Research Article

Odessa-512: Hybrid New Hash Function

Mr. Touraj Ostovari[a]

[a]Bachelor's degree in Computer Software Engineering from Tabriz Technical and Vocational University, Toraj.ostovari@gmail.com

Article History: Received: 14 July 2020; Accepted: 2 January 2021; Published online: 5 February 2021

Abstract: Nowadays, security is an important matter in all computer fields. The presence of computers of different sizes in our lives has multiplied the importance of this subject. For example, the security problem of MD5 [1] is not negligible, and it must be replaced with a new hash algorithm as soon as possible. A lower computational speed is one of the major issues of the hash algorithm of this study because we used multiple other hash algorithms to prevent collisions alongside the BEL [2] algorithm for creating random bits and the divide and conquer method for moving and dividing the bit blocks during the calculations. This hashing model completely solves the common collisions in MD5 and SHA1 [3], which we will discuss later on. This model is on par with Keccak-512 and Blake-512 in terms of the Avalanche effect. Also, the TMTO and Birthday attacks [4] were implemented on this model, and it successfully passed the no collision security test.

Keywords: Hash, Odessa Hash Function, One Way Function, Blake, Keccak.

Introduction

The one-way function model of this study uses the hash byte output addition as the initial value for producing random bytes using the BEL algorithm. The current algorithm is most useful for daily usages such as the MD5 because it solves all the collisions from MD5 and increases the safety of current hash methods. The current hash calculations use simple logical operations such as XOR and AND[1]. We took inspiration from the Merkle Damgard model for our compression, and use the divide and conquer method in a part of this paper [6][7]. We present the implementation parts of this article in a code-based manner without any mathematical proofs. The current one-way function model is simulated using C#, and you can access its source code. The collision security results of this paper are based on experimental and security results. We hope this article is useful for you and the future.

Implementation Method:

- **Initialization:**

- BitArray A = new BitArray(1024, true);

- BitArray B = new BitArray(1024, false);

- BitArray C = new BitArray(1024, false);

- BitArray D = new BitArray(1024, true);

- Long Sums;

- Long shift;

First, we define four 1024 array blocks. Blocks A and D have a Padding value of one (by default), while blocks B and C have a Padding value of zero.

Then, the input string is read in the UTF8 format and written in the blocks from A to D in reverse. We read and rewrite the blocks up to a maximum of 12 rounds if the user input text is larger than our 4096 block array. In the meanwhile, we place the UTF8 character byte addition result in the Sums variable. The shift value is a variable between 1 to 20 that reverts to 1 and goes through its incremental steps again whenever it reaches 20 during the block initialization (fetching the user input information).

```
{ // Scope Begins here
SHA512 sha512 = SHA512.Create();
byte[] temp = sha512.ComputeHash(‘User Input’);
```

[1] AND alone creates a one-way path [5].

```
for (int i = 0; i<temp.Length; i++)
        {
        Sums += (temp[i] * shift) + (long)Math.Pow(temp[i], 2);
        }
} //Scope Ends here
```

You can see in the semi-code above that we created a SHA512 function sample to perform the user input text calculations and add the output bytes to the previously used Sums variable.

The following semi-codes perform the same operation for the Sums variable, but their only difference is their simple mathematical equation.

```
{ // Scope Begins here
MD5 md5 = MD5.Create();
byte[] temp = md5.ComputeHash('User Input');
        for (int i = 0; i<temp.Length; i++)
        {
        Sums += temp[i];
        }
} //Scope Ends here
{ // Scope Begins here
SHA1 Sha1 = SHA1.Create();
byte[] temp = Sha1.ComputeHash('User Input');
        for (int i = 0; i<temp.Length; i += 2)
        {
        Sums += (temp[i] * shift) + (long)Math.Pow(temp[i], 3);
        }
} //Scope Ends here
{ // Scope Begins here
IHash hash = SHA3.CreateKeccak512();
HashResult r = hash.ComputeString('User Input');
byte[] temp = r.GetBytes();
        for (int i = 0; i<temp.Length; i += 2)
        {
        Sums += (temp[i] * shift) + (long)(temp[i] + shift);
        }
} //Scope Ends here
```

