

DCBC: A Distributed High-performance Block-Cipher Mode of Operation

Oussama Trabelsi¹^a, Lilia Sfaxi^{1,2}^b and Riadh Robbana^{1,2}

¹Faculty of Science of Tunis, Tunis el Manar University, Tunis, Tunisia

²INSAT, University of Carthage, Tunis, Tunisia

Keywords: Cryptography, Modes of Operation, CBC, Parallel Encryption, Hash-based Encryption.

Abstract: Since the rise of Big Data, working with large files became the rule and no longer the exception. Despite this fact, some data at-rest encryption modes of operation, namely CBC, are being used even though they do not take into account the heavy cost of running sequential encryption operations over a big volume of data. This led to some attempts that aim to parallelizing such operations either by only chaining isolated subsets of the plaintext, or by using hash functions to reflect any changes made to the plaintext before running parallel encryption operations. However, we noticed that such solutions present some security issues of different levels of severity. In this paper, we propose a Distributed version of CBC, which we refer to as DCBC, that uses an IV generation layer to ensure some level of chaining between multiple CBC encryption operations that run in parallel, while keeping CPA security intact and even adding new operations such as appending data without compromising the encryption mode's security. We will, also, make a theoretical performance comparison between DCBC and CBC under different circumstances to study optimal conditions for running our proposed mode. We show in this comparison that our solution largely outperforms CBC, when it comes to large files.

1 INTRODUCTION

Data security is becoming a very competitive field as the strategic value of data as a resource is increasing. However, adding security layers usually adds some performance cost overhead which gets even more noticeable for systems that handle large volumes of data with a relatively high velocity. Such systems are usually qualified as Big Data systems, for which securing data in storage while keeping an acceptable performance is a highly critical requirement, as data must be available for processing as soon as possible.

Moreover, Big Data systems have special characteristics and principles. For instance, increasing storage space usage in favor of enhancing performance and security is acceptable and even encouraged. Also, larger computational resources may be available either locally or in a cluster, so not taking advantage of such assets would be a waste of resources.


To ensure data security, we resort to implementing encryption mechanisms. However, these mechanisms usually use classical modes of operation that


can't present a solution for the previously mentioned requirements while respecting Big Data systems characteristics.

Modes of operation are defined as algorithms used to extend the use of Block Ciphers from the encryption of a single block made of a limited number of bytes, to a plaintext made of a, theoretically, infinite number of bytes. Each of these modes focuses either on performance or diffusion. Diffusion refers to having multiple bits flipped in the output when one or many bits in the input are flipped.

The most simplistic implementation of the currently existing modes is ECB (Pittalia, 2019) which consists on encrypting each block separately and concatenating the results in order to form the final cipher making it highly parallelizable and offering a ciphertext with no additional overhead in size. However, this solution presents multiple security issues and is not recommended in practice except for short messages where additional overhead in the ciphertext's size can be problematic (Stallings, 2010). This makes ECB unusable in a big data environment.

Chained Block Cipher (CBC) mode was suggested to offer some level of diffusion by chaining the different blocks of data (Dworkin, 2005). This ended

^a <https://orcid.org/0000-0002-0792-8763>

^b <https://orcid.org/0000-0002-9786-5961>

up offering multiple security features such as *Semantic Security* (Phan and Pointcheval, 2004), while requiring a sequential execution of the encryption operation as well as an overhead in the ciphertext's size because of the need to save an extra block containing an *Initialization Vector (IV)* used by this mode to ensure semantic security. Even though CBC seems to respect the requirement for securing data in storage by offering the diffusion property as defined in (Katos, 2005), it doesn't quite take advantage of the large computational resources that are usually offered by big data systems as the encryption process is sequential and therefore uses only a single CPU.

Other modes of operation opted for using the underlying Block Cipher as a key generator for a stream cipher. Unlike block ciphers that encrypt the message one block at a time, stream ciphers encrypt it bit by bit until the full message is encrypted. In all these modes of operation this is ensured thanks to the One Time Pad (OTP) that relies on the XOR operation to encrypt the plaintext using a secret key that is only used once per message. The most known modes that work in this manner are: Counter (CTR), Output Feedback (OFB) and Cipher Feedback (CFB) (Stallings, 2010).

When it comes to these three stream ciphers, they are not optimal for securing stored data as they are rarely used in systems where strong cryptographic security is required (Verdult, 2001). For instance CTR presents some security limitations such as malleability (McGrew, 2002) making it unsuitable for securing data at rest. Data at-rest encryption refers to encrypting data that will be persisted on a storage device, as opposed to data-in motion encryption which refers to encrypting data for transmission, generally over a network (Shetty and Manjaiah, 2017).

For the previously mentioned reasons, when it comes to securing data at-rest, multiple security tools opt for using CBC and therefore suffer from the cost of running a sequential encryption across all blocks. For this reason, there have been some attempts to provide parallelizable implementations derived from CBC, either by chaining only a subset of blocks at a time such as with Interleaved Chained Block Cipher (ICBC) (Duță et al., 2013), or by using a hash-based solution to offer a link between the plaintext blocks instead of CBC's chaining (Sahi et al., 2018). In both these cases, some weaknesses, that we will detail in section 5, were introduced in order to allow for parallel execution. For this reason, we looked for a solution that can provide a reasonable trade-off between the chaining level and performance while keeping the mode's security intact.

In this paper, we will suggest a hash-based solution that offers a parallelizable encryption and decryption

process while keeping some level of chaining between the different blocks of the plaintext. First we describe the suggested solution along with the encryption and decryption processes associated to it. Next, we cover some security aspects of the solution and how it behaves in cases where appending or editing data is necessary. Then, we estimate a theoretical cost of running the proposed mode and compare it to the cost of running CBC in multiple scenarios. Finally, we present in more depth other works that are related to this subject and compare them to the results we found for our proposed mode and proceed to a conclusion and some perspectives.

2 PROPOSED SOLUTION

In this paper, we propose a partially parallelizable encryption mode based on CBC and the use of hash functions, which we will be referring to as *Distributed Cipher Block Chaining* or *DCBC*. Before we proceed to describing our proposed solution, we first need to fully understand how *Cipher Block Chaining (CBC)* works.

