IJCCS (Indonesian Journal of Computing and Cybernetics Systems)
Vol.17, No.3, July 2023, pp. 227~236
ISSN (print): 1978-1520, ISSN (online): 2460-7258
DOI: https://doi.org/10.22146/ijccs.86118

C Source code Obfuscation using Hash Function and Encryption Algorithm

Sarah Rosdiana Tambunan*[1], Nur Rokhman[2]
[1]Master Program in Computer Science, FMIPA UGM, Yogyakarta, Indonesia
[2]Department of Computer Science and Electronics, FMIPA UGM, Yogyakarta, Indonesia
e-mail: ***[1]sarah.rosdiana.tambunan@mail.ugm.ac.id**, [2]nurrokhman@ugm.ac.id

Abstrak

Obfuskasi penting untuk dilakukan karena beberapa alasan, diantaranya meningkatkan keamanan code, melindungi hak cipta dan menguji ketahanan antivirus. Obfuskasi adalah teknik untuk mentransformasi kode program menjadi bentuk yang berbeda dengan mempertahankan semantik dan behavior-nya. Ada beberapa metode obfuskasi yang sering digunakan untuk obfuskasi source code diantaranya adalah deadcode insertion, code transposition dan string encryption. Pada penelitian ini dilakukan pengembangan sebuah obfuscator yang dapat bekerja pada source code bahasa C dengan metode code transposition yaitu mengacak susunan baris kode dengan fungsi hash kemudian menggunakan algoritma enkripsi DES untuk menyembunyikan parameter dari fungsi hash tersebut sehingga semakin sulit untuk menemukan format asli nya. Obfuscator yang dikembangkan memiliki routine untuk melakukan invers terhadap fungsi hash yang bekerja sehingga file yang diobfuskasi dapat dikompilasi secara normal. Obfuscator ini secara khusus digunakan untuk menjaga keamanan source code dalam bahasa C dari plagiarisme maupun pembajakan. Untuk mengetahui apakah obfuscator telah berjalan sesuai dengan harapan, maka dilakukan pengujian kualitatif terhadap dissimilarity dan potency. Dissimilarity diuji dengan menggunakan Euclidean Distance sementara Potency diuji oleh 9 orang programmer untuk melihat apakah source code yang telah diobfuskasi dapat dipahami oleh programmer tersebut. Hasil pengujian yang diperoleh menunjukkan bahwa obfuskasi dengan fungsi hash menghasilkan nilai dissimilarity yang lebih baik jika dibandingkan dengan metode obfuskasi yang telah berkembang, selain itu obfuscator juga secara efektif mampu menjaga keamanan code dan membuat analisis terhadap kode menjadi lebih rumit.

Kata kunci *– Obfuscator, source code, hash*

Abstract

Obfuscation is a technique for transforming program code into a different form that is more difficult to understand. Several obfuscation methods are used to obfuscate source code, including dead code insertion, code transposition, and string encryption. In this research, the development of an obfuscator that can work on C language source code uses the code transposition method, namely randomizing the arrangement of lines of code with a hash function and then using the DES encryption algorithm to hide the parameters of the hash function so that it is increasingly difficult to find the original format. This obfuscator is specifically used to maintain the security of source code in C language from plagiarism and piracy. In order to evaluate this obfuscator, nine respondents who understand the C programming language were asked to deobfuscate the obfuscated source code manually. Then the percentage of correctness and the average time needed to perform the manual deobfuscation are observed. The evaluation results show that the obfuscator effectively maintains security and complicates the source code analysis.

Keywords— *Obfuscator, source code, hash*

Received June 23th,2023; Revised July 30th, 2023; Accepted July 31th, 2023

1. INTRODUCTION

Intellectual property, especially algorithms, is an asset that is expensive to develop and risks being stolen. Cases of theft often come from the field of cryptography, where many algorithms and protocols have been damaged due to the high interest in cryptography which sacrifices easily accessible assets [1]. This attack occurs because the owned code is relatively easy to read, so piracy, modification, and misuse of the code often occur.

