A new collision resistant hash function based on optimum dimensionality reduction using Walsh-Hadamard transform

Barzan Mozafari and Mohammad Hasan Savoji
Electrical & Computer Engineering Department
Shahid Beheshti University
{mozafari, m-savoji}@sbu.ac.ir

Abstract

Hash functions play the most important role in various cryptologic applications, ranging from data integrity checking to digital signatures. Our goal is to introduce a new hash function using Walsh-Hadamard transform for achieving dimensionality reduction (compression) with a regular and one-way distribution. Merkle-Damgard (MD) transform is applied to this compression function in order to turn it into a hash function. Our algorithm has a flexible framework in which some parameters and steps could be changed according to different needs for more security or less computation time. Our emphasis is on its resistance against some variations of birthday attack. We evaluate collision resistant behavior of this algorithm for some configurations by calculating the balance factor. As we will see our balance factor is very close to SHA-1's. We present some experimental results to determine the balance factor of this algorithm for several output lengths. The experiment is done first, by converting the hash function to a similar one with a lower range size for an exact evaluation. Moreover, for larger range size, we use some bits of the output with a fraction of possible inputs, randomly chosen, to obtain another approximation of its balance factor. We also analyze avalanche effect of our proposed function which is another common measurement for cryptographic hash functions.

1. Introduction

1.1. Background on hash functions

Hash functions play an important role in several security topics. They could be used both for integrity checking and for message authentication. After transferring large amounts of data, it is important to verify whether it has been forged by an adversary. In insecure channels like Internet, we need a mechanism to be able to verify the integrity of a message and also to authenticate its original sender. These could be achieved by using cryptographic functions but many objections have been cited against encryption of the whole message by some algorithms like DES [1]. Applying cryptographic functions imposes a heavy computational load and makes communication much slower. So, other methods like HMAC are usually applied in which hash functions are used instead of encryption [2]. Briefly, for authentication and integrity verification, hash functions have some clear advantages over encryption functions: faster calculation, free library codes and no export restrictions.

A hash function h is defined as $h=H(M)$ where M is a bit string of arbitrary length but it is mapped onto a string h of fixed length, say n. So we have $H:\{0,1\}^* \rightarrow \{0,1\}^n$.

1.2. Basic Properties

A hash function usually should have some properties such as [3]:

1. Hash value is **easy to compute**: So its computation is fast and the cost of its hardware implementation is low.
2. **Preimage resistance or One-wayness**: for a given y it is computationally impossible to find a value x so that $h(x)=y$.
3. **2-nd preimage resistance or Weak collision resistance**: for a given x and $y=h(x)$ it is computationally impossible to find a value $x' \neq x$ so that $h(x')=h(x)$.
4. **Collision resistance or Strong collision resistance**: it is computationally impossible to find any two distinct values $x' \neq x$ so that $h(x')=h(x)$.

We always prefer to achieve the first property as much as possible. Preimage resistance is important in some authentication scenarios where we do not send plain messages with their hash values, so if adversary can reverse our hash function he will be able to find our original message. For more details see MAC [4]. 2-nd preimage is for preventing the adversary from changing the original message in a way that the hash value remains unchanged. Last property, strong collision resistance is usually important in variations of birthday attack, described below.

1.3. Birthday Attack

Given a hash function $h:D \rightarrow R$, for examining the birthday attack we choose q points $x_1,...,x_q$ from D and calculate $y_i=h(x_i)$ for $i=1,...,q$. The success of this attack is when there exists a pair $x_i \neq x_j$ for which $y_i=y_j$. There are several ways for choosing x_i points but usually they are chosen at random. In this attack, an upper bound to find a collision is $r^{1/2}$ trials, where r is the range size of h, namely $|R|$ [4]. But in real hash functions we expect that collision occurs much earlier than this theoretical worst case [5]. We will return to birthday attack in Section 4 of this paper.

1.4. Compression functions

A compression function g is defined as $g=G(M)$ where M is a bit string of a fixed length, say m, and g is a string of a smaller length, say n $(m>n)$. These functions are important because most of well-known hash functions consist of an underlying compression function and a method, like MD transform, to combine several outputs of this fixed-length input function to produce a final hash value for a variable-size input.