2.1 Distributed Cipher Block Chaining: DCBC

DCBC will operate on multiple chunks of data in a parallel manner, while using a chaining layer to allow for some level of diffusion between them. Up until now, we have only discussed the plaintext as the concatenation of blocks of data. *Blocks* of data are a subset of the plaintext for which the size is determined by the underlying Block Cipher. For instance, when using the Advanced Encryption Standard (AES) a block of data would refer to 16 bytes of data, however with the Data Encryption Standard (DES) a block refers to 8 bytes of data. When it comes to DCBC, we will be introducing what we refer to as a *Chunk* of data, which is a subset of the plaintext of a size determined by the user. Each of these chunks will be separately encrypted using CBC, so it is recommended to choose a chunk size that is a multiple of the block size used by the underlying Block Cipher, in order to avoid unnecessary padding with each chunk's CBC encryption. As shown in figure 1, the full plaintext message is made of multiple chunks, which in turn can be considered as a series of blocks for which the size is predetermined by the Block Cipher.

In the subsequent sections we will be using the following notations: M_i , H_i and C_i , respectively, denote the plaintext and ciphertext and Hash relative to



Figure 1: The adapted plaintext subsets.

the chunk of index i . IV_i represents the calculated Initialization Vector used to encrypt M_i and IV is the Initialization Vector supplied by the user to DCBC. The operations we will be using are: the hash function H , the CBC encryption and decryption functions E_{CBC} and D_{CBC} , the IV generation function G and finally the encryption using a block cipher for the IV Generation E_G .

In order for DCBC to be a usable mode of operation, it must satisfy some requirements that cover both its security and its performance which we will define in the next section.

2.2 Target Criteria

Our work on DCBC, will focus on four axes.

1. *Semantic Security under Chosen Plaintext Attack*: in a perfect system, the ciphertext should not reveal any information about the plaintext. This concept has been introduced as *Perfect Secrecy* by (Daemen and Rijmen, 1999). *Semantic Security under Chosen-Plaintext Attack*: is a looser and more applicable version of *Perfect Secrecy* as it refers to preventing an attacker from being able to extract any information about the plaintext using only the ciphertext in a polynomial time (Phan and Pointcheval, 2004). To achieve this level of security, an attacker must be unable to link different messages to their respective ciphertexts. As CBC is secure against Chosen Plaintext Attacks (CPA), as proved by (Bellare et al., 1997), DCBC must keep that property intact.
2. *Diffusion*: DCBC should provide some level of chaining to ensure that a slight difference in the input will affect all following blocks in the output.
3. *Parallelizability*: DCBC must be parallelizable to allow for the full use of available resources (CPUs, Cores or Machines).
4. *Secure Append Operations*: DCBC must allow for appending data securely without having to decrypt and re-encrypt the full plaintext.

2.3 How Does DCBC Work?

A message M is divided into multiple chunks of a fixed size. Each chunk can be encrypted using CBC independently from the encryption of all other

chunks, while using a function H to provide a “summary” of the corresponding chunk, which will be used later, to ensure some level of chaining throughout the whole message. This would allow us to run the encryption of the block n and all following blocks on a different CPU as soon as H finishes execution. Also, H runs on the plaintext and is independent of the encryption operation. So, H is a function that will take as input a chunk of data which will have its length be determined by the user and always output a fixed number of bytes that should be representative of the whole chunk taken as input. For this we chose to use hash functions as their properties correspond to these needs. Once the hashes are calculated, an operation is needed to ensure the chaining of the different blocks. We will be referring to it in this article as the **IV Generator**, since its output will be used as an IV for the CBC encryption of the chunk. The properties of this IV generator and the reasons behind them will be discussed in section 2.5, but for now all we need to know is that it takes as input the output of H when ran on the current chunk, as well as the IV of the previous chunk.

To sum up, the mode we are proposing can be viewed as the combination of three layers of processing, two of which are the most costly ones but are fully parallelizable and one is sequential but of very low cost.

The first layer is a hashing operation (**H**), where each chunk will be hashed independently from all previous chunks. The second layer is an IV generation layer (**G**), which generates a pseudo-random IV from the hash of the current layer and the IV generated by the previous layer, thus guaranteeing that any IV depends on all previous chunks. Moreover, the generated IV must be pseudo-random to ensure some security properties that we will discuss later in this article. The third layer is a regular CBC encryption layer (**E_{CBC}**), in which every chunk is encrypted independently from all other chunks’ encryption results, making it parallelizable.

2.4 Encryption and Decryption in DCBC

In this section we will be presenting the encryption and decryption operations at a chunk level as well as at a block level.

We denote by encryption at a chunk level, the expression of the encryption operation as a function using full chunks of plaintext as opposed to the encryption at a block level, where the operation is expressed as a function using single blocks of plaintext.

Figure 2 illustrates how the previously mentioned operations (E_{CBC} , H and G) interact and depend on

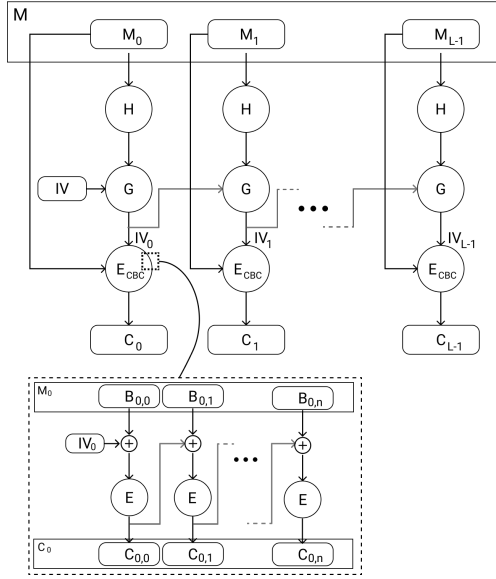


Figure 2: DCBC encryption.

each other for encrypting a message M .

The decryption process, as shown in figure 3, is fully parallel, as each chunk's ciphertext is decrypted independently from all others, using its corresponding IV which was generated and stored in the encryption phase.

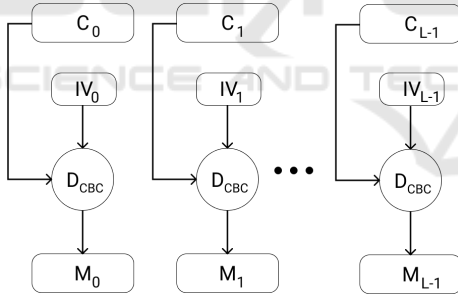


Figure 3: DCBC decryption.

At a chunk level, the resulting equations for encryption and decryption are:

$$C_i = E_{CBC}(M_i, IV_i) \text{ AND } M_i = D_{CBC}(C_i, IV_i)$$

At a block level, the cipher corresponding to the message block $B_{i,j}$, where i is the index of the chunk the message block is situated in and j is its index inside that chunk, is given by the following expressions:

$$C_{i,j} = \begin{cases} E(B_{i,j} \oplus C_{i,j-1}) & \text{if } j > 0 \\ E(B_{i,0} \oplus IV_i) & \text{if } j = 0 \end{cases}$$

The decryption is also as straight forward:

$$B_{i,j} = \begin{cases} D(C_{i,j}) \oplus C_{i,j-1} & \text{if } j > 0 \\ D(C_{i,0}) \oplus IV_i & \text{if } j = 0 \end{cases}$$

In both these cases IV_i is generated as follows:

$$IV_i = \begin{cases} G(H_i, IV_{i-1}) & \text{if } i > 0 \\ G(H_0, IV) & \text{otherwise} \end{cases}$$

Where G is an *IV Generator* respecting the properties we will be detailing in the following section.