Obfuscation is a technique for changing the structure of source code or executable files to make them challenging to understand without changing the semantics of the files [2]. Obfuscation will be performed on source code files, especially those written in the C programming language. In general, obfuscation methods that are often used today are packing, polymorphism, and metamorphism methods. Packing is an obfuscation method that hides the actual code where the contents of the executable file have been altered using some mechanism and must be restored to their original form before being executed [3]. Polymorphism is an obfuscation method that uses several transformations, such as adding nop-code, code transposition (changing the order of instructions and placing jump instructions to maintain the original semantics), and register reassignment, namely changing the arrangement of register allocations [4]. Metamorphism is a method of obfuscation that allows the malware to duplicate itself; the goal is to avoid detection from detectors because each duplicate generated has a different signature, and it is impossible to store multiple signatures for a single piece of malware [4]. Specifically for source code obfuscation, the methods often used are dead code insertion, code transposition, and string encryption. Deadcode insertion is done by inserting a block of code that will never be reached so that it will never be executed but will increase the code size and the analysis time [5]. Code transposition, which is included in layout obfuscation, is done by changing the identifier, changing the format, or deleting comments [6]. String encryption is done by hiding strings that are considered to contain sensitive information using symmetric encryption algorithms such as DES, AES, or XOR [5].

In this research, an obfuscator will be built that works on C language source code using the obfuscation layout method, especially the transformation of the code format. Format transformation is performed using a hash function where the parameters of the hash function are hidden using the DES encryption algorithm. Obfuscated files will be read per line by the obfuscator to be subjected to the hash function. The hash function will be a pseudo-random number generator (PRNG) to provide a random number. The random number will be used to determine the position of the line code after obfuscation. The output of this obfuscator is a new source code file whose lines have been scrambled.

The obfuscation in this research will help anyone with intellectual property in the form of C source code to protect their files from being hijacked or modified by other irresponsible parties.

2. METHODS

This section will discuss the problem analysis, architecture, or design method used. The portrayal of the stages used in this study is shown in Figure 1

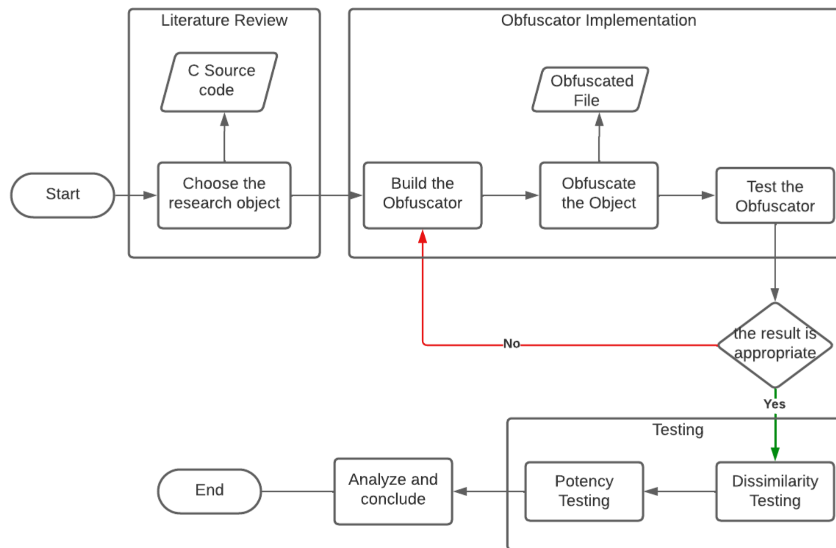

Figure 1 Research workflow

As shown in Figure 1, the first stage that is carried out is the selection of research objects, namely source code samples written in the C programming language. Objects which are source code files are obfuscated using the layout obfuscation method, which is randomizing lines of code using a hash function with parameters hidden using an algorithm. DES symmetric. The obfuscator will generate a new source code file with the same format, namely a source code file with the C programming language (.c).

The output file obtained is then tested to ensure the new file is scrambled correctly. This test will involve nine respondents who will test the obfuscator with three different hash algorithms. Finally, the respondent will try to rebuild the obfuscated simple C file to its original form and then observe the time it takes and the number of lines that have been successfully repaired.

2.1 Problem analysis

The obfuscator currently being developed uses packing techniques using the XOR algorithm and string encryption as the primary defense. However, as technology develops, the tools for deobfuscator or unpacker are also growing so that it is no longer efficient enough to use simple algorithms, so this research developed an obfuscator that works on C language source code using the layout obfuscation method, especially format transformation using three hash functions which are and the DES encryption algorithm. This obfuscator will disguise the code structurally but still maintain the original behavior of the code by hiding the deobfuscation routine so that it can still work like the original file.

