have been swapped. Consequently, this will enable the attacker to figure out the bit values of the secret key corresponding to the internal nodes, which are in the path from the root to the leaf of the symbol. For example, as shown on Fig. 3 the path from root to the leaf s_1 are the internal nodes ① and ②. Since $c_0 \oplus \overline{c_0} = 01$ (Table. 1), the attacker can conclude that the edge labels of node ① are not swapped

Symbol	c_i	$\overline{c_i}$	$c_i \oplus \overline{c_i}$
s_1	00	01	01
s_2	01	00	01
s_3	10	11	01
s_4	110	100	010
s_5	111	101	010

Table 1: Encoding strings before and after the mutation operation.

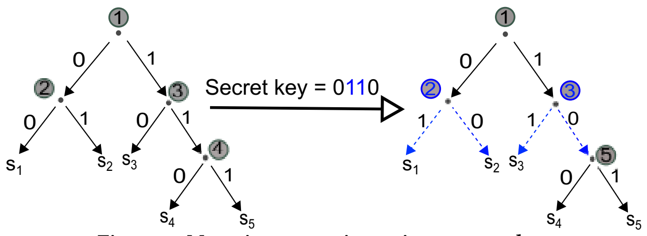


Figure 3: Mutation operation using a secret key

.

whereas, the other node ②'s edge label are swapped. Hence the attacker can conclude that the secret key should be "01xx". To find the 3rd and 4th bit, the attacker can do the same for other symbols. Hence the attacker can guess the all possible binary encoding strings for a symbol c_i (there are 2^l of them for a particular length l) and place them in the plain text. By observing the corresponding binary encoding string in the cipher text $\overline{c_i}$, from the values of $c_i \oplus \overline{c_i}$ the attacker finds some bit positions of the secret key. To figure out the full data, the complexity of the above mentioned *chosen plain text* attack would be $O(n \cdot 2^l)$ where n is the number of symbols in the data and l is the average length of binary encoding strings. Fortunately for the attacker the value of l is bounded by the entropy (H) in the data which is $H \le l \le H + 1$ [9].

4 Our Compression Based Encryption Method

To overcome the above challenges and limitations of existing encoding schemes, we propose and implement a new compression based encryption scheme. We named our method **H**uffman Compression based **E**ncryption using **li**ghtweight dynamic **O**rder **S**tatistic Tree (HEliOS). In this section, we will first discuss the design of HEliOS such as the building blocks of HEliOS, how HEliOS secures the data by encoding and the receiver retrieves the secured data by decoding. Then we will wrap up this section with some key lemmas regarding security and performance analysis of HEliOS.

4.1 HEliOS Design

4.1.1 Building Blocks of HEliOS: HEliOS modifies the traditional Huffman encoding by introducing variable coding and inserting sensitivity while encoding the data. To achieve sensitivity, HEliOS uses the pseudorandom behavior of chaotic maps and introduces variable encoding by using a dynamic order statistic tree.

Dynamic Order Statistic Tree: Dynamic Order statistic tree is a kind of binary search tree which in addition to providing traditional insertion, lookup and deletion, also supports the following two additional queries [7] in the binary search tree.

- *Select*(i) finds the i^{th} smallest element.
- *Rank*(x) finds the rank of x^{th} node.

When a balanced binary tree is used to implement dynamic order statistic tree, both of these queries can be answered in $O(log(n))$.

HEliOS associates two properties with each symbol which decides the rank of symbol based on Algo. 1

- *Current frequency:* which keeps a count of the number of times a symbol appears in the plain text as we read the plain text while compressing. The frequencies are initialized to zero before the start of the reading.
- *Chaotic weight:* A weight which is generated from a chaotic sequence.

HEliOS uses dynamic order statistic tree to assign variable encoding for each symbol. HEliOS achieves this by updating the rank of symbols in the dynamic order statistic tree. The rank of each symbol decides which binary encoding string should be used to encode the symbol. Initially, each symbol has zero current frequency.

As we read the symbols and increase the current frequencies one by one, the rank of the symbols also gets changed since the current frequencies have precedence over chaotic weights while considering the rank (Algo. 1). These rank updates, in return, change which binary encoding string is used to encode the symbol.

Algorithm 1: Compare The function decides the rank of the symbols in dynamic order statistic tree

```
Input: Two symbols to compare s1 and s2
Output: Returns the symbol which has higher rank
1  if s1.currentFrequency ≠ s2.currentFrequency then
2      if s1.currentFrequency ≥ s2.currentFrequency then
3          return s1
4      else
5          return s2
6  else
7      if s1.chaoticWeight ≥ s2.chaoticWeight then
8          return s1
9      else
10         return s2
```

Chaotic Sequences: The chaotic weights of the symbols are drawn from a chaotic sequence. These chaotic weights are important too because they decide the rank of the symbol when the current frequencies are equal, especially, when at the start of compression when all symbols have zero current frequencies. The chaotic sequences are characterized by their sensitive dependence on initial seed and random like behaviour [22]. There are many chaotic systems which produce finite, pseudorandom sequences such as Bernoulli shift chaotic sequences, piece-wise linear chaotic sequences , spatial chaotic sequences, etc. HEliOS uses Chebyshev chaotic system [23] to generate psudorandom weights for the symbols. The Chebyshev system takes w and c_0 as the initial random seed and generates a chaotic sequence recursively based on the Eqn 1 and 2 below,

$$c_{k+1} = \begin{cases} cos(w \cdot arccosc_k) & \text{if } k > 0 \\ c_0 & \text{if } k = 0, \end{cases} \tag{1}$$

