OWF Candidates Based on: Xors, Error Detection Codes, Permutations, Polynomials, Interaction and Nesting

Paweł Cyprys[1], Shlomi Dolev[1], and Oded Margalit[1]

Ben-Gurion University of the Negev
July 3, 2023

Abstract. Our research focuses on achieving perfect provable encryption by drawing inspiration from the principles of a one-time pad. We explore the potential of leveraging the unique properties of the one-time pad to design effective one-way functions. Our methodology involves the application of the exclusive-or (xor) operation to two randomly chosen strings. To address concerns related to preimage mappings, we incorporate error detection codes. Additionally, we utilize permutations to overcome linearity issues in the computation process.
In order to enhance the security of our approach, we propose the integration of a secret-sharing scheme based on a linear polynomial. This helps mitigate collisions and adds an additional layer of perfect security. We thoroughly investigate the interactions between different aspects of one-way functions to strengthen the reliability of commitments. Lastly, we explore the possibility of nesting one-way functions as a countermeasure against potential backdoors.
Through our study, we aim to contribute to the advancement of secure encryption techniques by leveraging the inherent strengths of the one-time pad and carefully considering the interplay of various components in the design of one-way functions.

1 Introduction

We propose the exploration of computationally efficient one-way functions that can serve as an alternative to Secure Hash Algorithms (SHA) [14]. These functions should be resistant to preimage and collision attacks, providing enhanced security for commitments and signatures, such as Lamport's signature [17]. Relying solely on block-cipher-based functions like SHA may lead cryptanalysts to focus their efforts on breaking these functions, as demonstrated by the vulnerabilities found in MD4 [18] and MD5 [19], not to mention the potential existence of backdoors [10]. In this paper, we propose simple constructions using provable cryptographic primitives like one-time pads and secret sharing.

Our objective is to explore a range of computationally efficient one-way functions that can expand the choices available to implementers. Commitments based on one-way functions (assuming the provability of such functions implies $P \neq NP$) find applications in various scenarios, including Zero Knowledge Proofs

(ZKP) [1; 13], where one-way functions are used as commitment primitives. While cryptographic hash functions like the SHA family are designed to handle long inputs (e.g., files) and are expected to have collisions due to the pigeonhole principle, our focus in this paper is on inputs of the same length as, or smaller than, the output. This enables us to examine the inherent collision properties of the proposed functions, as discussed in [12] regarding length-preserving one-way functions.

We explore techniques to enhance the one-way properties of existing one-way function candidates by utilizing xor operations in the style of a one-time pad. We employ xor operations among essential components of instances of the original one-way functions [6; 5; 3]. Our objective is to mimic xor with a one-time pad in a way that ensures the success criteria of an instance (e.g., the sum in a subset-sum instance) while limiting the possible number of preimages. In commitment schemes, it is undesirable to have multiple fitting preimages or collisions, as the committer could select a preimage from the colliding set when revealing the commitment. Additionally, if there are numerous colliding preimages, the task of reversing the output of the one-way function candidate may become relatively easy. The ease of reversing the output obtained by bitwise xoring two random sequences serves as evidence of both the existence of a large number of possible preimages and the ease of finding one among them.

Overview. We aim to achieve provable encryption by utilizing the principles of a one-time pad. In this study, we explore the potential of leveraging the inherent properties of the one-time pad to design one-way functions. We begin by randomly selecting two strings, s_1 and s_2, each consisting of n bits. It is important to note that the result of performing a bitwise xor operation on s_1 and s_2, denoted as r_{12}, encompasses all possible combinations of s_1 and compatible counterparts of s_2 that yield r_{12}. As a result, reversing the process and obtaining r_{12} is relatively easy and leads to a multitude of possible answers (collisions), which grows exponentially with the lengths of s_i ($n = |s_i|$).

To address the issue of an excessive number of collisions and enhance the difficulty of reversing the function, we propose the utilization of error detection codes such as Cyclic Redundancy Check (CRC), Hamming codes, Reed-Solomon codes, and binary Goppa codes. For each s_i, we introduce an error detection code edc_i. The computation of r_{12} is then performed as follows: $(s_1 \circ eds_1) \oplus (edc_2 \circ s_2)$, where $\circ$ denotes concatenation. To facilitate discussion, we set the length of the error detection codes equal to the length of the original strings they represent, i.e., $|edc_i| = |s_i|$.