2.2 Architecture

In general, the built obfuscator's architecture is shown in Figure 2.

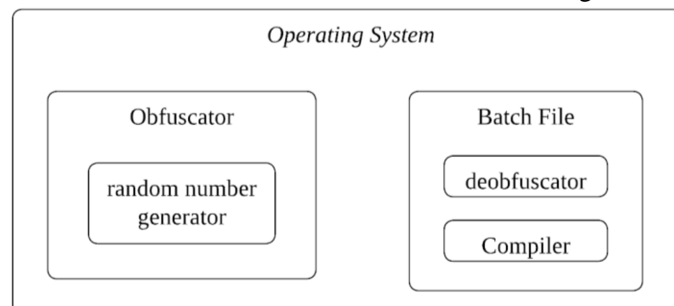

Figure 2 System Architechture

Based on the architecture in Figure 4.2, there are several essential components, namely:

1)	Obfuscator	:	Program to perform obfuscation, consisting of a random number generator and randomization algorithm
2)	Random Number Generator	:	Program to generate random numbers with three different types of hash algorithms
3)	Batch File	:	The batch program consists of instructions for deobfuscating and compiling.
4)	Deobfuscator	:	A program to deobfuscate files that have been obfuscated
5)	Compiler	:	Program to compile source code

Random number generator diimplementasikan dengan menggunakan algoritma hash yaitu MD5, SHA256, dan Blake2.

1) Message Digest 5 (MD5)

Message Digest Algorithm 5 is a hash algorithm invented by Ron Rivest in 1991 to replace MD4, which was successfully attacked by cryptanalysts [7]. MD5 is capable of processing 512-bit blocks and producing 128-bit digests. The input block is divided into 16 blocks with a size of 32 bits each, and then the output is set into four blocks which, after being combined, form a 128-bit digest. The MD5 algorithm works on an iteration process where messages with a bit size iterate like the following formula [7]:

$$H_{i+1} = f(H_i, M_i), 0 \le i \le t-1 \quad (1)$$

With $M = (M_0, M_1, \dots, M_{t-1})$ and $H_0 = IV_0$ which is the hash function's initial value (seed).

2) Secure Hash Algorithm (SHA256)

The Secure Hash Algorithm (SHA) was developed by NIST based on the MD4 function and published as a federal information processing standard in 1993 [8]. SHA-256 is an extension of SHA with a digest size of 256 bits. SHA-256 uses basic logic: AND, OR, XOR, shift right, and rotate right [9]. Three properties make SHA-256 a secure hash algorithm. First, it is almost impossible to find the initial value of the hash. Second, having two messages with the same hash value (collision) is doubtful because of the tremendous possible hash value i.e.. Third, minor changes to the original data will significantly change the hash value [10].

3) Blake2

Blake2 is a cryptographic hash algorithm used to generate hash values from data. There are two versions of Blake2, namely Blake2b, and Blake2s. blake2b is a more extended and powerful version, whereas Blake2s is shorter and faster [11].

2.3 Obfuscator

Several processes were carried out to obtain the pseudo-random number. First, the seed value is determined, the initial value used to generate random numbers. In this case, the seed value is a string encrypted with the DES algorithm. The hash function will then generate a unique value for each seed value. The hash value obtained is stored as a new seed value to generate the first random number. The first random number generated is used as the seed value to obtain the second random number, and so on. The iteration will be carried out as often as the random numbers want to generate. Do as many lines as the file wanted to obfuscate in this case.
The source code files are randomized after obtaining random numbers with a certain number. Randomization was performed by swapping the positions of two lines of code from the original file according to random numbers. For example, if a random number [5, 2, 10, 7, 6, 3, …] is obtained, then the 5th row is exchanged with the 2nd row and stored temporarily. The exact

process is carried out using random number pairs next, i.e., the 10th row is swapped with the 7th row, and so on, until all random numbers are used. The randomization process was carried out in more detail, as shown in the flowchart in Figure 3.