$$\text{where, } c_0 \in [-1, 1] \text{ and } w \ge 2.00. \tag{2}$$

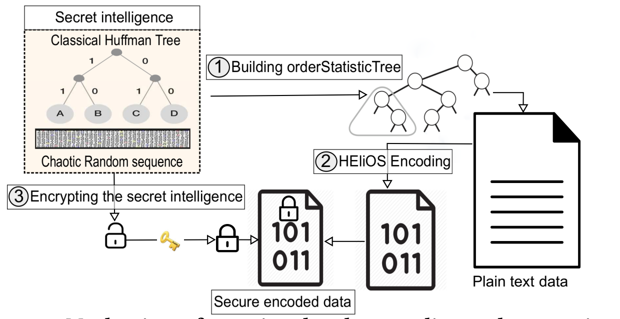


Figure 4: Mechanism of securing data by encoding and encrypting the secret intelligence as followed in HEliOS.

Even when we perturb the initial random seed (c_0, w) by a small fraction, the generated Chebyshev chaotic sequences change completely. As a result, the initial rank of symbols gets significantly changed. This sensitivity of the initial rank with respect to changes in the initial seed is a blessing for cryptographers. This is because the encoding process is extremely sensitive to the initial ranks of the symbols (as explained in Lemma 4.2). Moreover, the Chebyshev chaotic sequence is simple and has low resource requirements.

4.1.2 Securing data: As shown in Algo. 4 and on Fig. 4, we divide the securing the data process into three tasks.

(1) **Building Order Statistic Tree:** We initialize an empty order statistic tree. Each symbol from the classical Huffman tree is inserted in the empty order statistic. The Chebyshev chaotic sequences (Algo 2) are used to assign weights to each symbol.
(2) **HEliOS Encoding:** Using the built order statistic tree, we encode the symbols from the data one by one. For each symbol, we find the symbol's rank in the dynamic order statistic tree (say k) and use the k^{th} smallest length binary encoding string to encode it. Then we increase the current frequency of the symbol by one. Because of changing the frequencies of the symbols, the symbol's rank gets dynamically updated. Therefore, the updated symbols will use a different binary encoding string when they appear next in the plain text (Algo. 3).
(3) **Encrypting the secret intelligence:** We define *secret_ intelligence* as those components which are necessary to build the order statistic tree. More specifically the classical Huffman tree and chaotic random seed together make up the *secret_intelligence*. Therefore, we encrypt the small-sized classical Huffman tree and chaotic random seed using the receiver's public key to safeguard them from attackers.

Without the secret intelligence (i.e., classical Huffman tree and chaotic random seed) the attacker can not build the order statistic tree and decode the encoded compressed data even partially. This makes the encoded, compressed data secure. Hence we combine the secured secret intelligence and encoded data and send this secure data to the receiver.

4.1.3 Retrieving data: The receiver first extracts the encrypted secrete intelligence (i.e., classical Huffman tree and initial randome seed) from received data and then decrypts the secret intelligence using his/her private key. Then the receiver builds an order statistic

Algorithm 2: Building order statistic tree This function builds a order statistic tree from the initial seed and classical Huffman tree.

```
Input: initial seed $(c_0, w)$, classical Huffman tree
Output: An order statistic tree
1 Generate a Chebyshev chaotic sequence $c_1, c_2, c_3, \cdots, c_N$
    from the initial seed $(c_0, w)$
2 Create an empty $orderStatisticTree$
3 foreach $symbol \in classical\ Huffman\ tree$ do
4     $symbol.currentFrequency = 0$
5     $symbol.chaoticWeight = c_i$
6     $orderStatisticTree.insert(symbol)$
7 return $orderStatisticTree$
```

Algorithm 3: Encoding This function encodes the data using dynamic order statistic tree.

```
Input: $data$, $orderStatisticTree$
Output: $encodedData$
1 $encodedData = \phi$
2 foreach $symbol \in data$ do
3     $k = orderStatisticTree.Rank(symbol)$
4     $encodedData.append(k^{th}$ smallest encoding string)
5     $symbol.currentFrequency += 1$
6 return $encodedData$
```

Algorithm 4: Securing data. This algorithm secures the data before sending

```
Input: $data$
Output: $SecuredData$
1 Generate a $random\_seed$ $[c_0, w]$
2 Constract a $classical\_Huffman\_tree$ from $data$
3 $orderStatisticTree$ = BuildOrderStatisticTree
    ($random\_seed, classical\_Huffman\_tree$)
4 $encodedData$ = Encoding($data, orderStatisticTree$)
5 $Secret\_Intelligence =$
    $random\_seed + classical\_Huffman\_tree$
6 $Encrypted\_Secret\_Intelligence$ = Encryption
    ($Secret\_Intelligence, PU_{receiver}$)
7 $securedData = Encrypted\_Secret\_Intelligence + encodedData$
8 return $securedData$
```

tree, the same way the sender does in Algo. 2. With this tree, the receiver decodes the compressed text using Algo. 5.

4.2 Security and Performance Analysis

HEliOS secures the variable and sensitively encoded data by encrypting the small-sized *secret_intelligence*. Attackers cannot decode the encoded data without the *secret_intelligence*. Related work has shown that without the presence of Huffman tree, guessing the plain text of encoded data is NP-hard [13], and our HELiOS technique only makes the decoding harder due to introduction of more randomness in the ciphertext. As such, our technique is secure.