2.5 IV Generator

Even though the efficiency of this approach depends heavily on the Block Cipher and the hashing function used for encryption and hashing, its security relies mainly on the security of the IV generation algorithm which is why we propose three main conditions that must be met by any function in order for it to be usable as an IV generator for this mode.

1. The IV of a chunk i must depend on the IV of its previous chunk $i - 1$, in order to ensure some level of chaining between the different chunks. The first chunk is the only exception to this rule, for which an Initialisation Vector (IV) is used.
2. The IV of chunk i must depend on the chunk's hash, to make sure the IV is updated for every update on the chunk, and thus avoid having a predictable/known IV when modifying the contents of existing chunks.
3. The IV generator must be CPA secure to keep the CPA security of CBC intact. This will be detailed further in section 3.1.

For the purposes of this paper, we will be using the following function G :

$$G(H_i, IV_{i-1}) = E_G(H_i \oplus IV_{i-1})$$

Each of the components of the proposed IV Generation function verifies one of the three conditions we just mentioned:

1. H_i : Since each Initialisation Vector IV_i depends on the hash of M_i (H_i), this would ensure the existence of a link between the plaintext and the initialisation vector. One advantage to having this link is that, when M_i is updated, so is IV_i .
2. IV_{i-1} : By using IV_{i-1} in the expression of IV_i , the latter would depend on all previous Initialisation Vectors and therefore on all previous plaintext chunks, since each Initialisation vector depends on the plaintext chunk associated to it thanks to the use of H_i . This ensures the existence of some level of chaining between the different chunks.
3. E_G : Using a Block Cipher, the Initialisation Vector generated will be unpredictable, even when H_i and IV_{i-1} are known to potential attackers, which

is very important to the security of DCBC. The use of this property will be further detailed in sections 3.2.1 and 3.2.2

Notice that this suggestion is not a universal solution. It is designed for a particular case where the output of the hashing operation H is of equal size to the input of the Block Cipher used for CBC encryption. Similarly, we assumed that the Block Cipher used in the IV generation algorithm has an input of the same size as the one used in the CBC encryption. If this is not the case, a slightly more complex IV generation algorithm might be required.

Many of the choices mentioned in this section are adopted to ensure some security properties for our proposed solution. In the next section, we define these properties and explain how they are ensured by DCBC.

3 SECURITY PROPERTIES

3.1 CPA Security

CBC's CPA security was proved by (Bellare et al., 1997), under the condition that the IV is not predictable (Stallings, 2010). In its essence, the proposed mode is formed by multiple CBC encryption operations using calculated IVs instead of randomly generated ones. In this section we will demonstrate that the CPA security of the *IV Generator* G is a necessary condition for the CPA security of DCBC. In order to do this, we will show that if G is not CPA secure, then DCBC is not CPA secure either. Consider the following cryptographic game:

- The Attacker A sends two different messages M_0 and M_1 to the Challenger.
- The Challenger C chooses a message b randomly and return $C_b = E_{DCBC}(M_b)$, where the first blocks of C_b are the IVs used for the encryption.
- The Attacker inspects C_b and outputs $b' \in \{0, 1\}$.

The advantage of A is defined as:

$$Adv_{CPA}(A) = |Pr[b = b'] - Pr[b \neq b']|$$

If the *IV Generator* is not CPA secure, A can use the blocks containing the IVs, to figure out which of the messages M_0 or M_1 has been used to generate them and then conclude which message the cipher represents. This would give A an advantage of $Adv_{CPA}(A) = 1$ and DCBC would not be CPA secure. Therefore, the CPA security of the *IV Generator* is a necessary condition for the CPA security of DCBC.

3.2 Blockwise Adaptive Chosen Plaintext Attack Security

Even though CPA security ensures Semantic Security, it is limited to messages that are presented as an atomic unit, meaning that a message is fully received, fully encrypted and only then is the full cipher returned to the user or potential attacker. Blockwise Adaptive Chosen Plaintext Attack (BACPA), as described by (Joux et al., 2002), removes this constraint and handles non atomic messages. This includes cases where the plaintext message is either sent one or a few blocks at a time, or is appended to old, already encrypted data using the existing cipher's last block as IV for the newly received blocks' encryption.

Attacks against BACPA-vulnerable implementations have already been proved feasible, for instance they have been used to attack some old implementations of SSL as described in (Bard, 2006).

BACPA is a threat in two cases:

1. *Editing existing data*: The plaintext is modified but the old IV is preserved for the new encryption.
2. *Appending new data*: New blocks are encrypted using CBC's logic (C_{i-1} serves as IV for the encryption of C_i) and appended to the existing cipher.

3.2.1 Editing Existing Data

When it comes to modifying existing data, if CBC is used, it is mandatory that the whole message gets decrypted, modified then re-encrypted with a new IV. This is due to the fact that if we only edit the concerned blocks then re-encrypt them using their respective previous cipher blocks as IVs, we would be vulnerable to attacks that abuse predictable IVs.

Using our proposed mode, re-encrypting the whole message is recommended but not mandatory.

Since the IV of a chunk depends on the hash of the chunk itself, once a chunk is updated, its IV is also modified in an unpredictable manner thanks to the use of a Block Cipher. For this reason, it is possible to update only the concerned chunk and the ones following it without having to deal with all previous chunks, while keeping the mode's semantic security intact. However, doing things this way allows attackers to detect the first chunk where changes took place. In case this information is critical to the security of the encrypted data, it is recommended to re-encrypt the full plaintext.

3.2.2 Appending New Data

Chaining the chunks as described in this paper offers security against Adaptive Chosen Plaintext Attacks at a chunk level, meaning that it's possible to append new chunks to the existing data securely without having to interact with any of the old chunks. This is due to the fact that the IV used in each chunk is not deducible from any previously calculated values, including previous IVs, ciphers, hashes, etc ...

In general, an append operation will go through the following steps:

If the size of the last chunk is less than the predefined *Chunk Size*:

1. Decrypt the final chunk. We will consider n to be its index.
2. Append the new blocks to the plaintext of the final chunk.
3. Encrypt the resulting chunk/chunks using the proposed method and by providing the IV of the chunk $n - 1$ to the initial IV Generator.