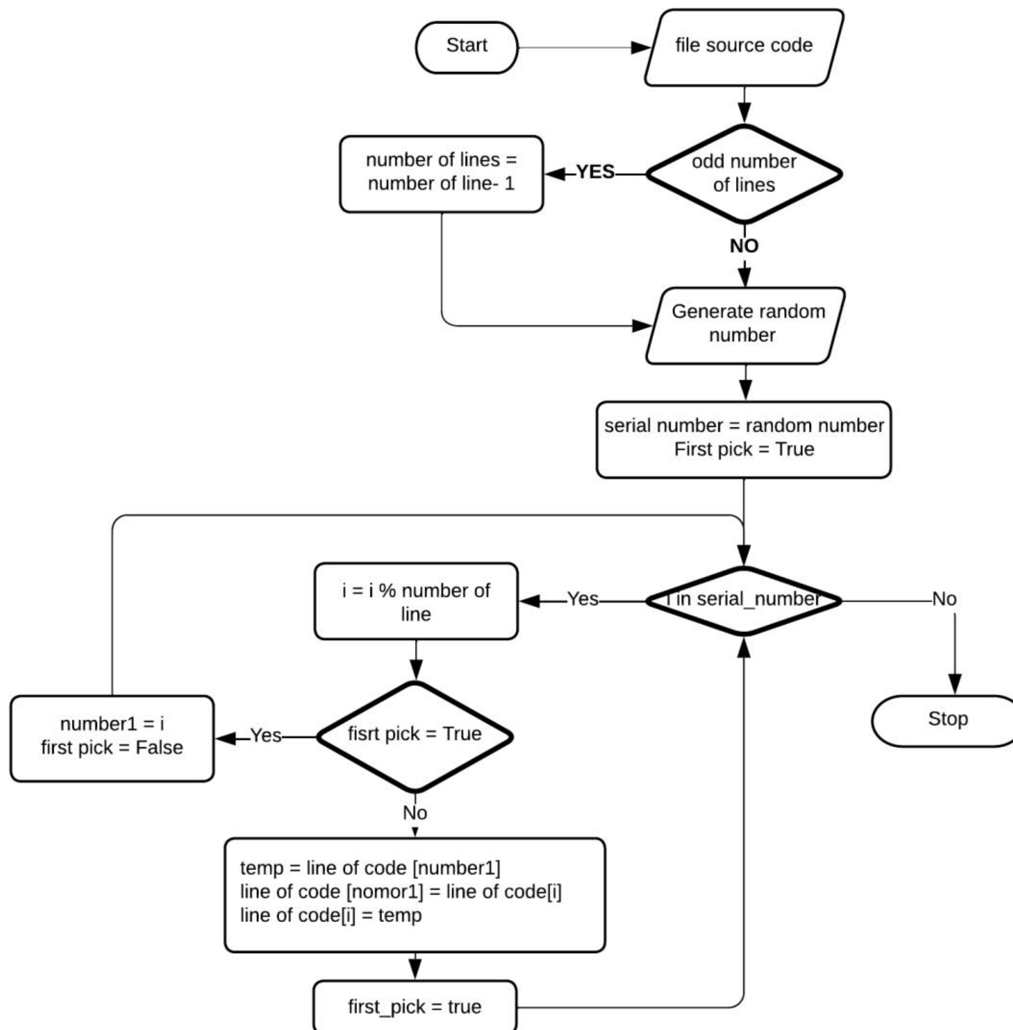

Figure 3 Randomization Flow

Based on Figure 3, initially, input is given as a source code file that wants to obfuscate and then checks the number of lines in the code file. If the number is odd, the number of rows stored is row-1 to generate a random number. The random number that has been generated is set as a serial number. The randomization algorithm involves the variables first_pick and number1 as row markers where the order will be swapped. By default, the value of the first_pick variable is false. Then repeat as many sequence numbers. In each iteration, the modulus is applied to the number of rows so that the random number in the serial number is within the range of the number of rows. Next, check the value of the first_pick variable. If it is true, then the current random number stored in the variable i is used as number1 then first_pick is set as false. If the value is false, the row position is exchanged between the number 1 and i index codes.

2.4 Deobfuscator

The deobfuscation process is carried out in the same way as the obfuscation process. The first time the pseudo-random number is obtained using the same hash function as the function used during obfuscation, but the random number obtained is inverted so that it can

return the lines of code to their original order. For example, if during the obfuscation process, a random number is obtained [5, 2, 10, …, 7, 6, 3], then the inverse random number obtained in the deobfuscation process becomes [3, 6, 7, …, 10, 2, 5].
The deobfuscation process will be combined with the compilation process using a batch program. In the batch program, instructions are stored to run the deobfuscator before compiling the source code so that the source code files can work normally.