Algorithm 5: Retrieving data This algorithm retrieves the secured data received from the sender

```
Input: $securedData$
Output: $retrievedData$
$[classical\_huffman\_tree, random\_seed(c_0, w)]$ =
    Decryption ($Encrypted\_secret\_intelligence, PR_{receiver}$)
$orderStatisticTree$ = BuildOrderStatisticTree
    ($random\_seed, classical\_Huffman\_tree$)
$retrievedData$ = Decoding ($orderStatisticTree, encodedData$)
return $retrievedData$
```

Algorithm 6: Decoding. This function decompresses the compressed file using the order statistic tree.

```
Input: $orderStatisticTree, encodedData$
Output: $retrievedData$
$retrievedData = \phi$
$binaryCodingString = \phi$
$currentNode = classical\_Huffman\_tree->root$
foreach $bit \in encodedData$ do
    if $currentNode$ is leaf then
        $k = binaryCodingString$ is $k^{th}$ smallest
        $Symbol = orderStatisticTree.Select(k)$
        $retrievedData = retrievedData + S$
        $Symbol.currentFrequency+ = 1$
        $binaryCodingString = \phi$
        $currentNode = classical\_Huffman\_tree->root$
    else
        $binaryCodingString+ = bit$
        if $bit = 0$ then
            $currentNode = currentNode->left$
        else
            $currentNode = currentNode->right$
return $retrievedData$
```

Now, we know that encryption is a time and resource hungry task and in our proposed mechanism, HEliOS only applies encryption to small-sized secret intelligence instead of encrypting the full data. The savings are immense in communication and computation time and energy, while still providing a very high degree of security. Lemmas 4.1 and 4.2 below present the proofs regarding variable encoding and sensitivity respectively. We present two more Lemmas (4.3 and 4.4) and their proofs about the compression ratio of HEliOS encoded data and scalability of encoding time respectively.

Lemma 4.1. *HEliOS uses variable binary encoding strings to encode the same symbol.*

Proof. HEliOS uses the rank of a symbol in the order statistic tree to encode a symbol. If the rank of a symbol is k, then HEliOS uses k^{th} smallest length binary encoding string to encode the symbol (as shown in Algo. 3). To see this, suppose the plain text is *"abbaaa"* and we assign weights c_a, c_b to symbols a and b respectively where $c_a < c_b$. At the same time, assume the encoding

Symbol	Rank	Encoded String	Updated order statistic tree	
-	-	-	$(b, 0, c_b)$	$(a, 0, c_a)$
a	2	011	$(a, 1, c_a)$	$(b, 0, c_b)$
b	2	011	$(b, 1, c_b)$	$(a, 1, c_a)$
b	1	01	$(b, 2, c_b)$	$(a, 1, c_a)$
a	2	011	$(b, 2, c_a)$	$(a, 2, c_a)$
a	2	011	$(a, 3, c_a)$	$(b, 2, c_b)$
a	1	01	$(a, 4, c_a)$	$(b, 2, c_b)$
Final encoded string= (011)(011)(01)(011)(011)(01)				

Table 2: Simulation of HEliOS variable encoding when $c_a < c_b$

Symbol	Rank	Encoded String	Updated order statistic tree	
-	-	-	$(a, 0, c_a)$	$(b, 0, c_b)$
a	1	01	$(a, 1, c_a)$	$(b, 0, c_b)$
b	2	011	$(a, 1, c_a)$	$(b, 0, c_b)$
b	2	011	$(b, 2, c_b)$	$(a, 1, c_a)$
a	2	011	$(a, 2, c_a)$	$(b, 2, c_b)$
a	1	01	$(a, 3, c_a)$	$(b, 2, c_b)$
a	1	01	$(a, 4, c_a)$	$(b, 2, c_b)$
Final encoded string= (011)(011)(01)(011)(011)(01)				

Table 3: Simulation of HEliOS variable encoding when $c_a > c_b$

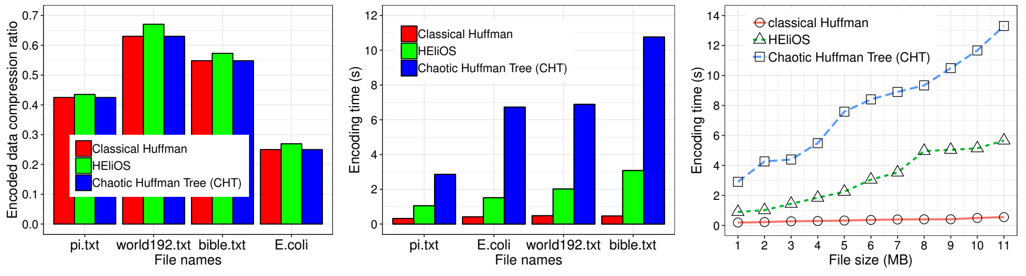
binary strings (from the classical Huffman tree) are (“10”, “011”) which means we will use encoding string “10” when the rank of symbol is 1 and use “011” when the rank is two. As we can see from Table. 2 the final encoded string is (011)(011)(01)(011)(011)(01). Here, a is represented by “011” three times and by “01” one time. In the same way b is represented by “011” and by “01” one time. The optimal encoding using classical Huffman tree would have been (01)(011)(011)(01)(01)(01) where a and b are encoded by 01 and 011 respectively each time. □

Lemma 4.2. *HEliOS compressed text is sensitive to small changes in initial random seed used in secret intelligence.*