The design endeavours to utilize s_1 (and s_2) as a one-time pad for edc_2 (and edc_1, respectively). However, we demonstrate that in cases where the error detection code is linear, there exist polynomial time algorithms that can invert r_{12} and recover s_1 and s_2 with relative ease. To cope with the (reversible) linearity of error detection codes, we suggest using permutations, permuting edc_i by the values of s_j. For ease of discussion, we suggest using $2\lg(n)+1$ pairs s_1^i, s_2^i, namely, s_1^1, s_2^1, s_1^2, s_2^2, s_1^3, s_2^3, $\cdots$, $s_1^{2\lg(n)+1}$, $s_2^{2\lg(n)+1}$ compute for each of s_1^i, s_2^i the value $r_i = (s_1^i \circ s_2^i) \oplus \pi_{s_1^1, s_2^1, \cdots s_1^{i-1}, s_2^{i-1}, s_1^{i+1}, s_2^{i+1}, \cdots s_1^{2\lg(n)+1}, s_2^{2\lg(n)+1}}(edc_1^i \circ edc_2^i)$.

Note that every bit in the permuted edc_i is a function of all random bits; thus, the output is *holographic* in a sense.

To prevent collisions where more than one preimage exists for the same function output, we suggest using secret-sharing schemes. Secret sharing is another (beyond one-time-pad) very useful, proven perfect information theoretical secure primitive. In this scheme, the value r_i may have multiple pre-images, but still, we manage to restrict collisions. The commitment value is determined by the intersection of a line (or polynomial) with the y-axis, denoted as $s_1^i \circ s_2^i$. An interactive commitment approach is proposed to enhance security, where the committing party receives random x-values. This process ensures that even if only two values on the line do not collide, the commitment is still unique and can be attributed to a single possible value.

We further suggest a nesting of one-way functions in which the committing party is instructed first to use a certain one-way function to eliminate planned collisions; then, the committing party uses the output of the given one-way function as an input for her/his choice of a one-way function to eliminate possible backdoors in the first determined one-way function. Both parties should agree (and possibly verify by say, probabilistic sampling) that the suggested functions imply a small number of collisions.

To make the presentation self-contained and concise, we present only explicitly used definitions in our analysis. For a more comprehensive background on one-way functions and related applications, see, e.g., [16; 15; 6; 5; 3].

Paper roadmap. Section 2 develops the reasoning for the need for permutation beyond xors (in the style of mutual one-time pads). Section 3 introduces the use of (linear) polynomials to cope with possible collisions. Section 4 uses the interaction between the committer and the verifier to cope with potential planned collision (by the committer) and planned backdoors (by the verifier). Finally, concluding remarks appear in Section 5.

Throughout the paper, we illustrate the proposed concepts using toy examples. For the convenience of readers, the implementations required to replicate these examples can be found in [4]. The software was implemented using the SageMath computational software environment [21].

2 XORS, Error Detection Codes, and Permutations

We illustrate that without the use of permutation, the application of xor, which aims to emulate Shannon's "one-time pad" concept (as described in [20]), by mutually masking the error detections of s_1 and s_2, can be easily inverted in polynomial time.

To demonstrate the potential ease of inversion, we consider the following specific example: s_1 and s_2 are randomly chosen four-bit strings. We utilize a standard CRC-4-ITU algorithm (with the polynomial representation $x^4 + x + 1$ as defined in [11]) to compute crc_1 for s_1 and crc_2 for s_2, where each crc value consists of four bits. Subsequently, we compute $f(s_1, s_2) = (s_1 \circ crc_1) \oplus (crc_2 \circ s_2)$.

Due to the linearity of the process, we observe that $f(s_1, s_2) = (s_1 \circ s_2) \cdot A$, where A is a square matrix that implements the function f.

Evidently, there exists a one-to-one mapping between the values of s_1 and s_2 and $f(s_1 \circ s_2)$, and it is feasible to construct a matrix that computes f.

Furthermore, f is invertible, as f^{-1} can retrieve $s_1 \circ s_2$ from the output of $f(s_1 \circ s_2)$. The inverse function f^{-1} can be computed in polynomial time by utilizing the inverse matrix A^{-1}, such that $s_1 \circ s_2 = f(s_1, s_2) \cdot A^{-1}$.

Appendix A presents a specific numerical example illustrating the straightforward nature of such inversion.

Incorporating permutations. To enhance the reconstruction of critical parts, such as s_1 and s_2, we can extend the self-masking technique with CRC codes by incorporating permutation using permutation indices. This approach allows us to define an actual permutation.