Otherwise, we get a simple one step process:

1. Encrypt the new chunk/chunks using the proposed method and by providing the IV of the last chunk (chunk n) to the initial IV Generator.

Using these steps to append data, the operation is secure and costs at most one extra decryption operation over a single chunk while retaining the chaining between old and newly added data.

4 THEORETICAL PERFORMANCE COST

In this section we will be running a theoretical performance comparison between the respective costs of using DCBC and CBC to encrypt some plaintext message M .

4.1 Assumptions

In this section we make the following assumptions:

- The time threads take to hash different chunks ($h(CS)$) is only function of the chunk's size.
- The time threads take to encrypt different chunks ($e(CS)$) is only function of the chunk's size.
- The time threads take to generate the IV for chunks of the same size (g) is constant.
- The whole encryption process of a chunk takes place on the same CPU.
- At the start of the operation all CPUs are free.

4.2 Theoretical Performance Comparison

In this section, our goal is to compare the theoretical cost of running DCBC to that of running CBC in different scenarios.

To do so, we need to define a function C representing the cost of computing the encryption of a given chunk starting from the origin, which we define as the point in time when the encryption of the first chunk started. We will denote by i , the index of the chunk we are interested in, so: $i \in [0, L]$.

The expression of $C(i)$ is, $\forall i \in [0, L]$:

Case 1: $(N - 1) \times g \leq e(CS) + h(CS)$:

$$C(i) = (\text{div}(i, N) + 1) \times (h(CS) + e(CS) + g) + \text{mod}(i, N) \times g$$

Case 2: $(N - 1) \times g > e(CS) + h(CS)$

$$C(i) = h(CS) + (i + 1) \times g + e(CS)$$

The proof behind the equations defining the function C is provided in the appendix.

In order to compare the function C to the cost of running plain CBC, we respectively vary the *file size* S , the *chunk size* CS then the *number of CPUs* N . For the fixed variables, we will work with empirically estimated values which requires us to specify the algorithms used for each step of DCBC's execution as well as the definition of a method that allows for a fair estimate of the needed values.

In this section, we will use MD5 for H , AES (Daemen and Rijmen, 1999) for E and E_G . As for G we will use: $G(H_i, IV_{i-1}) = E_G(H_i \oplus IV_{i-1})$.

When it comes to estimating a value empirically, we will use the following method:

1. The operation for which we wish to estimate the cost, is executed 100 times and the duration of each iteration's execution is saved.
2. The mean value is calculated over all recorded values.
3. Outliers are detected using simple conditions and replaced with the calculated mean value: We consider a value to be an outlier if the distance between it and the mean value is higher than 3 times the standard deviation.
4. If any outliers have been found, go to step 2. Otherwise exit and use the calculated mean value as the cost of that operation.

The results shown in table [1], will be used to calculate the theoretical performance of DCBC in different scenarios in order to compare it to the performance of CBC. When it comes to the IV generation, its cost is a constant value since it is independent of the size of the plaintext.

Table 1: Empiric Cost for AES_CBC and MD5.

Size (MB)	E (ms)	H (ms)	S (ms)
128	607.169	196.278	0.004
256	1209.63	386.864	
384	1807.156	558.95	
512	2410.118	771.739	
640	3007.476	930.84	
768	3612.55	1117.435	
896	4219.523	1301.641	
1024	4825.55	1542.107	

As we mentioned earlier, in order to estimate the total cost of running DCBC we will resort to the expression of $C(i)$ which, at the beginning of the section, we defined as the duration it takes to encrypt the chunk of index i since the start of the encryption of first chunk (chunk 0). This means that the cost of encrypting the whole plaintext is equal to the cost of encrypting the last chunk (chunk $L - 1$). So, from this point on, we will be using the expression of $C(L - 1)$ as the cost of encrypting the whole plaintext. Also, as you may notice, the value of g is negligible compared to that of $e(x) + h(x)$, where x represents values from the size column in table 1. This means that for realistic values of N , we will always verify: $(N - 1) \times g \leq e(CS) + h(CS)$ which in turn means that we only need to consider the expression of $C(i)$ according to **Case 1**. We conclude that the cost of running DCBC over the whole plaintext is:

$$C(L - 1) = (\text{div}(L - 1, N) + 1) \times (h(CS) + e(CS) + g) + \text{mod}(L - 1, N) \times g$$

Which is equivalent to:

$$C(L - 1) = (\text{div}(L - 1, N) + 1) \times (h(CS) + e(CS)) + (\text{div}(L - 1, N) + 1 + \text{mod}(L - 1, N)) \times g$$

However, we will need to use the chunk size CS as a parameter instead of the number of chunks L .

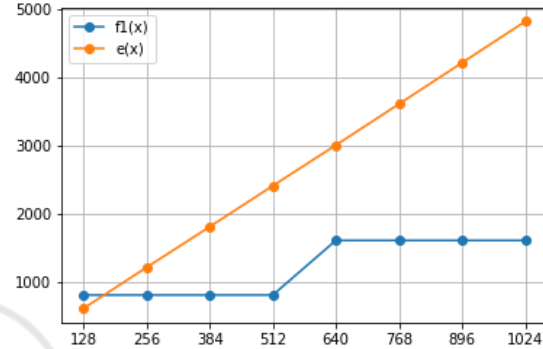
So, for all following sections, instead of using the number of chunks as a constant L , we will be calculating it by ceiling the result of dividing the full plaintext's size by the size of a single chunk: $L = \text{ceil}(\frac{S}{CS})$

4.2.1 Scenario 1: Varying the Plaintext Size

For this case, we will be using a function $f_1(x)$ to refer to the cost of encrypting a file of size x using DCBC, where x is expressed in MB . The expression of f_1 will be:

$$f_1(x) = (\text{div}(\text{ceil}(\frac{x}{CS}) - 1, N) + 1) \times (e(CS) + h(CS)) + (\text{div}(\text{ceil}(\frac{x}{CS}) - 1, N) + 1 + \text{mod}(\text{ceil}(\frac{x}{CS}) - 1, N)) \times g, \forall x > 0$$

Our goal is to compare the cost of running CBC (represented by the function $e(x)$, where x is the size of the plaintext), and DCBC for various file sizes while fixing all other parameters: $N = 4$ and $CS = 128MB$. To do this we start by plotting $f_1(x)$ and $e(x)$, for $x \in \{128, 256, 512, 513, 1024\}$. First, we use the empirically estimated values taken by $e(x)$ listed in table 1. Then, to get the values taken by $f_1(x)$, we use the value of $e(128)$ we got from the previous step as well as the approximation of $h(128)$.


 Figure 4: Performance comparison for various values of S .