Proof. The initial weights set by the chaotic sequences decides the initial rank of the symbols in order statistic tree. A small change in the initial random seed (c_0, w) induces a large change in the chaotic sequences. This large change in the chaotic sequences reflects a large change in the ranks of the symbols when the order statistic tree is built initially. Moreover, the initial ranks of the symbols, bring a significant amount of change in the way plain text is compressed. To see this, we once again encode the same string *"abbaaa"* by keeping every other parameter unchanged as shown in the Table. 2, except for the assumption that $c_a < c_b$. This time we assume $c_a > c_b$, to see how changing the value of the chaotic sequences affects the encoding string. As shown in Table. 3, this time the encoded final string is (01)(011)(011)(011)(01)(01) which is has a hamming distance of around 40% percent from the previously encoded final string in Table. 2. □

Lemma 4.3. *HEliOS encoding maintains near to optimal compression ratio.*

Proof. According to Algo. 1, the rank of each symbol is dependant on two properties a) current frequency of the symbol b) chaotic weights assigned form the Chebyshev chaotic random sequences. Since we use the k^{th} smallest encoding string to encode a symbol which has k rank, in an ideal case to achieve high compression ratio, a symbol which has high frequency in the data file, should have higher rank. In HEliOS, before the start of encoding when we insert the symbols in the dynamic order statistic tree (in Algo. 2), at that time the symbol's rank is only dependent on the chaotic weights of the symbols since each have an initial current frequency



(a) **(b)** **(c)**

Figure 5: Comparison of (a) encoded data compression ratio and (b) encoding time of the standard files from large corpus [6]. (c) exhibits that CHT is not scalable as the files sizes increase.

of zero. Therefore the initial encoding may not be optimal. More specifically, a symbol which has higher frequency may have a lower rank because it has been assigned a low chaotic weight. For this low rank, the high frequency symbol will use encoding strings of larger length compared to the other less frequent symbols deviating from the ideal case of using encoded strings of small size. However, as the encoding process starts, the value of current frequency of the symbols is likely to increase more often than the symbols which have less frequency. As a result, symbols which have higher frequency get higher ranks even if they have been assigned lower ranks before because of low chaotic weight. This is because the current frequency property has preference over the chaotic weight property while comparing the rank of the two symbols (as shown in Algo. 1).

Therefore, we infer that eventually HEliOS will achieve near to optimal compression ratio even though initially it sacrifices optimal symbols to binary encoding assignment. Encouragingly, our experimental results from the Canterbury Corpus [6] (which provides standard large files to compare compression ratios) shown in Fig. 5(a) is in agreement with our inference. It shows that HEliOS encoded data achieves almost the same compression ratio compared to two other optimal encoding techniques namely classical Huffman tree and CHT. □

Lemma 4.4. *HEliOS encoding is more scalable than CHT.*

Proof. CHT mutates the classical Huffman tree for each time a symbol in data. The number of changes in the classical Huffman tree, is proportional to the number of symbols n in it. Hence the complexity of mutating the classical Huffman tree is $O(n)$. Therefore in order to achieve variable encoding, the total running timing time of CHT is bounded by $O(n) \cdot size(data)$. On the other hand, to achieve variable encoding, HEliOS updates the symbol's rank in the order statistic tree by increasing the symbol's current frequency (as shown in Algo. 3). Updating a symbol in a dynamic order statistic tree is $O(logn)$ when an underlying height-balanced binary tree (in our case AVL-Tree) is used to implement the order statistic tree [7]. Therefore the running timing time of HEliOS is bounded by $O(logn) \cdot size(data)$. We have experimented with the large files from Canterbury corpus which are described in [2] to demonstrate the scalability of HEliOS as shown on Fig. 5(b) and (c). As we can see, the compression time of HEliOS increase by a factor of $Olog(n)$ when file sizes increases, whereas CHT increases by a factor of $O(n)$. Classical Huffman encoding has the lowest compression time without adding any security (i.e., variable encoding and sensitivity) to the encoded data. □

We summarize the efficacy of our method with other encoding methods in Table. 4

Method name	Variable encoding	sensitivity	Scalability
Multiple Huffman Tree[37]	○	○	●
Chaotic Huffman Tree[16]	●	◐	○
Swapped Huffman Tree[20]	●	○	◐
Our method (HEliOS)	●	●	●

Table 4: Comparisons among classical Huffman based compression schemes (●= Yes; ◐= Partially Yes; ○= No)

5 Experimental Evaluation

To evaluate how HEliOS holds up to the challenge of secure fast data transmission, we deploy HEliOS in different experimental real-life scenarios. In this section, we will first explain the experimental scenario setups. Then we summarize our implementation details. We conclude this section by analyzing our experimental results.

5.1 Experimental Scenario Setup

5.1.1 Online Personal Cloud File Uploading and Downloading: Personalized cloud data storage is a special folder on the local storage machine where any digital data placed inside that special folder is being continuously synced to a remote cloud service. Digital data kept inside the special folder is accessible from other devices including laptops and smartphones by downloading them from the remote cloud service. As connectivity is increasing so does the tendency to keep all digital data on the cloud storage among users. With this increasing tendency fast uploading and downloading of the digital data in a secure way without significant resource consumption is a compelling ask. Dropbox is one of the leading market holders in the business of personalized cloud data storage along with other big players such as Google-Drive, Sky-Drive, and iCloud, etc. Our experimental results suggest HEliOS can provide fast secure connectivity for uploading digital data to and downloading from Dropbox remote personalized cloud storage compared to other traditional methods. HEliOS can achieve this fast and secure connectivity while keeping memory and energy consumption at moderate level. We upload and download the five types of digital data (see Table 5) using HEliOS to and from Dropbox personalized cloud server as shown on Fig. 6.