In our example, we have presented a construction where a binary array is generated based on a string $s = s_1 \circ s_2$, with a binary length of $|s| = 8$ (suitable for representing a single character in ASCII Encoding). The first three bits of the binary array are utilized to determine the parameter p, while the next three bits determine the parameter q. These parameters, p and q, are then used to define a permutation denoted as π. The permutation function π is employed to map the elements of the binary string to a new instance of the binary string with permuted elements. Specifically, the elements (bit values) of the computed $crc_1 \circ crc_2$ are swapped based on the indexes defined by p and q using this permutation mapping. Namely, given two strings $s_1 = a_1, a_2, a_3, a_4,\ s_2 = b_1, b_2, b_3, b_4$, compute $crc_1 = c_1, c_2, c_3, c_4$ for s_1, and compute $crc_2 = d_1, d_2, d_3, d_4$ for s_2. Then, consider the sequence $crc = crc_2 \circ crc_1 = d_1, d_2, d_3, d_4, c_1, c_2, c_3, c_4 = e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8$, permute elements in a way that swapes e_{p+1} with e_{q+1}. The Blackbox then xors $s = a_1, a_2, a_3, a_4, b_1, b_2, b_3, b_4$ with the permuted crc.

Permutations example. Let us consider the following example that illustrates the ineffectiveness of polynomial-time inversion when permutation is applied in the BlackBox.

For instance, let us define $s_1 = 1001$ and $s_2 = 1010$. The resulting BlackBox output for s_1 and s_2 is denoted as r:

$$s_1 \circ s_2 = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix} \ r = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} (s_1^1 \circ s_2^1) \oplus \pi_{s^1}[crc_2^1 \circ crc_1^1] \\ (s_1^2 \circ s_2^2) \oplus \pi_{s^2}[crc_2^2 \circ crc_1^2] \\ \vdots \\ (s_1^8 \circ s_2^8) \oplus \pi_{s^8}[crc_2^8 \circ crc_1^8] \end{bmatrix}$$

$$B = \begin{bmatrix} 0\,0\,1\,0\,0\,0\,0\,0 \\ 0\,0\,0\,1\,0\,0\,0\,0 \\ 1\,1\,0\,0\,0\,0\,0\,0 \\ 0\,1\,1\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,1\,0 \\ 0\,0\,0\,0\,1\,1\,0\,1 \\ 0\,0\,0\,0\,0\,1\,0\,1 \\ 0\,0\,0\,0\,1\,0\,1\,0 \end{bmatrix}$$

We construct the matrix B to align with the new Blackbox, which incorporates a limited permutation and enables the computation of any row of the identity matrix I. This capability directly stems from the construction of matrices B (and A).

However, attempting to employ the same linear technique used in Appendix A by substituting matrix A with B proves unsuccessful due to the nonlinearity of the permutation.

In particular, the result of the Blackbox for the input $s_1 \circ s_2 = [1\ 0\ 0\ 1\ 1\ 0\ 1\ 0]$ is $r = \left[0\ 0\ 1\ 0\ 1\ 1\ 1\ 0\right]$ while the multiplication of $s_1 \circ s_2 = [1\ 0\ 0\ 1\ 1\ 0\ 1\ 0]$ by B yields a different vector of bits. When we perform xor operation over the rows $0, 3, 4$ and 6 of B we get the vector: $[0\ 1\ 0\ 0\ 0\ 1\ 1\ 1] \neq r = [0\ 0\ 1\ 0\ 1\ 1\ 1\ 0]$.

Based on the aforementioned example, it becomes evident that constructing matrix B according to the permuted Blackbox does not enable the utilization of B (as with A) to compute all the results of the new permuting Blackbox.

Although the limited permutation may reduce the number of combinations to be examined when attempting to reverse the function, we now aim to enhance the permutation operation to achieve a complete random permutation based on bits from other instances.

Better (full) permutation, the *holographic* approach. Although the minimal non-linearity in the solution presented in Appendix A is insufficient, we can improve upon it. The limited number of bits used for permutation (six out of the total eight in our example) restricts the available permutations to a very limited set. Therefore, it is preferable to have an (almost) uniformly chosen permutation. To achieve this, we introduce $2 \lg n + 1$ instances (specified as an input in Algorithm 1) of the s_1, s_2 scheme described earlier. In the second line of the pseudocode, we generate ℓ instances accordingly.

The index of a permutation is determined by $\lg((2n)!)$ bits, where $|s_i| = n$. By utilizing the random bits from all instances s_1^j, s_2^j where $j \neq i$, we obtain a total of at least $2n \lg n$ bits. This quantity is sufficient to fully define a permutation of the bits in $crc_2^i \circ crc_1^i$ (as demonstrated in [8]). We may choose to define the permutation index by starting with the most significant bit (MSB) of s_1^{i+1}, followed by the MSB of s_1^{i+2} up to the MSB of s_1^{i-1}, and then listing the bits that are second to the MSB of these s_1's, continuing until we reach the least significant bit (LSB) layer of s_2's. We refer to this sequence as MSBLSB_i. This operation is performed within the **for each** loop defined in line three. In each iteration of the loop, we execute the operations defined earlier (lines 4 and 5), resulting in the production of a puzzle with the corresponding index for that iteration.