The results shown in Figure 4, present some interesting findings: when running the encryption on a plaintext of size $128MB$, DCBC shows a slightly worse performance compared to CBC. This is to be expected since with $128MB$ of data and a *Chunk Size* of $128MB$ as well, DCBC will be using a single chunk and will therefore, not only run in a sequential manner over the whole plaintext, but also suffer from the added weight of the extra hashing operation compared to CBC.

We also notice that f_1 is constant for the interval $[128, 512]$ and is doubled for $x \in]512, 1024]$. This is due to the fact that, in the example we considered, we are using $N = 4$ and $CS = 128$, therefore, for a plaintext of size $x \leq 512$, we will be running a single iteration of encryption. However, for any value of x such that $x \in]512, 1024]$, two iterations are required, which explains why the cost doubles.

When running the encryption on a plaintext of $256MB$ of data or more, we notice that, the larger the plaintext, the higher the difference between the cost of running CBC and that of running DCBC, except for the points where transitions from using n iterations to $n + 1$ iterations take place ($x \in]512, 640]$ in our case), where this difference in performance gets slightly reduced especially for much bigger plaintexts.

In conclusion, DCBC presents a much bigger advantage in performance as the plaintext gets bigger in size, but it is not recommended for cases where the plaintext is of almost equal size to the *Chunk Size* chosen by the user. This leads us to watch how various

values for the *Chunk Size* can affect the performance of DCBC.

4.2.2 Scenario 2: Varying the Chunk Size

We will be using a function f_2 to represent the cost of encrypting a plaintext of a known size S (1024) with 4 CPUs using DCBC with various values for CS :

$$f_2(x) = (\text{div}(\text{ceil}(\frac{S}{x}) - 1, N) + 1) \times (e(x) + h(x)) + (\text{div}(\text{ceil}(\frac{S}{x}) - 1, N) + 1 + \text{mod}(\text{ceil}(\frac{S}{x}) - 1, N)) \times g, \forall x > 0$$

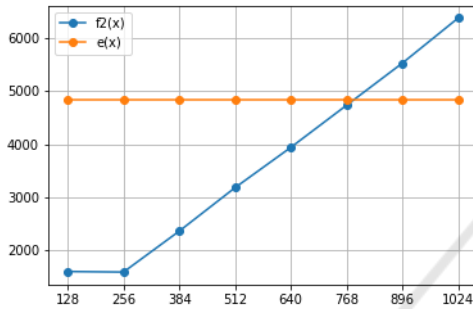


Figure 5: Performance comparison for various values of CS.

Notice that, since CBC does not use chunks, the cost of encryption using CBC, $e(x)$, is independent of the *Chunk Size* and will therefore be represented by a constant function. As shown in Figure 5, the cost of using a *Chunk Size* of 128MB or 256MB is exactly the same. This is explained by the fact that, even though running the encryption over a chunk of size 128MB takes half the time required to encrypt a chunk of 256MB, using a *Chunk Size* of 128MB will require two full iterations of execution over all 4 CPUs whereas using a *Chunk Size* of 256MB will only require one. So even though running DCBC with a *Chunk Size* of 128MB will take only half the time to encrypt a single chunk when compared to using a *Chunk Size* of 256MB, it would in return need to run twice as many times as in the latter case, which would eventually even out the results.

When CS is set to a value of 512MB, we notice that the cost of running DCBC doubles.

Intuitively, this is to be expected since $S = 1024$, therefore the plaintext will be split into only two chunks, which means that only 2 of the 4 CPUs will be used for the encryption as opposed to using all 4 CPUs when $CS \leq 256$. More generally, if the *Chunk Size* is higher than 256MB, only a subset of 4 available CPUs will be used. Also, encryption and hashing over a single chunk will cost more time the higher the *Chunk Size* is. These two factors add up to an overall higher cost when running DCBC with $CS \geq 256$.

Finally, we notice that at a certain point, DCBC presents a worse performance in comparison to CBC, as the cost of the added hashing over a relatively large chunk will surpass the benefit of running the encryption in parallel. In conclusion, the choice of the *chunk size* is important to the performance of DCBC, and even though it seems that, in theory, the smaller the *Chunk Size* is, the better the performance gets, this has to be verified in practice, as many other factors may interfere. We also have to keep in mind that we are working on cases where $(\text{div}(\text{ceil}(\frac{S}{CS}) - 1, N) + 1 + \text{mod}(\text{ceil}(\frac{S}{CS}) - 1, N)) \times g$ happens to be of negligible effect over the results. If this constraint is negated by the use of a very small *Chunk Size* compared to the *full plaintext size*, the value of $\frac{S}{CS}$ might get big enough for the previous expression to have an important impact on the results, even though $g \approx 0.004$.

4.2.3 Scenario 3: Varying the Number of CPUs

We will be using a function f_3 to represent the cost of encrypting a plaintext of a known size $S = 1024$ MB and a fixed *Chunk Size* $CS = 128$ MB using DCBC with various values for the *Number of CPUs* N :

$$f_3(x) = (\text{div}(\text{ceil}(\frac{S}{CS}) - 1, x) + 1) \times (e(CS) + h(CS)) + (\text{div}(\text{ceil}(\frac{S}{CS}) - 1, x) + 1 + \text{mod}(\text{ceil}(\frac{S}{CS}) - 1, x)) \times g, \forall x > 0$$

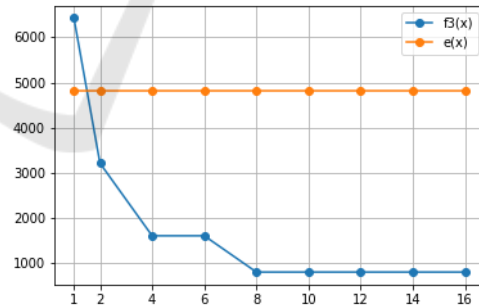


Figure 6: Performance comparison for various values of N.

Notice that the cost of running CBC encryption ($e(x)$) is independent of the *Number of CPUs* used, since it is a sequential process that, in theory, runs on a single process. Therefore e presents a constant function in this case. As Figure 6 shows, when $N = 1$, DCBC presents a much worse performance compared to CBC, as it will be running sequentially with the added weight of using the hash function H .

For $N = 2$, since $S = 1024$ and $CS = 128$ we will be encrypting $L = 8$ chunks of data. Each of the 2

CPUs will be responsible for encrypting 4 of these chunks, which is equivalent to encrypting and hashing 128MB of data 4 times. However, using CBC, the encryption will be sequential over a single CPU for the full 1024MB of data which results in a relatively high difference in performance as the overhead of hashing 128MB of data 4 times is less significant than the difference in time between running CBC over a full 1024MB of data and only 128MB of data 4 times. This result is confirmed by the distance between $f_3(2)$ and $e(2)$, in Figure 6.