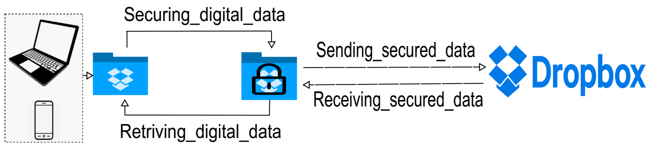


Figure 6: Uploading and downloading digital data to and from Dropbox personal cloud server

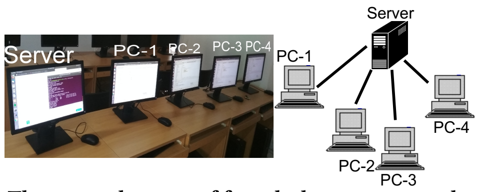


Figure 7: The network set up of four desktops connected to a server in a wired LAN.

5.1.2 Secured File Sharing in a Real Testbed Wired Network: For this experimental setup, we want to know how HEliOS will perform when there are multiple flows for each communicating node in a network. To achieve this, we set up a real test-bed wired network of four desktops as shown on Fig. 7. The four desktops are connected to each other by a server through CAT6 wires in a star topology. All four desktops are running Ubuntu 18.04.02 LTS, 15.5 GiB RAM having Intel Core i7 with 8 cores. The four desktops send and receive digital data in a random order ping-pong fashion among themselves.

5.2 Implementation Details

Device level details: We implement HEliOS on two client devices. One of them is a laptop (Mac-book Pro running on High Sierra Intel Core i5 2.3GHz) and the other one is a smartphone (Xiaomi Redmi 4X running Android-Nougat). **Network level setup:** For uploading and downloading the digital data explained on Table 5 to and from the Dropbox remote personalized cloud storage, we have used Dropbox *API v2* [10]. Both clients are connected through a wireless router (model TP_Link_6528) to an Internet connection which have an uploading and download speed about 1.2 Mbps. **Code level details:** We implement all components of HEliOS by modifying the classical Huffman tree compression and decompression implementation of https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/Huffman.java. Our implementation roughly adds 660 LoC to the above mentioned implementation. **Securing the secret intelligence:** We need to use a strong encryption technique to encrypt the secret intelligence, which should in return secure the whole compressed data. In this regard among popular techniques, we prefer Elliptic curve cryptography (ECC) over widely accepted alternative encryption techniques in research community such as AES and RSA. This is because (1) ECC does not need a key to be shared unlike symmetric key encryption techniques like AES. Sharing a symmetric key before the initiation of communication between sender and receiver in a secure way is itself a complicated problem to address. Besides, ECC achieves the best of near-to- best performance in terms of both time requirement and memory consumption resource constrained devices compared to AES [36]. (2) ECC is more efficient than the RSA technique. It can provide the same security as RSA, however by consuming less computation power than RSA [28]. At code level, to encrypt and decrypt the secret intelligence using ECC,

Data category	Average file size (KB)	# of files	Corpus sources
Audio	4106	500	500 Greatest Songs of All Time [31]
Video	2870	62	Open Preservation Foundation [12]
Ebooks	118	50	Open Preservation Foundation [12]
Office	103	25	Open Preservation Foundation [12]
Emails	4	$0.5M$	Enron email data set [32]

Table 5: Digital data sets used in our experimental evaluations.

we have taken the help of Bouncy Castle's crypto package [3] for the desktop. For the smartphone case, it is the Android repackage Spongy Castle [35]. **Data set details:** We have used five types of digital data as summarized in Table. 5. We do this to verify that our HEliOS method performs better for a variety of digital data formats. **Methods in consideration:** We have considered the following four methods to validate our experimental results. 1) **Traditional way of using compression and encryption:** In this method, the sender secures the digital data by compressing and then encrypting the compressed digital data to make it secure (Compression and encryption). The receiver will decrypt the encrypted compressed data and then decompress it to retrieve the data (Decryption and decompression). 2) **Full encryption-decryption:** The sender encrypts the full data to secure the data (Encryption on full data) and receiver applies full decryption on the encrypted data after receiving (Decryption on full data). 3) **HEliOS:** Our proposed way of securing the data. HEliOS builds an order statistic tree from secret intelligence (Algo. 2) and then encodes the data using the secretly built order statistic tree. (Algo 3). The encoded compressed data is secured by encrypting the small sized secret intelligence using receiver's public key. Only the receiver has the private key to decrypt the encrypted secret intelligence and using it the receiver can decode and retrieve the secured encoded data. On the other hand, without the correct secret intelligence the attacker cannot decode (even partially) the HEliOS encoded data. 4) **Chaotic Huffman Tree (CHT):** We have also experimented by replacing HEliOS with CHT while encoding the digital data to see how CHT will perform. We have not considered MHT and SHT. This because MHT encoding is not secure [18, 40] and SHT is vulnerable to *chosen plain text* attack (presented in Section 3.3). Note that, we only consider the resources (i.e., time, energy, memory) usage on the client side (i.e., laptop, mobile) to secure the digital data and then to send the digital data to the Dropbox personal cloud server while uploading. In the case of downloading, we only consider the resource usage to receive the secured data and then to retrieve the secure data. More specifically, we do not concern ourselves with the resources taken by the remote Dropbox personal cloud server's side. This is because we can safely assume it is equipped with enough resource and processing capacity.

5.3 Analysis of Results

We analyze the different experiment scenarios in terms of average time, JVM (Java Virtual Machine) memory and energy consumption.