It is important to note that every bit in the permuted $crc_2^i \circ crc_1^i$ is a function of all the random bits (s_i^j), making it "holographic" in nature.

Algorithm 1: Full permutation holographic hash

Input: $k = 2\lg n + 1$ = number of instances
Function `Generate_Puzzles()`:
  ℓ = Generate_Instances(k)
  for each instance i **in** ℓ:
    permuted$_i$=permute($crc_2^i \circ crc_1^i$, MSBLSB$_i$)
    puzzle$_i$= $(s_1^j \circ s_2^j) \oplus \mathit{permuted}_i$
    output puzzle$_i$

Note that one can replace s_1^i and s_2^i with a single longer s^i; the use of two parts here is for the sake of the gradual exposition for building our holographic one-way function candidate.

3 Polynomials for Collision Prevention

In this section, we propose a novel approach to address collisions that can be applied to other cryptographic hash functions, such as SHA-128. We utilize the concept of secret sharing, where the actual committed value is encoded using a polynomial, specifically a line.

Our new hash function employs a primitive cryptographic hash function to hash the y coordinates of points on a line that intersects the y axis at the committed value. The x values can range from 1 to m, where m is a chosen parameter. Alternatively, the verifying party can define the x values, creating a somewhat interactive commitment process.

We hypothesize that the committing party will be unable to coordinate two lines from the collisions of these m hashed values in a way that encodes a different line and a distinct secret. In cases where the number of collisions for each value is limited, as in our suggested one-way function, we can demonstrate that the number of possible collisions diminishes towards zero.

Let's assume a sufficiently large finite field F and $k + 2$ distinct numbers in F, denoted as $x_1, \ldots, x_{k+2}$ (with the possibility of x_i being equal to i). The pseudocode for the technique to prevent polynomial collisions is presented in Algorithm 2. As input to the algorithm, we generate the commitment (line 1) and employ it as part of secret sharing, where the constant term a_0 of the polynomial $P(x) = a_1 x + a_0$ defined over the finite field F represents the committed value (in our example, the result of f using s_1 and s_2). These values are declared as inputs to Algorithm 2. Next, we generate a random value from F to encode the polynomial coefficient a_1 (line 3). Finally, we employ Lagrange Interpolation to construct the polynomial $f(x)$ (line 4).

Algorithm 2: Polynomial generation

Input: $n = k + 2$ *distinct numbers*, $F = GaloisField(2^\ell)$, commitment
Result: $f(x)$ *polynomial*

1 **Function** `Generate_Input():`
2 $\quad a_0 = commitment$
3 $\quad a_1 = Generate\_Random\_Point(n, F)$
4 $\quad f(x) = Lagrange\_Interpolation(a_0, a_1)$

Continuing from the previous section, let's consider a Finite Field with an order of 2^ℓ, where each element in the field consists of precisely ℓ bits. As mentioned before, let t be the input for Algorithm 3. The subsequent step involves evaluating the polynomial $P(x)$ for input values $x = x_1, x_2, \ldots, x_{k+2}$ to generate a vector $t_1, t_2, \ldots, t_{k+2}$ (refer to lines 2 to 4 in Algorithm 3).

Subsequently, we can utilize the binary representation of t_i (line 5) to encode two strings s_1^i and s_2^i (line 6). Specifically, we take the first half of $\ell/2$ bits from t_i to form s_1^i, and in the subsequent line, we take the other half of $\ell/2$ bits to form s_2^i. Following this, we can calculate the permuted hash for each string (line 8).

It's important to note that the resulting value of $f(s_1^i, s_2^i)$ may not be unique, as there may exist s_1' and s_2' for which $f(s_1', s_2') = f(s_1^i, s_2^i)$, indicating the presence of collisions.

Algorithm 3: Calculation of permuted hash

Input: $f(x) = polynomial$, $F = GaloisField(2^\ell)$, *commitment*

1 **Function** `Generate_Values():`
2 $\quad t_{x_1} = f(x_1)$
3 $\quad \vdots$
4 $\quad t_{x_{k+2}} = f(x_{k+2})$
5 $\quad t_{xi_{binary}} = binary\_cast(t_{x_i})$
6 $\quad s_1 = t_{xi_{binary}}[0{:}\ell/2]$
7 $\quad s_2 = t_{xi_{binary}}[\ell/2{:}\ell]$
8 $\quad h = calculate\_permuted\_hash(s_1, s_2)$

To enhance the resistance against inversion attacks, we can empower the verifying party in the commitment process by allowing them to choose multiple values of x. They can request the corresponding committed values (y) before revealing the next challenge value of x.