For $N = 4$, the cost is reduced by half compared to using only 2 CPUs. However, using 6 CPUs instead of 4 doesn't affect at all the performance of DCBC, this is because in both cases we will have to run a total of 2 iterations.

For $N \geq 8$, we notice that the cost of running DCBC becomes constant. This is due to the fact that at this point we are only running a single iteration of encryption which gives the lowest cost possible, so all extra CPUs are not being used.

Even though these results make perfect sense in theory, in practice there can be a slightly different behaviour.

In a modern system, multiple other processes might be running alongside the DCBC encryption operation. This can cause a slightly higher cost than the theoretical values calculated above. It can also result in a slight improvement of performance when using 6 CPUs compared to the actual cost of using 4 CPUs and further increasing the number of CPUs will help us approach this theoretical value.

In conclusion, the performance of DCBC depends, not only on the computational resources available, but also on the chunk size chosen by the user and the size of the file they look to encrypt.

5 RELATED WORK

Some attempts have been made to reduce the cost of encrypting data through parallelism while keeping some level of chaining between the different blocks.

For instance, the works presented by (Duřa et al., 2013) and (Desai et al., 2013) use Interleaved Cipher Block Chaining (ICBC) to parallelize the encryption process and enhance its performance. ICBC consists on running N independent CBC operations with a randomly generated IV for each one.

In each operation, the chaining will occur on blocks with a step of N blocks. So, the first CBC encryption will run with blocks $0, N, 2N, 3N, \dots$, the second one on blocks $1, N+1, 2N+1, 3N+1, \dots$ and so on. Using ICBC, the more we parallelize the en-

ryption (the bigger N is), the less chained blocks we get. This means that there is a strict trade-off between performance and diffusion.

By comparison, our proposed solution presents a middle ground between performance and diffusion, as chaining is preserved for the whole plaintext no matter how parallelized the encryption gets, but in return, an added cost is present because of the hashing operation.

Other attempts tried to ensure parallelism and diffusion using hash-based solutions, such as the work presented by (Sahi et al., 2018), where the full plaintext is hashed, then the hash ($H(P)$) is used along with the IV and the encryption key K to generate a new key that will be used to encrypt the plaintext block P_i : $Y_i = E(P_i, IV \oplus K \oplus H(P))$.

However, this solution presents multiple issues: **Appending Data.** Just like in CBC, it is not possible to append data directly to the ciphertext. Any append operation would require the full decryption and re-encryption of the whole plaintext. In comparison, in the worst case scenario, our proposed solution only requires the decryption and re-encryption of the last chunk as well as the encryption of the appended data.

Hashing Cost. Even though the hash operation would take considerably less time than the encryption, running the hash function on the plaintext can prove to be very costly for very large files. In this aspect, our solution presents the advantage of parallelizing the hashing operation as well and not just the encryption. **CPA Security.** The presented mode does not offer security against chosen plaintext attacks, since all message blocks P_i are encrypted using the same key and with no IV.

A simple game scenario to show this is as follows: An attacker sends a message $M1$ composed of two identical blocks ($M1_0 = M1_1$) and a message $M2$ made of two distinct blocks ($M2_0 \neq M2_1$). In the resulting Cipher returned by the encryption oracle, if $C_0 = C_1$ then output 0, else output 1. This results in an advantage equal to 1 for the adversary, which means that the attacker can tell which of the two messages $M1$ and $M2$ was encrypted by the oracle, thus, making the discussed solution vulnerable to CPA.

On the other hand, we claim that our proposed solution is CPA secure thanks to the use of unpredictable IVs for the different CBC encryption operations running on the different chunks as explained previously.

6 CONCLUSION AND FUTURE WORK

In its essence, DCBC consists on running a chaining layer on top of multiple CBC encryption operations that run in parallel. This allows for a configurable trade-off between performance and diffusion by manipulating the number of independent CBC encryption operations that we run, which is equivalent to manipulating the Chunk Size, as each chunk will go through its own CBC encryption.

For its security, DCBC inherits the same properties as CBC but requires an equally secure IV Generator to make sure the underlying CBC encryption operations' security will not be compromised. In this article we offered a suggestion for such a function, but it may be possible to find more examples that respect the requirements we specified for a usable IV Generator.

When it comes to performance, the calculated theoretical values show promising results compared to using CBC, and the gap in performance seems to get even more interesting for larger files. This re-enforces the initial idea of adopting DCBC especially in Big Data environments.

Currently, we are looking to validate these theoretical findings with empiric experiments by implementing DCBC and running it on multiple cores locally and/or on multiple machines in a distributed system. Running tests on a working DCBC implementation would, also, allow us to compare the level of diffusion and its randomness, when using DCBC and CBC modes.

Later, we will study the effect of running DCBC on a real Big Data system by implementing its logic in a resource manager such as YARN (Vavilapalli and al., 2013).

Finally, we will look into generalizing the idea used to create DCBC, in order to give a more global solution that offers this trade-off between parallel execution and diffusion using any underlying mode of operation and not just CBC.

REFERENCES

- Bard, G. V. (2006). A challenging but feasible blockwise-adaptive chosen-plaintext attack on SSL. *SECRYPT 2006 - International Conference on Security and Cryptography, Proceedings*, pages 99–109.
- Bellare, M., Desai, A., Jokipii, E., and Rogaway, P. (1997). Concrete security treatment of symmetric encryption. In *Annual Symposium on Foundations of Computer Science - Proceedings*, pages 394–403.
- Daemen, J. and Rijmen, V. (1999). AES proposal: Rijndael.