Time comparison: Our experimental results show that HEliOS is significantly faster while securing the digital data. As we can see from Fig. 8(a), 9(a) and 10(a), our method gives a much faster average time to secure the digital data for laptop, mobile, and desktop clients respectively. The upload times of the secured digital data are competitive among the first three methods (HEliOS, CHT, Compression-encryption) as illustrated in Fig.9 (b), 9 (b) and 10 (b). This aligns with our expectation since the time to upload the secured digital data depends on the size of secure data itself and the

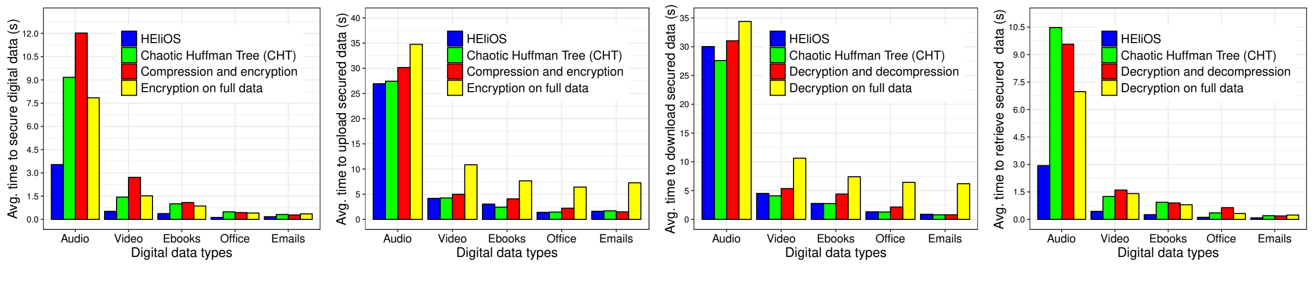


Figure 8: Comparisons of average time for laptop client to (a) secure the digital data, (b) upload the secured data to Dropbox personal cloud server, (c) downloading the secured data from Dropbox personal cloud server, and (d) retrieving the secured downloaded data

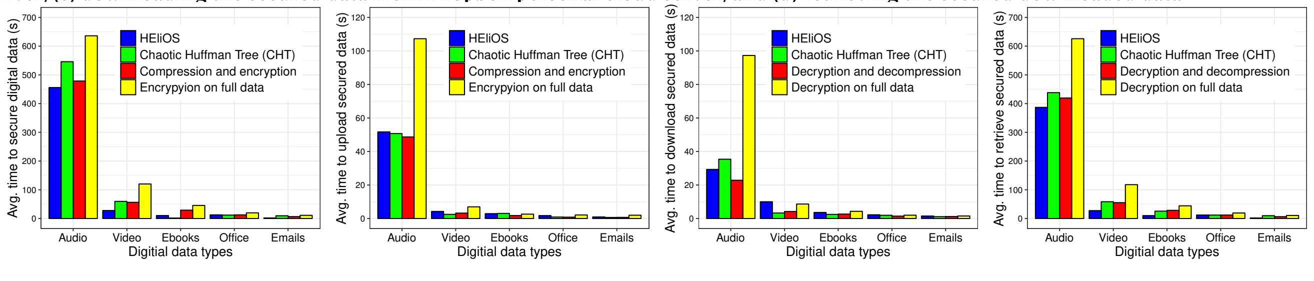


Figure 9: Comparisons of average time for mobile client to (a) secure the digital data, (b) upload the secured data to Dropbox personal cloud server, (c) downloading the secured data from Dropbox personal cloud server, and (d) retrieving the secured downloaded data

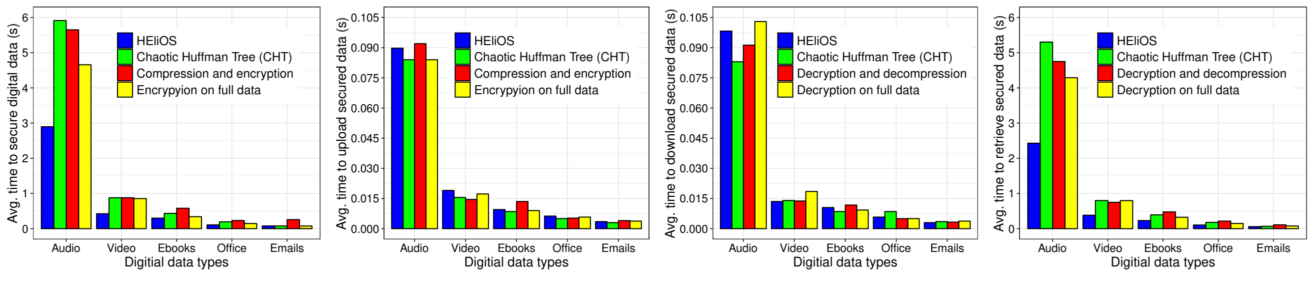


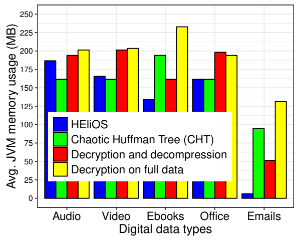
Figure 10: Comparisons of average time for four desktops connected in a LAN to (a) secure the digital data, (b) send the secured data to other connected desktops (c) receive the secured data from other connected desktops, and (d) retrieving the received secured data

first three methods produce secured digital data of almost the same size. For the fourth method, applying full encryption to digital data to make it secure increases the file size which adds to the slowness while uploading the full encrypted secure data upload time. In the same way, we show the time to download the secured digital data for the same set of clients (laptop, desktop, and mobile) respectively in Fig. 8 (c), 9 (c), and 10(c) respectively. As expected the secure data download time, which is dependant on the size of the digital data being downloaded follows the trend of secure data upload time. On the other hand, HELiOS shows significantly better performance time while retrieving the digital data from the downloaded secure data is illustrated in Fig. 8(d), Fig. 9(d) and Fig. 10(d).