In the following figures, we demonstrate the situation where the committer has the option to expose one of several lines (and corresponding commitments) when only the values for $x = 1$ and $x = 2$ are requested (indicated by the green color). Let's define $P(x) = a_1 x + a_0$, where a_0 represents the committed value (preimage of the one-way function), and a_1 is a randomly chosen value from the field. In Figure 1, the value $f(s_1, s_2)$ (or $f(s_3, s_4)$) is depicted as a blue horizontal line, where $s_1 \circ s_2$ corresponds to $a_0 + a_1$, and $s_3 \circ s_4$ corresponds to $a_0 + 2a_1$.

Figure 2 illustrates the green line representing $P(x)$, which shows the y coordinate for $x = 1$ (or $x = 2$) based on $s_1 \circ s_2$ (or $s_3 \circ s_4$) values. Interestingly, the blue horizontal lines in Figure 1 reveal a collision for $s_3 \circ s_4$, indicating the existence of $s'_3 \circ s'_4$ for which $f(s_3, s_4) = f(s'_3, s'_4)$. This collision implies that the committer can expose an additional committed value, which differs from the value represented by the green line.

When the committer needs to reveal $f(s_5, s_6)$, where $(s_5 \circ s_6)$ corresponds to the value of $P(3)$, the collisions indicated by the new blue line in Figure 3 do not align with the red line represented by $s_1 \circ s_2$ and s'_3, s'_4 (see Figure 4). Consequently, the committer is compelled to reveal the points on the green line, effectively exposing the original committed value.

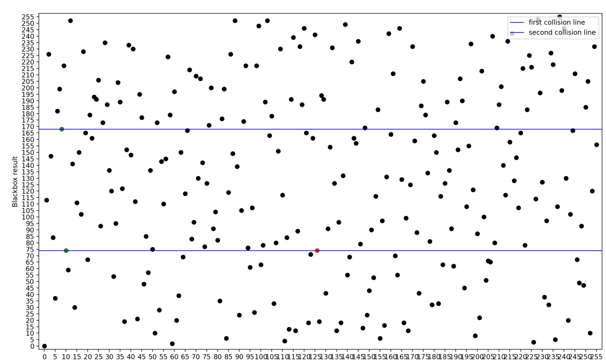

Fig. 1. Black-box values distribution for all possible s_1 and s_2 along with collision lines for $r_{1,2}$ and $r_{3,4}$

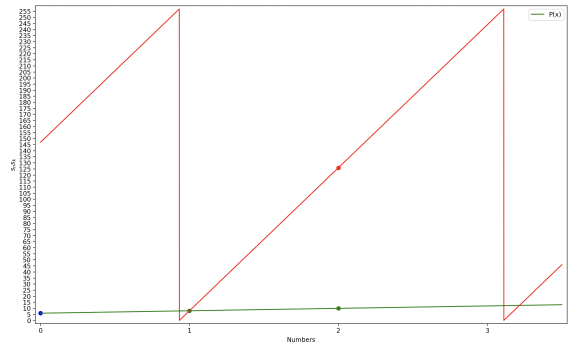

Fig. 2. Collision line for $P(2)$ points

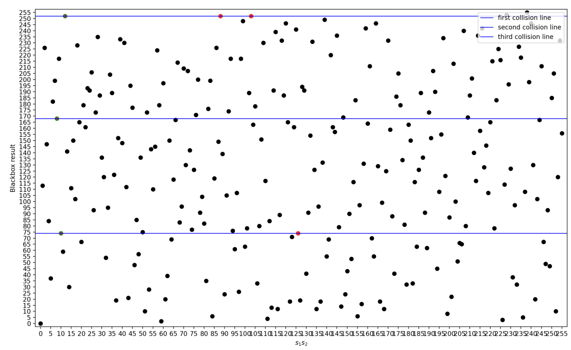

Fig. 3. Black-box values distribution for all possible s_1 and s_2 along with collision lines for $r_{1,2}$ and $r_{3,4}$

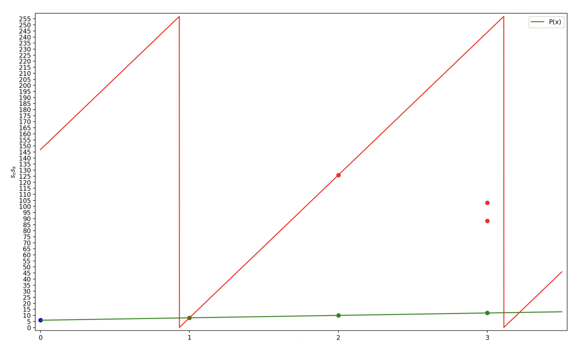

Fig. 4. Collision lines for $P(3)$ points

Note that such an approach can be relevant to other cryptographic hash functions where the input is padded by a random nonce chosen by the other party. The nonce can also encode a (partial) permutation index given to the committer. Possibly, the verifier and the committer may agree that the committer will permute the message and concatenate it with the nonce.