3. RESULTS AND DISCUSSION

In this section, the result of this research is discussed. It starts from the pseudo-random number generator in each algorithm, the randomization algorithm for obfuscating, compilation, and obfuscator testing.

3.1 Pseudo-random Number Generator Result

One type of pseudo-random generator is a hash-based pseudo-random generator, where the output is generated using a hash function instead of a cryptographic algorithm. This sub-chapter presents the pseudo-random number results obtained from the three algorithms. The random number obtained is not entirely random but depends on the initial seed value. Examples of results obtained with the three hash algorithms used are listed in Table 1.

Table 1 Pseudo-random number Result

Number.	Algorithm	Seed	Pseudo-random Number
1	MD5	“seed”	[16, 32, 45, 4, 47, 56, 45, 34, 85, 68]
2		encrypted seed	[74, 20, 26, 70, 74, 61, 62, 18, 7, 15]
3	SHA256	“seed”	[25, 77, 33, 93, 40, 16, 57, 15, 86, 17]
4		encrypted seed	[3, 56, 3, 66, 50, 44, 47, 89, 13, 0]
5	Blake2	“seed”	[18, 65, 14, 93, 99, 81, 38, 54, 19, 62]
6		encrypted seed	[69, 6, 89, 1, 5, 41, 69, 6, 62, 22]

The results in Table 1 are the random number results obtained using the MD5, SHA256, and Blake2 hash algorithms with two types of seeds, namely strings that are directly set as "seed" and seeds that are encrypted using the DES algorithm. In this case, the degenerated random number consists of 10-digit numbers from 1 to 100. In contrast, in the obfuscation process, the number of random numbers is adjusted to the number of lines of source code that wanted to obfuscate.

3.2 Randomization Result

Randomization was carried out based on the random number obtained, the number of which corresponds to the number of lines of code to be obfuscated. If the number of rows is odd, the random number generated is the number of rows-1, so it still has pairs of numbers for the randomization process. Details of the randomization algorithm can be seen in Figure 3.
This section will present the randomization results obtained from the MD5 hash algorithm. For example, the MD5 algorithm generated random numbers [74, 20, 26, 70, 74, 61, 62, 18, 7, 15, 40, 36, 74, 45, 19, 62] to obfuscate 16 lines of code. Furthermore, obfuscation occurred by carrying out the randomization process, as shown in Table 2.

Table 2 MD5 Algorithm Randomization Process

Indeks	Iteration 1 i = 74 mod 16 = 10 number1 = 10 First pick = False	Iteration 2 i = 20 mod 16 = 4 swap (line[10], line [4]) First pick = True	Iteration 3 i = 26 mod 16 = 10 number1 = 10 First pick = False	Iteration 4 i = 70 mod 16 = 6 swap (line [10], line [6]) First pick = True	Iteration 5 i = 74 mod 16 = 10 number1 = 10 First pick = False	Iteration 6 i = 61 mod 16 = 13 swap (line [10], line [13]) First pick = True	Iteration 7 i = 62 mod 16 = 14 number1 = 14 First pick = False	Iteration 8 i = 18 mod 16 = 2 swap (line [14], line [2]) First pick = True
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	14
3	3	3	3	3	3	3	3	3
4	4	10	10	10	10	10	10	10
5	5	5	5	5	5	5	5	5
6	6	6	6	4	4	4	4	4
7	7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9	9
10	10	4	4	6	6	13	13	13
11	11	11	11	11	11	11	11	11
12	12	12	12	12	12	12	12	12
13	13	13	13	13	13	6	6	6
14	14	14	14	14	14	14	14	2
15	15	15	15	15	15	15	15	15