1.5. Fast Walsh Hadamard Transform (FWHT)

Since we use FWHT in our proposed hash function, we briefly mention its definition here. Walsh-Hadamard transform is one of the orthogonal transforms which, because of its simplicity and fast implementation, is commonly used in coding theory and signal processing [6]. There are many fast algorithms for computing this transform like those mentioned in [7]. But, here we use a bit-reversal in-place algorithm whose pseudo code can be found in [8]. We refer to this transform by FWHT which for any $k>0$ takes 2^K integers as coefficients of original signal and returns the same number of integers as transformed signal coefficients. For calculating reverse Walsh-Hadamard transform, we can apply the FWHT again on the output, and then divide the result by 2^K, the number of coefficients. For a recursive definition of this transform see [7].

1.6. Our goal

As long as no provably secure and efficient hash functions are available, dedicated hash functions will play an important role [9]. It means that there is no 'theory' to design such functions, but a few design criteria should be met to prevent some known attacks.

For designing such functions we should meet some design criteria like completeness, nonlinearity, balancedness and correlation immunity, and the propagation criterion of degree k. Roughly speaking, these criterions are as follows [9]:

- **Completeness**: the output should depend on all input bits. If a function is not complete, the output will show specific symmetries.
- **Nonlinearity** is a desirable property in cryptography, as linear systems are known to be very weak. This characteristic is not **robust**: namely a small modification to the function can cause its linearity change to a high degree.
- **Balancedness**: In cryptographic applications it is often very important that the numbers that each output vector occurs be as much as possible the same. In this case, the uncertainty over the value of function f or its entropy is maximal.
- **Correlation immunity**: Let f be a hash function of n bits. Then f is ***m*th order correlation immune**, *CI(m)* ($1 \leq m \leq n$), iff knowledge of any m input variables gives no additional information on the output.
- **Propagation criteria**: This criteria study the dynamic behavior of a hash function, i.e., what happens if the input to the function is modified. Let f be a function of n bits. Then f satisfies the **propagation criterion of degree *k***, *PC(k)* ($1 \leq k \leq n$), if each bits of *f(x)* changes with a probability of *1/2* whenever i ($1 \leq i \leq k$) bits of x are complemented. The **strict avalanche criterion** or **SAC** that has been introduced in [10] is equivalent to *PC(1)*. And perfect nonlinearity, will be equivalent to *PC(n)*.

We have assured that our function includes nonlinear operations and has completeness, good balance, and strict avalanche criterion or SAC, *PC(1)* and fast computation using FWHT.

We describe our FWHT-based hash function in Section 2. In Section 3, we mention our design strategy in this new hash function. We first review balance definition in more detail as an important measurement of regularity for hash functions in Section 4. Then, we present some experimental results about collisions and balance factor on some configurations of our algorithm. Finally, in Section 5, we conclude and outline the possible further research on FWHT-based hash functions.

2. Our prototype of a hash function based on FWHT

2.1. Compression function: *gwh*

First, we introduce our compression function named *gwh*. We then convert it to a hash function, named *hwh*, by using MD transform as seen in figure 1. In this figure, the following symbols are used:

- L is an integer parameter between *8* and *512*.
- RF is an integer parameter between *1* and *3*.
- D is the input block whose size is $4*L$.

The used buffers are listed below:

- H is an L-bit integer.
- IV is the initial vector in our compression function. It has a size equal to H, namely L bits.
- T is a $(1+\log_2 L)$-bit unsigned integer.
- R, $R1$ and $R2$ are $(\log_2 L)$-bit unsigned integers.
- Z and M are $4*L$-*bit* blocks.

gwh(*D*,*IV*) :

a. Let T = Number of transitions of D's binary representation from 0 to 1 in left to right scan. Since D is a $4*L$-bit string, T is not greater than $2*L$ and so it has $(1+\log_2 L)$-bit representation (clearly, in this case zero and $2*L$ transitions will result in the same T).

b. Let $R1$ = Rightmost $\log_2 L$ bits of D

c. Let $R2$= Leftmost $\log_2 L$ bits of D

d. Let $R = R1$ *XOR* $R2$

e. $Z = D$.

f. Calculate FWHT on Z as a sequence of 4 L-bit integers, to obtain another $4*L$ block. Since FWHT is in-place, generated result is also in Z.