Interaction can be eliminated by the use of the Fiat-Shamir random oracle [9], where the next x coordinate is a function (say, xor based) of (several or all) the y values obtained so far.

4 Nesting

In a scenario where a commitment scheme is used, allowing one side to choose the one-way function can lead to adversarial behaviour. Let's consider a situation where a gambler wants to commit to a specific colour (red or black) for a bet in a casino's roulette game before the ball stops rotating.

The gambler may have doubts about the casino potentially manipulating the outcome in favour of the unchosen colour, even if they commit to their chosen colour using a one-way function (where the colour is encoded using enough bits combined with a random nonce). The gambler can choose a one-way function that exhibits collisions, meaning both red and black can be preimages of the function's output. This poses a risk since the casino needs to know the committed colour before the roulette outcome is visible. To mitigate this risk, the casino may enforce the selection of a specific one-way function. However, the gambler may suspect that the suggested function has a backdoor known to the casino, allowing them to know the committed colour in advance and potentially influence the outcome and the outcome of the bet.

In other words, if the party committing to the value determines the function, they can intentionally choose a function that has collisions for the committed value. On the other hand, if the verifier (once the committed value is revealed) selects the one-way function, it may have a backdoor that prematurely reveals the committed value.

To address these concerns, it can be advantageous to allow both parties to select a one-way function. The approach could involve using the one-way function chosen by the party to whom the secret will be revealed first, thereby eliminating planned collisions by the committing party. After that, another one-way function chosen by the committing party can be applied to the result, thus avoiding the existence of a backdoor in the first one-way function.

Care should be taken when nesting hash functions to ensure a low number of implied collisions. It is crucial to maintain a small number of collisions in each nesting stage and overall. Sampling techniques, similar to those employed in self-testing scenarios discussed in references such as [2; 7], can be utilized to estimate the probability of collisions.

Additionally, nesting can be combined with the utilization of polynomials to further eliminate potential collisions, as presented in the previous section.

5 Concluding Remarks

The pursuit of one-way functions that can be proven to be secure is closely interconnected with the investigation of a fundamental milestone in computer science, known as the $P \neq NP$ problem. By employing information-theoretically secure building blocks, such as one-time pads and secret sharing, as outlined in our approach, we have the potential to strengthen the trustworthiness of cryptographic commitments. This paper presents an efficient hash function that we have developed. By incorporating linear error detection techniques and permutations, we have effectively reduced the incidence of collisions and eliminated linearity. These measures significantly contribute to the resilience and dependability of the cryptographic commitments put forth in our proposal.

Finally, we introduce a novel and efficient candidate for a one-way function that is based on $2\lg(n)+1$ instances. This construction exhibits a distinctive property that we refer to as a "holographic" property.

References

[1] Babai, L., Moran, S.: Proving properties of interactive proofs by a generalized counting technique. Information and Computation **82**(2), 185–197 (1989)

[2] Blum, M., Luby, M., Rubinfeld, R.: Self-testing/correcting with applications to numerical problems. In: Proceedings of the twenty-second annual ACM symposium on Theory of computing. pp. 73–83 (1990)

[3] Cohen, A., Cyprys, P., Dolev, S.: Single instance self-masking via permutations. IACR Cryptol. ePrint Arch. p. 416 (2023), https://eprint.iacr.org/2023/416

[4] Cyprys, P., Dolev, S., MargalitS, O.: Owf candidates, examples implementations. GitHub (2023)

[5] Cyprys, P., Dolev, S., Moran, S.: Self masking for hardering inversions. IACR Cryptol. ePrint Arch. p. 1274 (2022), https://eprint.iacr.org/2022/1274

[6] Dolev, H., Dolev, S.: Toward provable one way functions. IACR Cryptol. ePrint Arch. p. 1358 (2020), https://eprint.iacr.org/2020/1358

[7] Dolev, S., Frenkel, S.: Extending the scope of self-correcting. In: Proceedings of the XIII International Conference Applied Stochastic Models (ASMDA-2009) (2009)