Indeks	Iteration 9 i = 7 mod 16 = 7 number1= 7 First pick = False	Iteration 10 i = 15 mod 16 = 15 swap (line[7], line [15]) First pick = True	Iteration 11 i = 40 mod 16 = 8 number1= 8 First pick = False	Iteration 12 i = 36 mod 16 = 4 swap (line [8], line [4]) First pick = True	Iteration 13 i = 74 mod 16 = 10 number1= 10 First pick = False	Iteration 14 i = 45 mod 16 = 13 swap (line [10], line [13]) First pick = True	Iteration 15 i = 19 mod 16 = 3 number1 = 3 First pick = False	Iteration 16 i = 62 mod 16 = 14 swap (line [3], line [14]) First pick = True
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
2	14	14	14	14	14	14	14	14
3	3	3	3	3	3	3	3	2
4	10	10	10	8	8	8	8	8
5	5	5	5	5	5	5	5	5
6	4	4	4	4	4	4	4	4
7	7	15	15	15	15	15	15	15
8	8	8	8	10	10	10	10	10
9	9	9	9	9	9	9	9	9
10	13	13	13	13	13	6	6	6
11	11	11	11	11	11	11	11	11
12	12	12	12	12	12	12	12	12
13	6	6	6	6	6	13	13	13
14	2	2	2	2	2	2	2	3
15	15	7	7	7	7	7	7	7

The randomization results obtained against the source code files are shown in Figure 4.

```
soal4.c
> Semester 4 > Prasidang > source code c > soal-latihan-dasar > materi-03 > C
1   #include<stdio.h>
2
3   int main(){
4       int n, i;
5       int jumlah = 0;
6
7       printf("masukkan nilai n : ");
8       scanf("%d", &n);
9       jumlah = 0;
10      for(i = 0; i <= 2*n; i++){
11          if(i % 2 != 0)
12              jumlah = jumlah + i;
13      }
14      printf("jumlah bilangan ganjil =%d", jumlah);
15      return 0;
16  }
```

```
obf_code_soal4.c 5
1. MD5 > C obf_code_soal4.c > scanf
1   #include<stdio.h>
2
3       return 0;
4   int main(){
5       jumlah = 0;
6
7       int jumlah = 0;
8   }
9   if(i % 2 != 0)
10      for(i = 0; i <= 2*n; i++){
11      printf("masukkan nilai n : ");
12              jumlah = jumlah + i;
13      }
14      printf("jumlah bilangan ganjil =%d", jumlah);
15      int n, i;
16      scanf("%d", &n);
```

Figure 4 MD5 obfuscation results

soal4.c is the original file that wanted to obfuscate while obf_code_question4.c results from obfuscation using the MD5 algorithm with random numbers [74, 20, 26, 70, 74, 61, 62, 18, 7, 15, 40, 36, 74, 45, 19, 62].

3.3 Obfuscator Testing Result

At the testing stage, an evaluation of the Dissimilarity and Potency of the obfuscator is carried out. Dissimilarity is measured using the Euclidean distance by conducting a series of obfuscation experiments on one source code file using a different seed ten times for each implemented hash algorithm. Then, the average value of the ten trials is taken as the dissimilarity value. Meanwhile, Potency was evaluated through the participation of 9 respondents who tried to deobfuscate manually.

3.3.1 Dissimilarity

Dissimilarity testing is done to see the level of difference between the original file and the obfuscated file. Testing is done using the Euclidean Distance.

$$d^2_{x,y} = \sum_{i=1}^{n}(x_i - y_i)^2 \quad (1)$$

Table 3 Dissimilarity Value

Number of Lines	Dissimilarity (%)		
	MD5	SHA256	BLAKE2
16	50	78	67
25	52	61	58
35	43	55	53
45	52	58	53
55	44	46	43
Average	**48,2**	**59,6**	**54,8**

In the tests conducted, the obfuscator using the MD5 algorithm resulted in a difference of 48.2% between the source code file and the obfuscated source code file. Meanwhile, the obfuscator using the SHA256 algorithm recorded a different rate of 59.6%, and the obfuscator using the Blake2 algorithm achieved a different rate of 54.8%. Based on these results, it can be concluded that the SHA256 hash algorithm produces the highest dissimilarity value, although, in general, there is no significant difference between the algorithms. The larger the file size to be obfuscated, the smaller the visible difference. This is influenced by the range of random numbers used, in this case, numbers between 1 and 100.
Until now, no research has established a standard level of dissimilarity that can be used as a reference for assessing the quality of a good obfuscator. However, when compared with the results of previous research in 2020, which tested source code obfuscation of code variables

using the Euclidean distance method with an average value of 12.7 on ±15 lines of code (Frank & Koay, 2020), it can be concluded that the obfuscation using the algorithm hash is capable of producing higher differences between the original file and the obfuscated file, namely 25 for the same number of lines. Therefore, it can be said that the analysis of obfuscated code will be more complex and require more time.