- **The Hash Function Calculations Stage**

We compress the blocks similar to the Markle Damgard compressor. This is done using the following semi-code:

```
for (int i = 1; i<= 24; i++)
```

```
{
Bit.lst.Length = 1024;
Bit.BEL((Sums * i), (Sums * shift * i), 1023); // Initializing BEL Algorithm and
generates random 1024 (1023) bit
A = A.And(Bit.lst).LeftShift(2); // A AND 'Random generated bits of BEL Algorithm'
Then 2 times logical left shift happens
A = B.Xor(A).RightShift(1); // 1 time Logical Right Shift
A = C.Xor(A);
A = D.Xor(A);BitArray _C_ = new BitArray(1024, false);
BitArray _D_ = new BitArray(1024, false);
_D_ = (BitArray)D.Clone();
D = (BitArray)A.Clone();
D = D.LeftShift(1); // 1 time Logical Left Shift
A = (BitArray)_D_.Clone();
A = A.RightShift(1); // 1 time Logical Right Shift
_C_ = (BitArray)C.Clone();
C = (BitArray)B.Clone();
C = (BitArray)C.LeftShift(9); // 9 times Logical Left Shift
B = (BitArray)_C_.Clone();
B = (BitArray)B.RightShift(9); // 9 times Logical Right Shift
}
```

As you can see, the block compression process goes on for 24 rounds, but the A and C blocks change their places with the D and B blocks in every round while the logical shift occurs towards left or right.

Then, we will compress the A, B, C, and D block values in a method similar to the Markle Damgard model using the XOR bit operator.

```
A = A.Xor(B).Xor(C).Xor(D);
```

- **Divide and Conquer Stage**

In this stage, the resulted block A is divided into two halves as follows:

```
BitArray A_ = new BitArray(512);
BitArray B_ = new BitArray(512);
```

In reality, 512 is our block length, and the halved block A values are inserted into the A_ and B_ halves.

- **Final Calculation**

Finally, we perform the simple BEL algorithm seeding formula for ten rounds (from one to ten) and use the XOR operator on A_ and B_ block random bit results. Afterward, we XOR the A_ and B_ blocks in regards to each other for ten rounds, similar to the Markle Damgard compressor.

```
for (int i = 1; i<= 10; i++)
```

```
        {
Bit.lst.Length = 512;
Bit.BEL((Sums * shift * i), (Sums + shift * (long)Math.Pow(i, 10) * (long)Math.Pow(shift,
2)), 511);
A_ = A_.Xor(B_);
A_ = A_.Xor(Bit.lst);
B_ = B_.Xor(Bit.lst);
        }
A_ = A_.Xor(B_);
```

Examination:

In this section, we review the security problem and hash model of this system.

- **TMTO & Birthday Attacks**

The compressor hash model (one-way function) has successfully overcome a chain of 2,000 TMTO attacks with more than 1 million hash outputs. These TMTO attacks include the Birthday Attack; therefore, our model is immune against this type of attack.

- **Collision in MD5 and SHA1**

We present an example of an MD5 collision showing the Odessa hash's superiority compared to the MD5 hash. For example, the MD5 hash has the same output for the following two images, but Odessa does not.

Figure 1.A simple image of an airplane

Figure 2. A simple image of a ship

Table 1. Table of Collision

Image Name	Compressor Result (One-Way Function)	Compressor Name
Airplane	38DBB54A58771C86F96B8F5683421B8C226F93C6650EBFA2C48D6EBA97AC2FEBF334D0C5B05C66B1EB9A5842DA9222F14F1EFB728D98FDBCDB065F5590169BBB	Odessa-512
Ship	371F97B4D4A5C6A57121AB5858599E2B692AB8FCAA59EBD207022E2D77F3E86FABEA887CB4F9C51216675F3EE677B238D5412E50FE8344323A36CA9D4AD6E28A	Odessa-512
Ship	253DD04E87492E4FC3471DE5E776BC3D	MD5
Airplane	253DD04E87492E4FC3471DE5E776BC3D	MD5

You can download these images alongside the source code from the appendix.

Now, we want to analyze two files presented by Google to show the collision of SHA1.