[8] Dolev, S., Lahiani, L., Haviv, Y.A.: Unique permutation hashing. Theor. Comput. Sci. **475**, 59–65 (2013). https://doi.org/10.1016/j.tcs.2012.12.047, https://doi.org/10.1016/j.tcs.2012.12.047

[9] Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) Advances in Cryptology — CRYPTO' 86. pp. 186–194. Springer Berlin Heidelberg, Berlin, Heidelberg (1987)

[10] Fischlin, M., Janson, C., Mazaheri, S.: Backdoored hash functions: Immunizing HMAC and HKDF. In: 31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018. pp. 105–118. IEEE Computer Society (2018). https://doi.org/10.1109/CSF.2018.00015, https://doi.org/10.1109/CSF.2018.00015
[11] ITU-T Study Group 15 Frame alignment and cyclic redundancy check (CRC) procedures relating to basic frame structures defined in Recommendation G.704. Standard, ITU (1991-04-05)
[12] Goldreich, O., Levin, L.A., Nisan, N.: On constructing 1-1 one-way functions. In: Goldreich, O. (ed.) Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation - In Collaboration with Lidor Avigad, Mihir Bellare, Zvika Brakerski, Shafi Goldwasser, Shai Halevi, Tali Kaufman, Leonid Levin, Noam Nisan, Dana Ron, Madhu Sudan, Luca Trevisan, Salil Vadhan, Avi Wigderson, David Zuckerman, Lecture Notes in Computer Science, vol. 6650, pp. 13–25. Springer (2011). https://doi.org/10.1007/978-3-642-22670-0_3, https://doi.org/10.1007/978-3-642-22670-0_3
[13] Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM Journal on Computing **18**(1), 186–208 (1989). https://doi.org/10.1137/0218012, https://doi.org/10.1137/0218012
[14] Handschuh, H.: SHA Family (Secure Hash Algorithm), pp. 565–567. Springer US, Boston, MA (2005)
[15] Håstad, J., Impagliazzo, R., Levin, L.A., Luby, M.: A pseudorandom generator from any one-way function. SIAM Journal of Computing **28**, 12–24 (1999)
[16] Impagliazzo, R., Naor, M.: Efficient cryptographic schemes provably as secure as subset sum. Journal of cryptology **9**(4), 199–216 (1996)
[17] Lamport, L.: Constructing digital signatures from a one way function. Tech. Rep. CSL-98 (October 1979), https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/, this paper was published by IEEE in the Proceedings of HICSS-43 in January, 2010.
[18] Leurent, G.: MD4 is not one-way. In: Nyberg, K. (ed.) Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers. Lecture Notes in Computer Science, vol. 5086, pp. 412–428. Springer (2008). https://doi.org/10.1007/978-3-540-71039-4_26, https://doi.org/10.1007/978-3-540-71039-4_26
[19] Sasaki, Y., Aoki, K.: Finding preimages in full MD5 faster than exhaustive search. In: Joux, A. (ed.) Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5479, pp. 134–152. Springer (2009). https://doi.org/10.1007/978-3-642-01001-9_8, https://doi.org/10.1007/978-3-642-01001-9_8

[20] Shannon, C.E.: Communication theory of secrecy systems. The Bell system technical journal **28**(4), 656–715 (1949)

[21] The Sage Developers: SageMath, the Sage Mathematics Software System (Version 9.8.0) (2023), `https://www.sagemath.org`

A The Insufficiency of Linear Error Detection Codes, Toy Example

We present a simple example of numerical values to the reader to show the risk of using linear error detection. The example is later used to introduce permutations to eliminate linearity.

Let us consider the following example. Define the binary vector $r = [10011010]$. Compute $f(s_1, s_2) = (s_1 \circ crc_1) \oplus (crc_2 \circ s_2) = [11010111]$. To reverse the process of this operation, define an identity matrix I with eight rows and eight columns. Let s_1^i and s_2^i be defined as the first four bits and the next four bits in the $i'th$ row of the matrix I. Define matrix A as follows: the i'th row of the matrix consists of $(s_1^i \circ crc_1^i) \oplus (crc_2^i \circ s_2^i)$, where s_1^i (s_2^i) are the four first (last, respectively) bits in the ith row of the identity matrix I and crc_j^i is the CRC result over s_j^i. Below is an example of numeric values to help the reader understand the process. We use r to denote the result of f (also called the Blackbox) over s_1, s_2, which are, in our toy example, four bits each.

