Acta Scientific COMPUTER SCIENCES

Volume 4 Issue 4 April 2022

Research Article

Character Shift and XOR String Cryptography

Terry Watson, David Montes De Oca and Kanwalinderjit Kaur Gagneja*

Department of Computer and Electrical Engineering and Computer Science California State University, Bakersfield, USA

***Corresponding Author:** Kanwalinderjit Kaur Gagneja, Department of Computer and Electrical Engineering and Computer Science California State University, Bakersfield, USA.

Received: December 14, 2021

Published: March 17, 2022

Abstract

In current computing methods, encryption is widely used and highly valued. Encryption complexity has developed and progressed greatly since the creation of the Internet. This is mainly due to the higher demand for security and privacy amongst the Internet's users. The increase in demand for security has also led to an increase indemand for services capable of breaking this security. That led to the race of encryption vs. code-breaking. Modern encryption methods are highly advanced and difficult to break but come with the caveat of needing higher and higherprocessing power and time to complete, especially on large data sets. Thereby necessitating a new approach. The proposed approach uses the character shift to decrease the time it takes for an encryption approach to complete in a distributed way. This approach requires long strings of characters and shifting them/transforming them to properlyshow that this methodology saves time.

Keywords: Cryptography; Distributed Encryption; Time; Runtime; Code-breaking

Introduction

Modern-day life is prevailed by an abundance of messaging in various forms. Things like Twitter, Facebook, Instant Messaging (IM), and others are a part ofnearly everyone's life. The world is more interconnected and commutative than ever before as a result. This does lead to the necessity for peoples' messages to be as secureas possible [11]. Thus, enters the concept of encryption or: taking a piece of data, shifting it in some way as a preparation to be sent, then sending it. Upon arrival, an encrypted piece of data is then decrypted to show its original form or message. This is great for privacy acrossthe vast sea of information that is the Worldwide Web [7]. The problem arises when the necessary methods for encryption become so bloated that they begin to slow down said communication.

People expect that when a message is sent, the receivergets it as close to instantly. This is becoming more difficultto maintain the speed while maintaining privacy as the methods that have been developed to break encryptions are becoming more and more powerful, and worse yet, are nottime-dependent. The only way to solve this issue is to devise methods to accomplish meaningful, robust encryption in less and less time.

Related work

There is already some research related to this topic. Elena., *et al*. in their paper "Forkcipher: A New Primitive for Authenticated Encryption of Very Short Messages," have demonstrated a method that encrypts short messages quickly [1]. While the intention was to design a framework in which many different encryption methods can be used and operate more quickly. Liu and Tu, in their paper "A control policy for the fork-join queueing network." have presented a control policy for the network to work at its maximum rate and uses minimum buffer [2]. Ammar

and Todd, in their paper, "A design methodology to develop efficient fork-join structures," proposed a structure for shared memory that has parallel architectures [3]. Chandrasekaran., *et al*. in their paper, "A fast and secure image encryption algorithm using number theoretic transforms and discrete logarithms," used logarithms and arithmetic for writing encryption and decryption methods. They also used the XOR operator for generating the random keys [4].

The character shifting method aids in the parallelization and multitasking capabilities of the main program. However, it is also very helpful in applications intended to perform multiple tasks simultaneously that happen to sharethe same or similar data sets. It operates by creating a copyof the process in which it is called, also copying all present variables and existing states [5]. This means that when it'sutilized for performing joint string manipulation, it's actually two separate instances of the same string or character array. This means that even if one of the processes becomes corrupted, fails, or is otherwise compromised, the other process will continue to function to completion [10].

The exclusive or (XOR) operation is used in many ciphers to date. XOR operates on binary data and helps with the creation of strong encryption systems [7]. Thus, itis essentially a fundamental element used in modern ciphers. The XOR algorithm provides us with a verysimple encryption method and allows us to manipulate character arrays into a cipher [8].

Problem formulation

Current encryption methodologies are not very performance friendly as they demand thousands and sometimes millions of computations to achieve their goals [9]. In a high-demand environment, several of the more complex encryption methods are simply unusable for anyone without the capability of outputting a lot of processing power and parallelization [12].

It was the intention to design a framework using the character shift method that could potentially be used to increase the speed of many different encryption methodologies. For this implementation, a simple cryptography approach of letter shifts on the ASCII-text scale is used to show that the implementation does, in fact, decrease run times [6]. It's also intended to implement the ability for the program to decrypt the information that is in the stored files. Figure 1 depicts how the character shift method operates. The parent process is duplicated, resulting in the creation of a parent and child process. Both processes then continue running the same program statements.