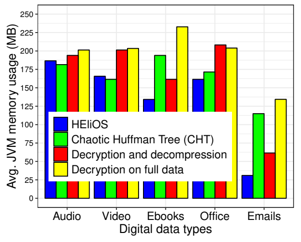
Energy and memory comparison: We also present the average JVM memory usage for laptop and energy consumption of the mobile client. In order to measure the energy consumption on mobile, we have used Trepn Profiler App [27] (available on Google-play store), and to log the memory requirement of laptop we have used an excellent Java class monitoring tool VisualVM [8]. Our experimental results show that HEliOS consumes much less energy than other methods for the mobile client as shown on Fig. 11(c) and 11(d). This result is encouraging because the mobile client is a resource constrained device and also needs fast connectivity, both of which can be facilitated by HEliOS. The JVM memory requirement for laptop is comparable among all methods as shown on Fig. 11(a) and 11(b).

6 Conclusion and Future Work

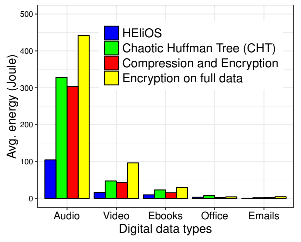
In this paper, we design and implement HEliOS, a novel, fast and secure data transmission mechanism for smart devices. Here, we carefully integrate a novel notion of Huffman compression and encryption by exploiting a synergy between unique properties of both Huffman compression and dynamic order statistic trees. Extensive real experiments in diverse settings demonstrate efficacy of our scheme in terms of multiple performance metrics such as time and energy efficiency. In future, we intend to develop mathematical models for our proposed scheme to enable analysis of its performance in larger scale implementations.



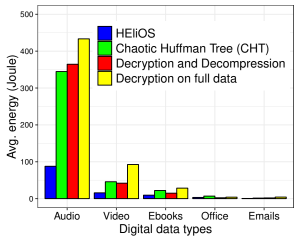
(a)



(b)



(c)



(d)

Figure 11: Comparisons of average JVM memory and energy consumption to secure and upload the secured data to Dropbox personal storage cloud server for (a) laptop (c) mobile client and to download the secured data from Dropbox personal storage cloud server and retrieving the secured downloaded data for (b) laptop (d) mobile client.

7 Acknowledgements

This work was supported by the Information and Communication Technology Division (ICT Division), Government of the People's Republic of Bangladesh. This work was also supported in part by US National Science Foundation (Grant # 1718071). Any opinions, findings and conclusions are those of the authors alone, and do not reflect views of the funding agency.

References

[1] M. Ambrosin, A. Anzanpour, M. Conti, T. Dargahi, S. R. Moosavi, A. M. Rahmani, and P. Liljeberg. 2016. On the Feasibility of Attribute-Based Encryption on Internet of Things Devices. *IEEE Micro* 36, 6 (Nov 2016), 25–35.

[2] R. Arnold and T. Bell. 1997. A corpus for the evaluation of lossless compression algorithms. In *Proceedings DCC '97. Data Compression Conference.* 201–210.

[3] The Legion of the Bouncy Castle Inc. Australian Charity. Last accessed on 2018-09-30. Legion of the Bouncy Castle Java cryptography APIs. http://www.bouncycastle.org.

[4] R. G. Baraniuk. 2007. Compressive Sensing [Lecture Notes]. *IEEE Signal Processing Magazine* 24, 4 (July 2007), 118–121.

[5] R. Beaulieu, S. Treatman-Clark, D. Shors, B. Weeks, J. Smith, and L. Wingers. 2015. The SIMON and SPECK lightweight block ciphers. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC).* 1–6.

[6] T. Bell. Last accessed on 2018-09-30. The Canterbury Corpus. http://corpus.canterbury.ac.nz.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[8] Oracle Corporation and its affiliates. Last accessed on 2018-09-30. VisualVM: All-in-One Java Troubleshooting Tool. https://visualvm.github.io.

[9] P. Cuff. Last accessed on 2018-09-30. Information, Entropy, and Coding. https://www.princeton.edu/~cuff/ele201/kulkarni_text/information.pdf.

[10] The Dropbox API developers. Last accessed on 2018-09-30. Dropbox API v2. https://www.dropbox.com/developers/documentation/http.

[11] Q. Do, B. Martini, and K. Choo. 2017. Is the Data on Your Wearable Device Secure? An Android Wear Smartwatch Case Study. *Softw. Pract. Exper.* 47, 3 (March 2017), 391–403.

[12] Open Preservation Foundation. Last accessed on 2018-09-30. An openly-licensed corpus of small example files. https://github.com/openpreserve/format-corpus.

[13] A. S. Fraenkel and S. T. Klein. 1994. Complexity aspects of guessing prefix codes. *Algorithmica* 12, 4-5 (1994), 409–419.

[14] H. Gan, S. Xiao, and Y. Zhao. 2018. A Novel Secure Data Transmission Scheme Using Chaotic Compressed Sensing. *IEEE Access* 6 (2018), 4587–4598.

[15] D. W. Gillman, M. Mohtashemi, and R. L. Rivest. 1996. On breaking a Huffman code. *IEEE Transactions on Information Theory* 42, 3 (May 1996), 972–976.