3.3.2 Potency

Potency testing is conducted to determine how confused the user (human) is in reading the obfuscation results. Each algorithm used was tested by three respondents using 5 C language source code files with different line lengths, namely 15 lines, 25 lines, 35 lines, 45 lines, and 55 lines. From each respondent, information was collected about the time required and the new file generated, then the percentage of the correctness of the corrected code was seen. The correctness rate is obtained by counting the number of lines of code that are successfully repaired.

Table 4 Average time needed

No.	Algorithm	15 line code	25 line code	35 line code	45 line code	55 line code
1	MD5	00:01:19	00:06:54	00:13:16	00:24:27	00:25:48
2	SHA256	00:03:13	00:07:03	00:20:54	00:25:45	00:29:42
3	Blake2	00:03:43	00:11:57	00:19:38	00:23:14	00:27:40

It can be seen in Table 4 that the average time needed to deobfuscate 15 lines of code manually is 1 minute 19 seconds, while the SHA256 and Blake2 obfuscators need more than 3 minutes. Even though the number of instructions in the 15 lines of code is relatively small, the time required to carry out the obfuscation process is still fairly long. From these results, the more lines of code that must be obfuscated, the more time required will increase.

Table 5 Average Percentage of Correctness

No.	Algorithm	15 line code	25 line code	35 line code	45 line code	55 line code
1	MD5	93,33	91,33	64,67	66,67	83,67
2	SHA256	100	89,33	69,33	63	80,33
3	Blake2	100	90,67	76,67	65	89

Based on Table 5, the percentage of correctness is generally influenced by the number of lines of code that wanted to be deobfuscated. The larger the file size, the more difficult it is to deobfuscate correctly by the user.

4. CONCLUSIONS

In this research, the obfuscation method has been developed, namely the obfuscation layout method, especially changing the code format. The obfuscators developed utilize hash functions in the obfuscation process, namely MD5, SHA256, and Blake2. The hash function generates a hash-based pseudo-random number modified with the DES encryption algorithm to disguise the seed used.

The results showed that using the hash algorithm for the obfuscation process effectively maintained security and complicated the analysis of a C language source code. In the tests conducted, it was seen that obfuscation using the hash algorithm could change the source code to be difficult to recognize, with an average level of difference 54.2%. Respondents took an average of 28 minutes to understand and analyze the 40-line C language source code, with an average percentage of truth of 64.7%.

The security of objects to be obfuscated depends on choosing the suitable method and algorithm. The results of this research can open opportunities for the development of other obfuscation techniques that can protect secret objects from hacker attacks or copyright theft.

REFERENCES

[1] S. Lipner, "Security and Source Code Access: Issues and Realities," in *IEEE Xplore*, 2014.

[2] A. Ivanovski and T. Stojanovski, "Software protection using obfuscation," 2010.

[3] C. Beheraa and L. Bhaskari, "Different Obfuscation Techniques for Code Protection," 2015.

[4] J. Schneider and T. Locher, "Obfuscation using Encryption," 2016.

[5] P. Faruki, H. Fereidooni, V. Laxmi, M. Conti and M. Gaur, "Android Code Protection via Obfuscation Techniques: Past, Present and Future Directions," 2016.

[6] L. Durfina and D. Kolar, "C Source Code Obfuscator," 2012.

[7] R. C. E. Frank and P. P. A. Koay, "Embedding Java Classes with code2vec: Improvements from," in *International Conference on Mining Software Repositories (MSR)*, Hamilton, New Zealand, 2020.

[8] X. Wang and H. Yu, "How to Break MD5 and Other Hash Functions," in *Advances in Cryptology – EUROCRYPT* , 2005.

[9] W. Stallings, Cryptography and Network Security, Pearson, 2017.

[10] A. W. Appel, "Verification of a Cryptographic Primitive: SHA-256," in *ACM Transactions on Programming Languages and Systems*, 2015.

[11] Nable, "Security - SHA-256 Algorithm Overview," 12 September 2019. [Online]. Available: https://www.n-able.com/blog/sha-256-encryption#:~:text=SHA%2D256%20is%20a%20patented,as%20long%20as%20when%20unencrypted..

[12] J.-P. Aumasson, S. Neves, Z. O'Hearn and C. Winnerlein, "BLAKE2: Simpler, Smaller, Fast as MD5," in *International Conference on Applied Cryptography and Network Security*, Canada, 2013.