Problem formulation

The character shift method is designed as a parent child program statement *S,* where it is split into 2 halves and stored in arrays A and B.The first half's characters $A = A_0A_1..A_{n-1}$ are all shifted upone $A' = A_1A_2..A_{n-1}A_n$ while the second half's characters $B = B_1B_2..B_n$ are all shifted down one $B' = B_0B_1..B_{n-1}$. Thedynamic programming optimization relations are as follows:

$$Obj(S) = A_0A_1..A_{n-1} + B_1B_2..B_n$$

$$= \alpha(A_0A_1..A_{n-1}) + \beta\ (B_1B_2..B_n)$$

$$= A_1A_2..A_{n-1}A_n + B_0B_1..B_{n-1}$$

$$= A' + B'$$

Figure 1: Character shift functionality.

This section presents the technical approach that is used to implement the algorithm.

Modified Ceaser Cipher

A general program is designed that can accept any string within a length of 10000, shift the input into an unreadable output, and then save the output into twoseparate files. The string is taken in by the program, shifted and stored in arrays, then dumped into two separate text files to be viewed and possibly decrypted later.

To show the impact of speed for the character shift functionality, the runtimes are compared of hard-coded versions of

the program with identical, randomized strings of lengths 0, 5000, 10000, 20000, and 30000 with and without character shifting. It shows that the character shifting implemented parallelization increases the speed of computations on strings or arrays of characters and develops a framework in which two separate encryption algorithms can act on the same input simultaneously. This will allow the in put to be more secure in the long run. As malicious agents successfully decrypt one of the algorithms used to encrypt the message/string, the other half of the input will still be safe. This will essentially double the work malicious agents have to do to successfully decipher transformed input. Figure 2 below displays a flow chart of the programs' general flow.

For this proof-of-concept implementation, the input is being split in two halves. The first half's characters $A = A_0A_1..A_{n-1}$ are all shifted up one $A' = A_1A_2..A_{n-1}A_n$ while the second half's characters $B = B_1B_2..B_n$ are all shifted down one $B' = B_0B_1..B_{n-1}$.

For the decrypt portion of the program, the algorithm is essentially the same but in reverse. This simple character shifting serves as an excellent calculation to compute several times to allow for a thorough display of the framework's capabilities. Because the proposed work is related to character shifts, so required to work with the ASCII table to ensure proper output is being produced.

The issue of the ends of the generally used ASCII characters arose. These are the 32nd and the 126th characters on the list, or [space] and '~'. Below 32nd and above the 126th character, ASCII characters are not displayable within a text file. This is because the 31st ASCII character is reserved for the 'Unit Separator' character, and the 127th ASCII character is reserved for the DEL character. It requires extra precaution when dealing with these characters. Without taking extra measures, a text input including a [space] would result in the decryption algorithm cutting off a piece of the string due to the unit separator.

To avoid this problem when decrypting a string, an if statement is added to check for instances where the ASCII value is equal to 31 (the unit separator) as a result of the encryption. If a value of 31 is detected, then this value is changed to 32, which is a [space] character. This change allows the program to decrypt the string successfully.

Despite this, the characters can still be used and outputted to text files; they just don't represent what they were originally intended for. However, in general, these ASCII characters do not usually get used for their original intention regardless [3]. Below in figure 3, an ASCII table can be found and referred to for a representation of what inputs will be converted to depending on where in a string they are located.

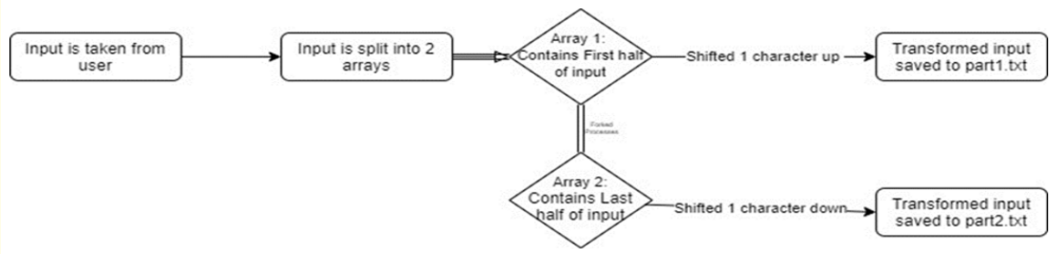

Figure 2: Flow chart of program's general workflow.

XOR Cipher

A second program is written to further test the character shifting operation in hopes that it will speed up other methods of encryption as well.