- Desai, A., Ankalgi, K., Yamanur, H., and Navalgund, S. S. (2013). Parallelization of AES algorithm for disk encryption using CBC and ICBC modes. *2013 4th International Conference on Computing, Communications and Networking Technologies, ICCCNT 2013*, (November 2001).
- Duță, C. L., Michiu, G., Stoica, S., and Gheorghe, L. (2013). Accelerating encryption algorithms using parallelism. *Proceedings - 19th International Conference on Control Systems and Computer Science, CSCS 2013*, pages 549–554.
- Dworkin, M. (2005). Recommendation for Block Cipher Modes of Operation. *National Institute of Standards and Technology Special Publication 800-38A 2001 ED*, X(December):1–23.
- Joux, A., Martinet, G., and Valette, F. (2002). Blockwise-Adaptive Attackers Revisiting the (In)Security of Some Provably Secure Encryption Modes: CBC, GEM, IACBC. In Yung, M., editor, *Advances in Cryptology — CRYPTO 2002*, pages 17–30, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Katos, V. (2005). A randomness test for block ciphers. *Applied Mathematics and Computation*, 162(1):29–35.
- McGrew, D. a. (2002). Counter Mode Security : Analysis and Recommendations. pages 1–8.
- Phan, D. H. and Pointcheval, D. (2004). About the security of ciphers (semantic security and pseudo-random permutations). *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3357:182–197.
- Pittalia, P. P. (2019). A Pros and Cons of Block Cipher Modes for Symmetric Key Algorithms. 1(06):6–9.
- Sahi, A., Lai, D., and Li, Y. (2018). An Efficient Hash Based Parallel Block Cipher Mode of Operation. *2018 3rd International Conference on Computer and Communication Systems, ICCCS 2018*, (iv):212–216.
- Shetty, M. M. and Manjaiah, D. H. (2017). Data security in Hadoop distributed file system. *Proceedings of IEEE International Conference on Emerging Technological Trends in Computing, Communications and Electrical Engineering, ICETT 2016*, pages 10–14.
- Stallings, W. (2010). NIST block cipher modes of operation for confidentiality. *Cryptologia*, 34(2):163–175.
- Vavilapalli, V. K. and al. (2013). Apache hadoop YARN: Yet another resource negotiator. *Proceedings of the 4th Annual Symposium on Cloud Computing, SoCC 2013*.
- Verdult, R. (2001). Introduction to Cryptanalysis: Attacking Stream Ciphers. pages 1–22.

APPENDIX

We aim to calculate a theoretical approximation for the cost of encrypting some plain text M that will be divided into L chunks and encrypted using N CPUs/Cores.

First, we will define $C(i)$ as a combination of two different expressions over two disjoint intervals:

1. $0 \leq i < N$: We will be referring to it as *the first iteration*, during which the first batch of chunks will be encrypted.
2. $i \geq N$, which represents all following iterations.

Then, we will look to provide a unified expression for the cost function over both of these intervals.

First Iteration: $0 \leq I < N$. During the first iteration of encryption operations, all CPUs are free. So, the encryption of a chunk i starts as soon as the chunk $(i-1)$ finishes the IV Generation step (except, of course, for the initial chunk). So, we define $C(i)$ as follows:

$$C(i) = \begin{cases} h(CS) + g + e(CS), & \text{if } i = 0 \\ C(i-1) + g, & \text{if } i \geq 1 \end{cases}$$

We can proceed by induction to show that the previous expression is equivalent to:

$$C(i) = h(CS) + i \times g + g + e(CS), \forall i \in [0, N[\quad (1)$$

Where $i \times g$ represents how long it takes for the previous IV to be calculated using the function G .
Base case($i = 0$): By definition:

$$\begin{aligned} C(0) &= h(CS) + g + e(CS) \\ \iff C(0) &= h(CS) + 0 \times g + g + e(CS) \end{aligned}$$

So $C(0)$ is correct.

Induction Hypothesis: Suppose that there exists $i < N$ such that: $\forall k \leq i, C(k) = h(CS) + k \times g + g + e(CS)$

Induction step: We need to prove that:

$$C(i+1) = h(CS) + (i+1) \times g + g + e(CS)$$

We know that $i \geq 0$, so $i+1 \geq 1$ which gives the following expression of $C(i+1)$:

$$\begin{aligned} C(i+1) &= C(i+1-1) + g \\ &= C(i) + g \\ &= h(CS) + i \times g + g + e(CS) + g \\ &= h(CS) + (i+1) \times g + g + e(CS) \end{aligned}$$

Therefore, $\forall i \in [0, N[$, equation 1 is correct.

Following Iterations: $N \leq I < L$. In general, $C(i)$ can be expressed as the sum of: the duration it took for the current chunk to get on the CPU (which is the same as calculating the cost of encrypting the chunk $i-N$), the hashing duration, the encryption duration, the IV generation duration and, possibly, a waiting duration where the thread is blocked until the previous IV is generated.

This translates into the following formula:

$$C(i) = C(i-N) + h(CS) + wd(i) + g + e(CS) \quad (2)$$

For all iterations except the first one, the waiting duration for chunk i is the difference between

the point in time where the previous IV is calculated, which is expressed by $(C(i-1) - e(CS))$, and the one where the hashing operation for chunk i finishes execution, denoted by $(C(i-N) + h(CS))$:

$(C(i-1) - e(CS)) - (C(i-N) + h(CS))$
 The waiting duration is either a positive value or 0, so instead we define the waiting duration $wd(i)$ as:

$\max(0, [C(i-1) - e(CS)] - [C(i-N) + h(CS)])$
 Since the expression of $wd(i)$ depends on the sign of $(C(i-1) - e(CS)) - (C(i-N) + h(CS))$, then so does the expression of $C(i)$.

When replacing $wd(i)$ by its expression we get:

Case 1: $wd(i) = 0$

$$C(i) = C(i-N) + h(CS) + g + e(CS) \quad (3)$$

Case 2: $wd(i) > 0$

$$C(i) = C(i-1) + g \quad (4)$$

Figure 7, helps visualize the execution scenario of DCBC's threads on each CPU for both cases 1 and 2. * **Case 1:** $wd(i) = 0$

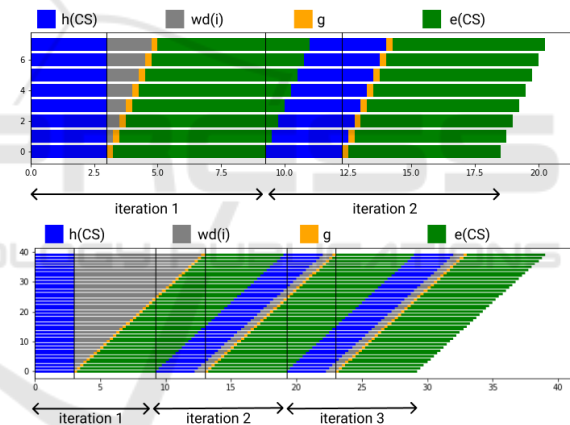


Figure 7: DCBC's execution scenario according to Cases 1 and 2.

Our goal is to prove that equation (3) is equivalent to:

$$C(i) = (\text{div}(i, N) + 1) \times (h(CS) + e(CS) + g) + \text{mod}(i, N) \times g \quad (5)$$

To prove this, we will proceed by induction:

Base case: $i = N$

$$\begin{aligned} C(N) &= C(0) + h(CS) + g + e(CS) \\ &= 2 \times (h(CS) + g + e(CS)) \end{aligned}$$

So $C(N)$ is correct.