g. Reduce the dimensionality of Z by zeroing its rightmost RF bits.

h. Reconstruct the original signal by calculating in-place reverse FWHT on Z as a sequence of 4 integers of L bits. For reverse FWHT, simply apply FWHT on Z, and then divide each of L-bit integers of its output by a factor of 4.

i. Let $M = Z - D$. Consider M as a sequence of 4 L-bit integers named $M(0)$, $M(1)$,$M(2)$ and $M(3)$ in right to left order.

j. $H = IV$;
For t = 0 to 3 do
$H = H + M(t)$;
$H = RSHL(H, R)$;
$R = RSHL(R, T)$;

k. Return H.

Figure 1: Algorithm of *gwh* compression function

gwh is used to calculate message digest for string blocks of size $4*L$ bits. The produced message digest's length is L bits. In this algorithm all integers have been considered unsigned. Moreover, we use $RSHL(A,B)$ to indicate rotational left shift of A by B bits, in which we could use the remainder of B to length of A (in bits), instead of B itself. The main idea of this paper is in steps *f* to *i*. We believe that step *j* could be easily replaced with some other operations that convert *4* blocks of L-bits into one block provided that they amplify the one-wayness of *gwh* and also do not decrease *gwh*'s regular distribution too much. As an example, the impact of using XOR instead of addition (ADD) in step *j* is shown in Section 4.

2.2. Hash function: *hwh*

Now we apply Merkle-Damgaard (MD) transform on *gwh* to convert it to a hash function named *hwh*. To generate the overall message digest, the input bit string S is padded to a multiple of $4*L$ by appending a bit of *1* and one or more necessary bits of *0*. The actual length of S before padding, in a $4*L$-bit representation, forms IV as first block $D(1)$. After this, S is divided into the $4*L$-bit blocks $D(2), D(3),..., D(n)$. Now according to MD transform, these blocks are processed in order. Refer to figure 2 and 3 to see the pseudo code and diagram of this transform on *gwh*. The first value of IV is zero. The final hash value is generated in a L-bit buffer named H with initial value of $H=0$.

```
hwh (S):
D(1) = length of S shown in 4*L bits.
Append necessary 0 bits to S (for padding) to make its length a
multiple of 4*L.
Divide S into blocks D(2),...,D(n)
H = 0;
For t = 1 to n do
        H = gwh(D(t), H);
Return H;
```

Figure 2: Pseudo code for *hwh* hash function

2.3. Suitable values for parameters

Two parameters in figure 1 are L and RF. L is the size of output in bits. The number of output bits depends on the needed security threshold for birthday attack or on other considerations like time performance. One advantage of this algorithm is the fact that the general behavior of the hash function remains approximately unchanged for different values of the L parameter. In Section 4 we will see that the balances of *gwh* for several values of L are almost the same. So we could choose a suitable value for L and then rely on this prediction that birthday attack will succeed only after about 2^{L-1} trials.

By RF we mean Reconstruction Factor. RF is the number of rightmost coefficients in the transformed signal which are zeroed before reversing the result of the transform. Experimental results show that for $L=8$, maximum balance or in other words minimum collisions, is achieved when we choose $RF=3$ out of *4*.

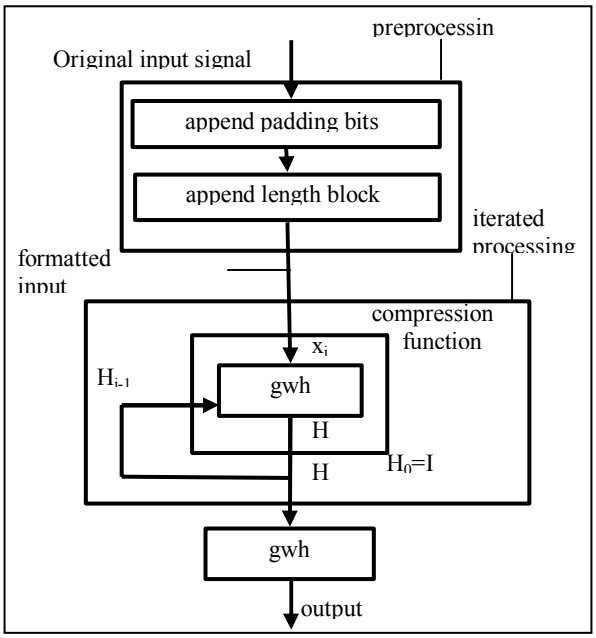

Figure 3: Diagram of applying MD transform on *gwh* to convert it to hash function *hwh*