The Exclusive OR (XOR) operation is a "fundamental building block of practically all modern ciphers" [8]. This operation essentially compares two bits and outputs true if both bits are opposites. Otherwise, the output is false. Figure 4 shows an example of how the XOR function operates. Consider a set A and a binary operation ^ defined on the set, making an object (A, ^). The defined object holds the following properties:

- Performing binary operation on A will always produce an element of B.

- There exists an identity element 0. When it's combined with any element of B, nothing changes.
- The binary operation, ^, is associative.
- There exists an inverse for each element of B, such that when combined it produces an identityelement.

For the XOR operations that are applied to set A as Boolean vectors A = {1, 0}^{J}, then this set of vector can only take input values as 1 or 0 or Ture or False, where thevector length is J.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	⌂

Figure 3: ASCII values.

Input		Output
A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4: XOR ⊕ operations.

Code Snippet I: XOR encryption algorithm

```
//xorCipher function performs the XOR operation on a
//character string using the pre-defined key
void xorCipher(char* inputStr)
{                              /
   char key = 'X';
   int length = strlen(inputStr);
// Loop through input string and perform XOR
//operation with the key
   for (int i = 0; i < length; i++)
   {
      inputStr[i] = inputStr[i] ^ key;
   }
}
```

Figure a

Code Snippet II: System call to character shift method

```
// character shift system call
        pid_t pid = CharacterShift();
//Checking if the pid for the current process == 0,
//meaning it's the child
        if (pid == 0)
        {
        // child process
           ofstream myfile;
//opens a file to store the first half of the encrypted text
           myfile.open ("part1.txt");
           // char array of half the size of counter
           char texta[counter/2+1] = {'\u0000'};
// Deposits first half of the text[] array into texta[] array
           for(int i = 0; i < counter/2; i++){
              if(text[i] != '\u0000') {
                 texta[i] = text[i];
              }
           }
// Calls xorCipher function and uses texta array as input
           xorCipher(texta);
// Stores encrypted text into part1.txt
           myfile << texta;
           cout << endl << "First half encrypted saved
in part1.txt" << endl;
```

```
           myfile.close();
// Stopping clock and calculating runtime
           auto stop = high_resolution_clock::now();
           auto duration = duration_cast
<microseconds>(stop - start);
           cout << endl << "Duration: " <<
duration.count() << " microseconds" << endl;
           exit(0);
        }
```

Figure b

The decryption of the XOR is quite simple as long as the user has the key that was used to encrypt the text. To decrypt, simply take the previously generated ciphertext and call the XOR function once more to generate the original string as shown in code snippet III. Essentially a cipher is equal to its own inverse when using XOR.

Code Snippet III: Decryption algorithm

```
// Concatenating the first and second half of the
decrypted array to create a single result array
        for (int j = 0; j < strlen(p1_text); j++)
        {
           result[j] = p1_text[j];
        }
        for (int k = strlen(p1_text); k < text_length;
k++)
        {
           result[k] = p2_text[c];
           c++;
        }
        cout << "\nDecrypted text: ";
// Outputting decrypted text
        for (int s = strlen(result); s >= 0; s--)
        {
           cout << result[s];
        }
```

Figure c

Performance measure

The processors have to do work, W, to measure the performance. A processor communicates with other processors, W_c, involving data retrieval from memory. Any processor work also involves execution and overhead work. Overhead work, W_o, involves memory allocation to variables, queueing or stacking of variables, garbage collection, etc. However, the execution work, W_e, involves assigning values to various variables, doing computations, and evaluating statements [12]. T_m calculates the minimum execution time.

It is assumed that to avoid conflict all communications are performed serially. The minimum time, T_m, required for processing is $W_c + 1$, as only one computation can precede one communication [12]. It implies that only W_c processors can be used, as each module requires to access shared memory once they complete their computation. The threshold number of processors that would be sufficient for communication work and execution work is given as

$P_c = \lceil (W_c + 3/2) - \text{sqrt}(\text{pow}((W_c + 3/2),2) - 2*(W_c+W_e)) \rceil$ ----------(1)

When these processors work together, there are some overheads and are computed as follows:

$W_o = (W_c - P_c)(P_c - 1) + P_c * (P_c + 1)/2 - W_e$---------(2)

Results and Analysis

All runtimes are taken from the Unix-based Odin server available to California State University - Bakersfield Computer Science students. Times are collected using the C++ Chrono library's high_resolution_clock. Results were recorded in microseconds.

Caesar Cipher algorithm time results

Table 1 below presents the information regarding the time results without character shifting. Table 2 presents the information regarding the time results with character shifting. Table 3 shows the average difference between the two methods, as well as the percentage of time saved using the character shifting implementation.