Induction Hypothesis: Suppose that there exists an i such that $\forall k \leq i$:

$$C(k) = (\text{div}(k, N) + 1) \times (h(CS) + e(CS) + g) + \text{mod}(k, N) \times g$$

Induction step: We need to prove that $\forall i \geq N$:

$$C(i+1) = ((\text{div}(i+1), N) + 1) \times (h(CS) + e(CS) + g) + \text{mod}(i+1, N) \times g$$

According to equation (3):

$$C(i+1) = C(i+1-N) + h(CS) + g + e(CS)$$

So, we need to distinguish between two cases:

$i+1-N < N$ and $i+1-N \geq N$.

*If $i+1-N < N$, we can apply equation (1):

$$\begin{aligned} C(i+1) &= h(CS) + (i+1-N+1) \times g + e(CS) + h(CS) \\ &\quad + g + e(CS) \\ &= 2 \times (h(CS) + g + e(CS)) + (i+1-N) \times g \end{aligned}$$

However, $0 < i+1-N < N$

$$\iff N < i+1 < 2N$$

$$\iff \text{div}(i+1, N) = 1$$

$$\Rightarrow \text{mod}(i+1, N) = (i+1) - \text{div}(i+1, N) \times N$$

$$\begin{aligned} \Rightarrow C(i+1) &= 2 \times (h(CS) + e(CS) + g) + (i-N+1) \times g \\ &= (\text{div}(i+1, N) + 1) \times (h(CS) + e(CS) + g) \\ &\quad + \text{mod}(i+1, N) \times g \end{aligned}$$

So, equation (5) is correct for $i+1$ if $i+1-N < N$.

*If $i+1-N \geq N$: According to equation (3):

$$C(i+1) = C(i+1-N) + h(CS) + g + e(CS)$$

Since $N \geq 1$ then, $i+1-N \leq i$, therefore the induction hypothesis is applicable to $i+1-N$, which gives us:

$$\begin{aligned} C(i+1) &= (\text{div}(i+1-N, N) + 1) \times (h(CS) + e(CS) + g) \\ &\quad + \text{mod}(i+1-N, N) \times g + h(CS) + g + e(CS) \\ &= (\text{div}(i+1, N) - 1 + 1) \times (h(CS) + e(CS) + g) \\ &\quad + \text{mod}(i+1, N) \times g + (h(CS) + g + e(CS)) \\ &= (\text{div}(i+1, N)) \times (h(CS) + e(CS) + g) \\ &\quad + (h(CS) + g + e(CS)) + \text{mod}(i+1, N) \times g \\ &= (\text{div}(i+1, N) + 1) \times (h(CS) + e(CS) + g) \\ &\quad + \text{mod}(i+1, N) \times g \end{aligned}$$

So, equation (5) is correct for $i+1$ if $i+1-N \geq N$.

By induction, we conclude that, $\forall i \geq N$, equation (5) is correct.

* **Case 2:** $wd(i) > 0$

In this Case, we will be considering the expression of $C(i)$ as given by equation (4).

We notice that in this case $C(i)$ is just an extension of the expression we calculated for the first iteration over the interval $[N, L]$. In conclusion, $\forall i \in [N, L]$:

$$C(i) = h(CS) + (i+1) \times g + e(CS)$$

General Formula. In this section, we will calculate an expression for $C(i)$, $\forall i \in [0, L]$. For the first case: $wd(i) = 0$, we notice that if we apply the expression of $C(i)$, as defined by equation (5), for $i < N$ we get:

$$\begin{aligned} C(i) &= (\text{div}(i, N) + 1) \times (h(CS) + e(CS) + g) \\ &\quad + \text{mod}(i, N) \times g \\ &= (0+1) \times (h(CS) + e(CS) + g) + i \times g \\ &= h(CS) + (i+1) \times g + e(CS) \end{aligned}$$

So, for $i < N$, both equations (5) and (1) are equivalent and we can use equation (5) as a general expression for $C(i)$, $\forall i \in [0, L]$, as long as $wd(i) = 0$.

For the other case, where $wd(i) > 0$, $C(i)$ has the same expression for all iterations. So equation (1) will be the general expression of $C(i)$ under that condition.

In conclusion, we can express $C(i)$ as: $\forall i \in [0, L]$, if $wd(i) = 0$: $C(i) = (\text{div}(i, N) + 1) \times (h(CS) + e(CS) + g) + \text{mod}(i, N) \times g$

otherwise: $C(i) = h(CS) + (i+1) \times g + e(CS)$

Now that the expression of $C(i)$ is determined, we need to simplify the conditional expression $wd(i) > 0$, to have it use initial parameters only.

If we consider $wd(i) > 0$, then in this case:

$$wd(i) = C(i-1) - e(CS) - C(i-N) - h(CS) \quad (6)$$

We will be proceeding by induction to prove that:

$$wd(i) = (N-1) \times g - (e(CS) + h(CS)) \quad (7)$$

Base case: $i = N$

$$\begin{aligned} wd(N) &= C(N-1) - e(CS) - C(0) - h(CS) \\ &= C(N-1) - C(0) - (e(CS) + h(CS)) \end{aligned}$$

Using the expression of $C(i)$ in equation (1), we get:

$$wd(N) = (N-1) \times g - (e(CS) + h(CS))$$

Which means that, $wd(N)$ is correct.

Induction Hypothesis:

$$\forall k \leq i, wd(k) = (N-1) \times g - (e(CS) + h(CS))$$

Induction step: We need to prove that:

$$wd(i+1) = (N-1) \times g - (e(CS) + h(CS)), i \geq N$$

$wd(i) > 0$, then according to equation (4):

$$\begin{aligned} C(i) &= C(i-1) + g \\ \Rightarrow wd(i+1) &= C(i) - e(CS) - C(i+1-N) - h(CS) \\ &= C(i-1) + g - e(CS) - (C(i-N) + g) - h(CS) \\ &= C(i-1) - e(CS) - C(i-N) - h(CS) \\ &= wd(i) = (N-1) \times g - (e(CS) + h(CS)) \end{aligned}$$

So, equation (7) is correct for $i+1$.

By induction, we conclude that if $wd(i) > 0$, then equation (7) is correct $\forall i \in [N, L]$.

In conclusion, $\forall i \geq N$:

$$wd(i) > 0 \iff (N-1) \times g > e(CS) + h(CS)$$

$\forall i \in [0, L]$, the final expression of $C(i)$ is:

<p>Case 1: $(N-1) \times g \leq (e(CS) + h(CS))$: $C(i) = (\text{div}(i, N) + 1) \times (h(CS) + e(CS) + g) + \text{mod}(i, N) \times g$</p> <p>Case 2: $(N-1) \times g > (e(CS) + h(CS))$ $C(i) = h(CS) + (i+1) \times g + e(CS)$</p>