3. Design Strategy

3.1. Motivation

Almost all existing cryptographic hash functions, like MD4, MD5 [8, 9] and SHA-family [11] use some steps which are rounded several times. The iterated steps usually include expanding, additions, rotational shifts and other boolean operations like AND, XOR, etc. In our proposal, we first use a Walsh-Hadamard transform and then apply a few steps on the difference between reconstructed and original signals. All orthogonal transforms result in a regular distribution and permit carrying out reconstruction with dimensionality reduction using a smaller number of operations instead of several high cost iterations. Here WHT is a good candidate for this purpose because its base functions are formed by *0* and *1* and so the transform is suitable for dealing with binary input sequences. Also there is a fast implementation of Walsh-Hadamard transform [8], which has a running time of $(\log_2 n)*(1+12*\log_2 n)$ in terms of basic computer operations, for a signal consisting of n values, each with machine word size. It is also clear that gwh, and hwh, have completeness property since all input bits are involved in the final result.

3.2. Dimensionality reduction with a regular distribution

As can be seen in figure 1, the core of the *gwh* is based on the reconstruction of the original input (as a signal) by applying reverse Walsh-Hadamard transform with a reduced dimensionality. According to experimental results (Section 4), we believe that subtracting reconstructed signal from the original input (*M* in figure 1) provides us a rather[1] regular distribution as well as random behavior which is usually a good design principle for hash functions[2].

3.3. Non-linear operations

Step *j*, is just for strengthening the one-wayness property of *gwh* by a series of nested rotational shifts of both coefficients and amounts of their shifts, according to a signal dependent parameter, namely *T*. This is because of the fact that values of *T* have a non-linear distribution on possible binary signals. Just a few number of signals have a very low or very high amount of *T* (like *T=0,1*) but, many other signals have an average value for *T*, like *T=2*L*.

Although this non-linear distribution of *T* decreases the reversibility of overall hash function but, also clearly has an undesirable impact on its regularity, as we will see more quantitatively in Section 4. Indeed, in hash function design there is always a trade off between regular distribution and irreversibility.

3.4. Keyed Parametric Hash by dimensionality reduction

We can easily change our *gwh* function to accept a secret key as well as the input sequence. Using this secret key, the *gwh* function decides which coefficients take part in the reconstruction process. In other words, our hash function can get two inputs: a message sequence and a secret key that is already shared between the sender and the receiver. Therefore, to forge this hash value, the attacker will have to guess the used secret key first in order to be able to attack the algorithm. This means a stronger keyed hash function that has its own benefits and applications.

4. Security measurements on *gwh*

In this Section, we analyze the security aspects of *gwh* by calculating two common security measurements on our proposed function: Balance Factor and Strict Avalanche Criterion (SAC) or PC(1). But first, we briefly introduce our simple tester software that we have used for calculating these parameters efficiently. Also we present the computable measures that we have calculated as the indicators of balancedness and avalanche effect.

[1] For real non-trivial hash functions, this distribution is not - or could not be - exactly regular. See [5].

[2] Randomness is usually a good policy, but not the best. Random functions are weaker than regular ones against birthday attack.[5]

4.1. Tester program

For calculating balance and SAC factors of *gwh* we have written an ad-hoc tester application in C language. This tester is able to generate a given number of a specified-length array of *L*-bit elements in 3 different modes:

1. Fills the array with successive values starting from a given number up to another given number.
2. Fills the array by random numbers for a specified number of iteration.
3. Fills the array in one of the above modes and then produces other numbers each by alternating one bit of the first numbers.

After generating above big numbers (implemented by arrays), the tester could do any of following operations:

1. Counts the number of occurrence of any portion of output bits.
2. Reports the first occurrence of a collision on any portion of output bits.
3. In the third mode of data production, calculates the average number of output bits which are changed after inverting one bit of input.

4.2. Balance factor for *hwh*

BALANCE MEASURE [5]: Consider the hash function $h:D \rightarrow R$ for which we have $|D|=d$ and $|R|=r$. Name the elements of range R as $R_1,...,R_r$ and define d_i as number of input points which are mapped to R_i.