Characters	Time 1 (µs)	Time 2 (µs)	Time 3 (µs)	Time 4 (µs)	Time 5 (µs)	Average Time
5000	525	498	558	550	490	524.2
10000	528	511	529	520	563	530.2
20000	938	960	900	952	923	934.6
30000	1254	1236	1205	1220	1200	1223

Table 1: Time results without CharacterShift ().

Characters	Time 1 (µs)	Time 2 (µs)	Time 3 (µs)	Time 4 (µs)	Time 5 (µs)	Average Time
5000	479	525	474	462	482	484.4
10000	481	495	508	492	554	506
20000	873	866	877	840	879	867
30000	1082	1071	1061	1087	1102	1080.6

Table 2: Time results with CharacterShift().

Characters	Table 1 Avg – Table 2 Avg (µs)	Percentage of time saved with CharacterShift()
5000	39.8	7.59%
10000	24.2	4.56%
20000	67.6	7.23%
30000	142.4	11.64%

Table 3: Time difference without and withCharacterShift().

The following formula is used to calculate the percentage of time saved when using the character shiftingimplementation.

$$X = (Time_{without\_charShift} - Time_{with\_charShift}) \div Time_{without\_charShift}$$

From table 3, it is clear that the CharacterShift() implementation is much more efficient in its utilization of resources. An interesting point to note is the apparent increase in the efficiency disparity as the number of characters in the string increase. This could be primarily due to the nature of the CharacterShift() method and how exactly it saves time. The CharacterShift() method is so useful because it allows for multitasking. So, time is savedon every iteration of the algorithm. This means that as the string size increases, the methodology becomes more andmore useful.

Another interesting note on the data is the potential ofan outlier within the 10000 Character tests. In table II, time 5, the processing time is a whopping 554 µs. That value is significantly higher than the surrounding values of the same test run. This is probably due to the server loadon Odin at the time being slightly higher than at the otherruns' times.

With this potential outlier thrown out, the average time of the runs for 10000 characters is reduced down to 494 µs, with a difference of 36.2 µs which is significantly higher than the recorded average difference of 24.2 µs.

In terms of a percentage difference in the time, it would equate to 6.83% rather than the 4.56% recorded. Therefore, a 2.23% difference could have been introducedby this potential outlier.

XOR algorithm time results

The same tests are executed on the XOR algorithm to test the performance increase when using a CharacterShift() method. The tests consist of the 5000, 10000, 20000, and 30000-character input strings and the time in microseconds in which the program took to complete.

Table 4 below contains the information regarding the time results using the XOR cipher without CharacterShift() method. Table 5 contains the information regarding the time results using the XOR cipher with a CharacterShift() method.

Table 6 shows the average difference between the two methods, as well as the percentage of the time, saved using the CharacterShift() implementation. Table 7 describes the difference in time saved between the Caesaralgorithm and the XOR algorithm.

An enormous improvement in speed is observed as the input character size increases. As mentioned before, this is due to the CharacterShift() method's ability to run a parent and a child process simultaneously, where each performs the operations on their halves of the input string.

There are a couple of outliers in this data where other circumstances may have impacted the performance of thealgorithm. More so, one of the recorded times for the 30,000-character test in table 4 shows 3790 µs, and the average is 3189.6 µs. Another outlier appears in table 5 under the 30,000-character test being 2057 µs while the average is 1967.4 µs.

When the outliers are disregarded, the averages for the table 4 and table 5 entries become 3039.5 µs and 1945µs, respectively. The time saved by using the CharacterShift() method is 36% which is very impressive.

Characters	Time 1 (µs)	Time 2 (µs)	Time 3 (µs)	Time 4 (µs)	Time 5 (µs)	Average Time
5000	817	829	801	1037	838	864.4
10000	1237	1333	1246	1259	1354	1285.8
20000	2184	2228	2247	2147	2127	2186.6
30000	3014	3122	3008	3790	3014	3189.6

Table 4: Time results without CharacterShift() usingXOR cipher.

Characters	Time 1 (µs)	Time 2 (µs)	Time 3 (µs)	Time 4 (µs)	Time 5 (µs)	Average Time
5000	780	836	783	778	790	793.4
10000	1085	1057	1037	1124	1057	1072
20000	1493	1454	1509	1549	1448	1490.6
30000	1941	1960	1933	1946	2057	1967.4

Table 5: Time results with CharacterShift() using XORcipher.

Characters	Table 4 Avg –Table 5 Avg (µs)	Percentage of timesaved with CharacterShift()
5000	71	8.21%
10000	213.8	16.62%
20000	696	31.83%
30000	1222.2	38.31%

Table 6: Time differences without and withCharacterShift() using XOR cipher.