Table 2. Table of Collision (Google Project)

File Name	Compressor Result (One-Way Function)	Compressor Name
shattered-1.pdf	07C218D4C4581BCC38E1995E5DB1ED1517A51D73AC4574A2706924F91891DE153155B4549CC841E6155E78B9C5A33058E26C1FF5FB49F2E10C52289A010DB50C	Odessa-512
shattered-2.pdf	FB32CEAD1497A7A6089CFCE3E1F5B1A537B7E9E4004F4879C5FE0D6CE7E64944DF8CBF9FDC3125AAA5C27092A16F2C4EBC831F1C0F1CDB28DB3ACE6B95E29722	Odessa-512
shattered-1.pdf	38762CF7F55934B34D179AE6A4C80CADCCBB7F0A	SHA1
shattered-2.pdf	38762CF7F55934B34D179AE6A4C80CADCCBB7F0A	SHA1

We analyze the Avalanche effect in the following. The Avalanche value changes in the function output; therefore, we tried to use similar inputs (least possible change) to better expose the quality of the current model.

Table 3. Table of avalanche effect changes

#	Input String	Compressor Result (One-Way Function)	Compressor Name
116	123	66AD16BEE2DC257B3CFE91B7C6C06514D1D5814ED728E22739D481054655607D0A6E6A892458BF13D535318384C3A9315909771BCE601D86A32B3DD8605D3177	Odessa-512
	132	AEA89DF1150DAB9B1112BEE1C0DD5F1CC0A40DA869F3AB4E9416B11D933F101026B9FE2CEC9A825BDF054F247B0C36382A1A397B24823BA05FBF855D97B99077	Odessa-512
119	123	8ae5f90863e7984d9db61a67a38907f81de3c60a48f032d7ad5c10fef3f5a30705cae8bb76d80bf92d3da9a7f970507254f46f1bbe22db1d2d3ae8582c9625a5	Keccak-512
	132	b47f8ac1b3710912f9db1878a8eb8f370b4be589f619f6c7067ea39d08c0f69f20a82d58ff48ef19492a2876630f25c79e17fb766979421fa07dee8c26dfbe8c	Keccak-512
121	123	bebabdca7fd8b26a157a7aeb0ea9e860669c80e5168085aca321c0fa2dd25e43210c8e89784beffdad521347b3468037908d9eabaa458a49728c0769dace54ad	Black-512
	132	4411051884fb5a322d2af81fabb3544a861388572569e363885d99c2ab911e9f2cad790a522d76463b6459ac17ccaa95efb949cfff91d29c898955dbd6db7891	Black-512

Conclusion:

As you can see, the current compressor (hash) function is on par with other modern models in terms of the Avalanche effect while solving the problems from MD5 and SHA1. Therefore, it could be used as a replacement for the MD5 and SHA1 functions.

Appendix:

The appendix file contains all images, documents, and source code.

https://mega.nz/file/AKgi1bzY#Mv0yw3jCDCfhl_ZqaV6rXfO0YdIp9ByILiR8kwKiyGY.

References

1. Mendel F., Rechberger C., Schläffer M. (2009) MD5 Is Weaker Than Weak: Attacks on Concatenated Combiners. In: Matsui M. (eds) Advances in Cryptology – ASIACRYPT 2009. ASIACRYPT 2009. Lecture Notes in Computer Science, vol 5912. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-10366-7_9
2. K. M. M. Kumar and N. R. Sunitha, "Hybrid cryptographically secure pseudo-random bit generator," 2016 2nd International Conference on Contemporary Computing and Informatics (IC3I), 2016, pp. 296-301, doi: 10.1109/IC3I.2016.7917978.
3. http://shattered.io/
4. http://www.cs.haifa.ac.il/~orrd/HashFuncSeminar/Lecture3.pdf
5. Milad Jafari Barani &Favad Jalili, Cryptography Concepts, Second Print, Naghos Publication, Tehran, 2018.
6. EynollahJafarnezadGhomi, Data Structures in C, Eleventh Print, OlomRayaneh Publication, Tehran, 2013.
7. Mehrdad Tavana& Saeed Haratian, Algorithm Design Principles, Second Print, Parseh Publication, Tehran, 2010.
8. Behrouz A. Forouzan, Cryptography and Network Security, First Edition, McGraw-Hill, 2007.
9. Henk C. A. van Tilborg and Sushil Jajodia, "Encyclopedia of Cryptography and Security", Springer, 2011.
10. Bruce Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition, John Wiley & Sons, 1996.