Precisely, $d_i=|\{x \in D \text{ where } h(x)=R_i\}|$. The balance of h is defined as:

$$\mu(h) = \log_r \left[\frac{d^2}{d_1^2 + \ldots + d_r^2} \right]$$

We calculated balance of this algorithm for different cases, summarized in figure 4. First we chose $L=8$ and cover all input space of *gwh* to find an exact value of balance for this case. Distribution of hash values for all possible 2^{32} different inputs over *256* output points is presented in figure 5. Our tests show that using XOR instead of addition in step *j* of *gwh* algorithm has a bad impact on the balance of *gwh*, reducing it from *0.999994* to *0.835881*. For comparing distributions of original *gwh* and the above variation see figures 5, 6.

Parameters	Inputs	$\mu(gwh)$
$L=8$ and $RF=3$	All possible 2^{32} inputs	*0.99999410522*
$L=8$ and $RF=3$ (In this test we used XOR instead of addition in step *j*. of gwh)	All possible 2^{32} inputs	*0.83588135980*
$L=32$ and $RF=3$	2^{32} random inputs	*0.99999998830* (over first *8* bits of output)
$L=128$	2^{16} random inputs	*0.99999999916* (over first *8* bits of output)

Figure 4: Summarized results for balance of *gwh*

In the second attempt, we chose $L=32$ and calculated the hash values for 2^{32} random inputs with size *128* bits. In this test we just considered first *8* bits of hash output to find an approximation of overall balance. Here d_i is the frequency of appearing i as the rightmost *8* bits of 2^{32} hash values $(0 \le i \le 255)$, and $d=2^{32}$, $r=256$.

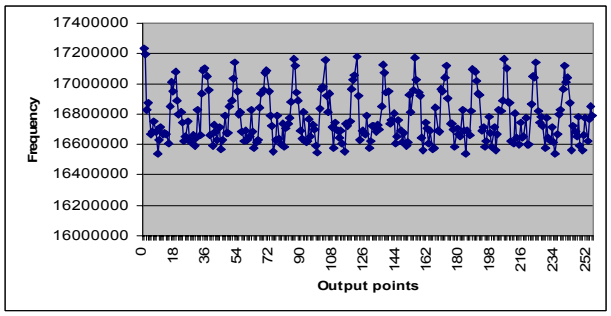

Figure 5: Frequency of appearing each possible *256* points in a whole covering of input space for $L=8$ in *gwh*.

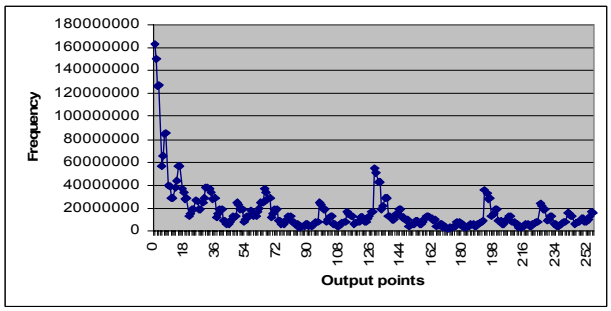

Figure 6: Impact of replacing additions of step *j*. of *gwh* with XORs on the frequency of appearing each possible *256* points in a whole covering of input space for $L=8$.

As the last case, we again limited our balance estimation only on *8* bits of output for 2^{16} randomly chosen inputs. The considered *8* bits of output in this case were in the positions of *0*, *17*, *50*, *67*, *85*, *100*, *118* and *127*. Here $d=2^{16}$, $r=256$.

Although it is shown in [5] that MD transform doesn't preserve the balance of the underlying compression function (here *gwh*), we do not calculate the balance factor of the final hash function (*hwh*) because it is usually expected that the balance of the final hash be not much less than that of its compression function. By comparing these to SHA-1's results we see that our balance is very close[3] to its.

4.3. Avalanche effect of *gwh*

[3] According to [5], the balance factor of SHA-1 for 32-bit inputs on several sub ranges of 8-bit outputs ranges from 0.99999998769 to 0.99999999083.