$$s_1 \circ s_2 = \begin{bmatrix} 1\,0\,0\,1\,1\,0\,1\,0 \end{bmatrix} \; r = \begin{bmatrix} 1\,1\,0\,1\,0\,1\,1\,1 \end{bmatrix}$$

$$A = \begin{bmatrix} (s_1^1 \circ crc_1^1) \oplus (crc_2^1 \circ s_2^1) \\ (s_1^2 \circ crc_1^2) \oplus (crc_2^2 \circ s_2^2) \\ \vdots \\ (s_1^8 \circ crc_1^8) \oplus (crc_2^8 \circ s_2^8) \end{bmatrix}$$

$$A = \begin{bmatrix} 1\,0\,0\,0\,1\,0\,1\,0 \\ 0\,1\,0\,0\,0\,1\,0\,1 \\ 0\,0\,1\,0\,1\,1\,1\,0 \\ 0\,0\,0\,1\,0\,1\,1\,1 \\ 1\,0\,1\,0\,1\,0\,0\,0 \\ 0\,1\,1\,0\,0\,1\,0\,0 \\ 0\,1\,1\,0\,0\,0\,1\,0 \\ 0\,1\,1\,1\,0\,0\,0\,1 \end{bmatrix} \quad A^{-1} = \begin{bmatrix} 0\,1\,1\,1\,1\,1\,0\,0 \\ 1\,1\,1\,1\,0\,1\,1\,0 \\ 1\,0\,1\,1\,0\,0\,1\,1 \\ 1\,0\,0\,1\,1\,1\,0\,1 \\ 1\,1\,0\,0\,0\,1\,1\,1 \\ 0\,1\,1\,0\,1\,1\,1\,1 \\ 0\,0\,1\,1\,1\,0\,1\,1 \\ 1\,1\,0\,1\,1\,0\,0\,1 \end{bmatrix}$$

$$f(s_1, s_2) \cdot A^{-1} = \begin{bmatrix} 1\,1\,0\,1\,0\,1\,1\,1 \end{bmatrix} \begin{bmatrix} 0\,1\,1\,1\,1\,1\,0\,0 \\ 1\,1\,1\,1\,0\,1\,1\,0 \\ 1\,0\,1\,1\,0\,0\,1\,1 \\ 1\,0\,0\,1\,1\,1\,0\,1 \\ 1\,1\,0\,0\,0\,1\,1\,1 \\ 0\,1\,1\,0\,1\,1\,1\,1 \\ 0\,0\,1\,1\,1\,0\,1\,1 \\ 1\,1\,0\,1\,1\,0\,0\,1 \end{bmatrix} = \begin{bmatrix} 1\,0\,0\,1\,1\,0\,1\,0 \end{bmatrix} = s_1 \circ s_2$$

We recovered the original values from the output of the Blackbox. All-time complexity (matrix multiplication and matrix inversion) is polynomial. The previously described operations are described in the following pseudocode:

The CRC-4 toy example.

Algorithm 4: Polynomial time inversion with CRC-4-ITU

Function `BlackBox`(s_1, s_2):
  $z_1 = s_1 \circ calculate\_crc(s_1)$
  $z_2 = calculate\_crc(s_2) \circ s_2$
  return $z_1 \oplus z_2$
Function `GenerateMatrix()`:
  A_matrix = Empty 0×8 matrix
  foreach *row in (all* 8 *unit vectors)* **do**
    x = $BlackBox$(row[0:3], row[4:8])
    A_matrix.add(row, x)
  return A_matrix
Function `Inverse()`:
  $r_{12} = BlackBox(s_1, s_2)$
  A_matrix = Generate_Matrix()
  return recorvered_$s_1 s_2 = r_{12} \times A_matrix.inverse()$

Description of the pseudocode. The pseudocode commences by defining a function named "BlackBox." This function takes two binary strings, denoted as s_1 and s_2, as input. Subsequently, the CRC-4 value is computed for each of these strings. The result of the function is a binary string obtained from performing the bitwise XOR operation, represented as $z_1 \oplus z_2$. Following that, another function called "GenerateMatrix" is introduced. This function constructs a diagonal matrix with a size equivalent to the length of the binary string obtained from the previous XOR operation. Within a loop, the "BlackBox" function is invoked to calculate the values for the first four elements (designated in the $[n : m]$ list notation, where n is the index of the first element and m is the index of the last element) of each row. Subsequently, the function calculates the values for the remaining four elements. These computed values are then used to create each matrix row, denoted as $A_m atrix$. Finally, the function returns the resulting matrix. The last function, referred to as "Inverse," is the program's main function. It begins by calling the "BlackBox" function with the selected inputs s_1 and s_2, which produces a vector named r_{12}. Subsequently, this vector is utilized to restore the original values of s_1 and s_2 by performing multiplication with the inverted A_matrix.