Characters	Percentage time saved withshifted Char	Percentage oftime saved with XOR	Percentage difference in time saved
5000	7.59%	8.21%	0.62%
10000	4.56%	16.62%	12.06%
20000	7.23%	31.83%	24.6%
30000	11.64%	38.31%	26.67%

Table 7: Time saved by both algorithms and theirdifferences.

Now, when comparing the time differences from table 3 and table 4. It is observed that the time saved from using XOR cipher increases at a greater rate than thatof the character shift algorithm as the input takes in morecharacters. This is likely due to the XOR operation being slightly more computationally expensive when largerinputs are being processed.

Table 7 presents a disparity in the time saved. The percentage of time saved is listed for each for the variousarray sizes, and the difference is essentially (XOR percentage time saved – character shift percentage time saved).

Conclusion

In conclusion, character shifting is an excellent method of splitting encryption-type calculations or processes into smaller, concurrent jobs, allowing for much faster processing and completion of the overall encryption task. Another important deduction is the endless possibilities for splitting the encryption types or methods within the same data, allowing for increased security.

An important thing to note is the time differences between the original character shift implementation and the XOR-Encryption implementation. The improvement in execution time shows that the time saved with the XOR algorithm with a character shift as compared to the original simple implementation of the algorithm works ina much more exponential way in relation to the number ofcharacters. The primary reason for this could be the necessary computational power disparity between the two algorithms. In comparison to a simple character shift encryption, the XOR algorithm is much more demandingcomputationally. This would account for the seemingly large disparity in the time saved by the character shifting between the two algorithms. Overall, this shows that the character shifting implementation is more useful for large,repetitive, and computationally demanding task sets or large tasks that can be easily separated into small subtasks.

Future Directions

With this methodology, similar applications could be designed that utilize much more complex encryption functions or methods such as hashing integers with division hashing, Fibonacci hashing, or multiplicative hashing. In this instance, the CharacterShift() implementation was faster at almost all character array sizes, except for perhaps incredibly small arrays (i.e., 1-50 characters), because of the relatively low overhead of spawning a new process using character shifting. It was especially fast for character arrays of arbitrarily large sizes. This would most likely prove even more true should a more complex encryption function be implemented, asthe processing per character would likely increase.

Bibliography

1. Elena Andreeva., *et al*. "Forkcipher: A New Primitive for Authenticated Encryption of Very Short Messages". *Lecture Notes in Computer Science Advances in Cryptology - ASIACRYPT* (2019): 153-182.

2. Liu Ruihua and Tu Fengsheng. "A control policy for fork-join queueing network". Proceedings of 1995 34th IEEE Conference on Decision and Control (1995): 3648-3649 vol.4.

3. R A Ammar and R Todd. "A design methodology to develop efficient fork-join structures". *Proceedings Second IEEE Symposium on Computer and Communications* (1997): 601-608.

4. J Chandrasekaran and T S Jayaraman. "A fast and secure image encryption algorithm using number theoretic transforms and discrete logarithms". 2015 IEEE International Conference on Signal Processing, Informatics, Communication and Energy Systems (SPICES) (2015): 1-5.

5. "Linux Programmer's Manual" (2020).

6. "ASCII Table and Description". Ascii Table - ASCII character codes and html, octal, hex and decimal chart conversion (2020).

7. L Wagner. "Why is Exclusive Or (XOR) Important in Cryptography? - Qvault". Qvault (2020).

8. M Lewin. "All about XOR". ACCU professionalism in programming 20.109 (2012): 14-19.

9. Gagneja KK and Nygard K. "Key Management Scheme for Routing in Clustered Heterogeneous Sensor Networks". IEEE NTMS 2012, Security Track, Istanbul, Turkey): 1-5, 7-10 May (2012).

10. Tirado E., *et al*. "A New Distributed Brute-Force Password Cracking Technique". Future Network Systems and Security, FNSS Communications in Computer and Information Science 878 (2018): 117-127.

11. K K Gagneja., *et al*. "Tabu-Voronoi Clustering Heuristics with Key Management Scheme for Heterogeneous Sensor Networks". IEEE ICUFN 2012, Phuket, Thailand (2012): 46-51, July 4-6.

12. Kanwalinderjit Kaur Gagneja and Riley Kiefer. "IoT Devices with Non-interactive Key Management Protocol". 2020 Sixth International Conference on Mobile And Secure Services, MobiSecServ 2020, Miami, USA, (2020).

13. Todd Robert. "Performance-Based Quality Measures for Parallel Software Design". Phd Dissertation, in preparation, (1994).