AVALANCHE EFFECT: For achieving more one-wayness, there is a mathematical abstraction, called avalanche effect, which says that an average of one half of output bits should change when a single input's bit is changed. More changes in output's bits imply higher degree of non-linear dependency between input and output bits. This is one of the desirable properties of a cryptographic hash function because it shows that hash values of some neighbor inputs are diffused all over the output space [12].

Our calculations for amount of avalanche effect in *gwh*, for *L=8* and *L=32* show that, on average, *3.849713* and *15.759494* changes occur respectively in output bits whenever a single bit of input is changed. In both cases, the figure is very close to one half of *L*.

5. Conclusion and further works

As we mentioned in Section 4, some important measurements, such as balance factor and avalanche effect, show that *gwh*, and probably *hwh*, have a desirable behavior as a cryptographic hash function. For example, a balance factor of *0.99999998830* for *8* bits of *32*-bit output size which is very close to – and sometimes better than - the same measurement for SHA-1 (US standard Secure Hash Algorithm, the most common universal hash function). It shows strongly that *gwh* has a very regular distribution, and therefore, has a good resistance against birthday attack. We just need to choose a suitable value for output size, *L*, and expect that our hash function can resist up to 2^{L-1} steps of birthday attack. It seems that for currently computational power of super computers, a value between *160* and *256* suffices for *L* as output size.

Also experimental data shows that approximately half of the output bits are altered when a single bit of input is inverted. This good avalanche effect, PC(1), can guarantee the one-wayness property of *gwh* and also *hwh*, which in turn causes better resistance of our hash function against more hash attacks like compression function attack and chaining attack that we did not consider in this paper. Our further research will focus on improvement of *hwh* as well as *gwh*, with more attention to other hash function attacks like differential attacks.

Moreover, we are interested in optimizing our hash function implementation to make it less costly in computation time.

10. References

[1] R.C. Merkle, "One Way Hash Functions and DES", *Advances in Cryptology Proceedings - CRYPTO '89*, Vol. 435, 1990, pp. 428.

[2] M. Bellare, R. Canettiy and H. Krawczykz, "Message Authentication using Hash Functions The HMAC Construction", *RSA Laboratories' CryptoBytes*, Vol. 2, No. 1, Spring 1996.

[3] A. Menezes, P. Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

[4] W. Stallings, *Cryptography and network security: Principles and practices*, Prentice Hall, 2003.

[5] M. Bellare and T. Kohno, "Hash Function Balance and its Impact on Birthday Attacks", *Advances in Cryptology EUROCRYPT*, May 2-6, 2004.

[6] K.G. Beauchamp, *Applications of Walsh and related functions*, Academic Press, 1984.

[7] J. Johnson and M. Puschel, "In Search of the Optimal Walsh-Hadamard Transform", *Proc. ICASSP*, 2000.

[8] T.H. Tossavainen, "Fast in-place Walsh-Hadamard Transform", *Music-DSP Source Code Archive*, *World Wide Web,* www.musicdsp.org .

[9] Bart Preneel, "Analysis and Design of Cryptographic Hash Functions", *Ph.D. thesis of Katholieke Universiteit Leuven*, Jan 1993.

[10] A.F. Webster and S.E. Tavares, "On the design of S-boxes", *Advances in Cryptology, Proc. Crypto'85*, LNCS 218, H.C. Williams, Ed., Springer-Verlag, 1985, pp. 523–534.

[11] NIST/NSA, "Secure Hash Standard (SHS)", *FIPS 180-2*, August 2002 (change notice: February 2004).

[12] H. Feistel, *Cryptography and Computer Privacy*, Scientific American, 1973, 228(5),pp 15-23.

[13] R.L. Rivest, "The MD4 Message Digest Algorithm", *Advances in Cryptology,CRYPTO'90*, LNCS 537, Springer-Verlag, pp. 303–311, 1991.

[14] R. Forr´e, "Methods and instruments for designing S-boxes", *Journal of Cryptology*, Vol. 2, No. 3, 1990, pp. 115–130.

[15] K.G. Beauchamp, *Walsh Functions and Their Applications*, Academic Press, New York, 1975.

[16] D.Hong, J.Sung, S.Lee, D.Moon and S.Chee, "A New Dedicated 256-bit Hash Function: FORK-256", *Cryptographic Hash Workshop*, NIST , October 2005.