[16] H. Hermassi, R. Rhouma, and S. Belghith. 2010. Joint compression and encryption using chaotically mutated Huffman trees. *Communications in Nonlinear Science and Numerical Simulation* 15, 10 (2010), 2987 – 2999. http://www.sciencedirect.com/science/article/pii/S1007570409006108

[17] D. A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (Sep. 1952), 1098–1101.

[18] G. Jakimoski and K. P. Subbalakshmi. 2008. Cryptanalysis of Some Multimedia Encryption Schemes. *IEEE Transactions on Multimedia* 10, 3 (April 2008), 330–338.

[19] K. James and T. Roberto. 2014. Secure Compression: Theory & Practice. *IACR Cryptology ePrint Archive* 2014 (2014), 113.

[20] Y. S. Jang, M. R. Usman, M. A. Usman, and S. Y. Shin. 2016. Swapped Huffman tree coding application for low-power wide-area network (LPWAN). In *2016 International Conference on Smart Green Technology in Electrical and Information Systems (ICSGTEIS).* 53–58.

[21] S. T. Klein, A. Bookstein, and S. Deerwester. 1989. Storing Text Retrieval Systems on CD-ROM: Compression and Encryption Considerations. *ACM Trans. Inf. Syst.* 7, 3 (July 1989), 230–245.

[22] L. Kocarev. 2001. Chaos-based cryptography: a brief overview. *IEEE Circuits and Systems Magazine* 1, 3 (2001), 6–21.

[23] X. Li, L. Bao, D. Zhao, D. Li, and W. He. 2011. The analyses of an improved 2-order Chebyshev chaotic sequence. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, Vol. 2. IEEE, 1224–1227.

[24] H. Liu and X. Wang. 2011. Color image encryption using spatial bit-level permutation and high-dimension chaotic system. *Optics Communications* 284, 16-17 (2011), 3895–3903.

[25] N. Oualha and K. Nguyen. 2016. Lightweight attribute-based encryption for the internet of things. In *Computer Communication and Networks (ICCCN), 2016 25th International Conference on.* IEEE, 1–6.

[26] H. Peng, Y. Tian, J. Kurths, L. Li, Y. Yang, and D. Wang. 2017. Secure and energy-efficient data transmission system based on chaotic compressive sensing in body-to-body networks. *IEEE transactions on biomedical circuits and systems* 11, 3 (2017), 558–573.

[27] A product of Qualcomm Technologies Inc. Last accessed on 2018-09-30. Trepn Power Profiler. https://developer.qualcomm.com/software/trepn-power-profiler.

[28] G. Raju and R. Akbani. 2003. Elliptic curve cryptosystem and its applications. In *SMC'03 Conference Proceedings. 2003 IEEE International Conference on Systems, Man and Cybernetics. Conference Theme-System Security and Assurance (Cat. No. 03CH37483)*, Vol. 2. IEEE, 1540–1543.

[29] R. Rana, M. Yang, T. Wark, C. Chou, and W. Hu. 2014. Simpletrack: Adaptive trajectory compression with deterministic projection matrix for mobile sensor networks. *IEEE Sensors Journal* 15, 1 (2014), 365–373.

[30] E. Setyaningsih, R. Wardoyo, and A. K. Sari. 2018. New Compression-Encryption Algorithm Using Chaos-Based Dynamic Session Key. *International Journal on Smart Sensing & Intelligent Systems* 11, 1 (2018).

[31] E. Lustig T. de Clercq, D. Temperley and I. Ta. Last accessed on 2018-09-30. A Corpus Study of Rock Music. http://rockcorpus.midside.com.

[32] CALO Project (A Cognitive Assistant that Learns and Organizes). Last accessed on 2018-09-30. Enron Email Dataset. https://www.cs.cmu.edu/~./enron/.

[33] L. Touati and Y. Challal. 2015. Efficient cp-abe attribute/key management for iot applications. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing.* IEEE, 343–350.

[34] L. Touati, Y. Challal, and A. Bouabdallah. 2014. C-cp-abe: Cooperative ciphertext policy attribute-based encryption for the internet of things. In *2014 International Conference on Advanced Networking Distributed Systems and Applications.* IEEE, 64–69.

[35] R. Tyley. Last accessed on 2018-09-30. Spongy Castle, repackage of Bouncy Castle for Android. https://rtyley.github.io/spongycastle/.

[36] A. S Wander, N. Gura, H. Eberle, V. Gupta, and S Shantz. 2005. Energy analysis of public-key cryptography for wireless sensor networks. In *Third IEEE international conference on pervasive computing and communications.* IEEE, 324–328.

[37] C. Wu and C. Kuo. 2005. Design of integrated multimedia compression and encryption systems. *IEEE Transactions on Multimedia* 7, 5 (2005), 828–839.

[38] W. Xue, C. Luo, G. Lan, R. Rana, W. Hu, and A. Seneviratne. 2017. Kryptein: A Compressive-sensing-based Encryption Scheme for the Internet of Things. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN '17).* ACM, New York, NY, USA, 169–180.

[39] X. Yao, Z. Chen, and Y. Tian. 2015. A lightweight attribute-based encryption scheme for the Internet of Things. *Future Generation Computer Systems* 49 (2015), 104–112.

[40] Q. Zhou, K. Wong, X. Liao, and Y. Hu. 2011. On the Security of Multiple Huffman Table Based Encryption. *Journal of Visual Communication and Image Representation* 22, 1 (Jan. 2011), 85–92.