An Experimental Evaluation of Goldreich's One-Way Function

Saurabh Kumar Panjwani[*]
Department of Computer Science and Engineering,
Indian Institute of Technology,
Bombay, India.
panjwani@cse.iitb.ac.in

July 20, 2001

Abstract

In this manuscript we present the results of the experimental evaluation of a candidate one-way function suggested in [1] and discuss the behaviour of the function against a few proposed attacks. Specifically, we study the collision properties of the function and the performance of algorithms designed to invert the function. We also propose an attack on one paricular version of the function which defies the lower bound on inversion time, as claimed in [1].

[*]The work was done while working as a visiting student at the Weizmann Institute of Science, Rehovot, Israel

1 Introduction

One-way functions form an integral part of all cryptographic schemes because they abstract the desired gap between the ease of computation of efficient algorithms and the computational infeasibility of the ones designed to break them. Intuitively speaking, a function is called one-way if it is easy to compute in one direction but hard to compute in the other i.e. we can find an efficient algorithm to compute the value of the one-way function at every point in it's domain but no efficient algorithm which finds the preimage of any point in it's range with non-negligible probability. One-way functions are simple cryptographic toolboxes using which more sophisticated schemes can be designed.

The purpose of this document is to describe the experimental analysis of one particular (suggestion for a) one-way function proposed by Oded Goldreich. The detailed description of the function is given in [1] and our purpose here is to make a useful addendum to the same by providing experimental results on the properties of this function. We don't attempt to prove the intractibility of the function in any way and we don't even try to challenge the same, even though we are inclined to believe that the function is a good candidate for being one-way.

1.1 A brief description of the function and it's variants

The suggested one-way function is a function (rather a collection of functions), f_n, that maps strings in $\{0,1\}^n$ to $\{0,1\}^n$. It uses as parameters two sets of values

- A collection (of size n) of small overlapping subsets of $[n]$, $C \stackrel{\text{def}}{=} \{S_i : |S_i| = d; S_i \in [n]; i \in [n]\}$ (typically d is chosen to be of the order of $\log(n)$) and
- A predicate $\pi : \{0,1\}^d \rightarrow \{0,1\}$.

For every string, $x = x_1x_2..x_n$, in $\{0,1\}^n$, computing $f_n(x)$ involves, first, projecting x onto each of the subsets on the collection (if $S_i = \{i_1, i_2, .., i_d\}$ then the projection of x on S_i, denoted x_{S_i}, is a string of length d which is given by $x_{i_1}x_{i_2}..x_{i_d}$) and then evaluating the predicate π on each of the n projections, thus giving us the n bit values which represent the output string. In other words, $f_n(x)$ is the bit string in $\{0,1\}^n$ equal to

$$\pi(x_{S_1})\pi(x_{S_2})...\pi(x_{S_n}) \qquad (1)$$

As should be very clear, computing the function in the forward direction is quite simple. But it appears that if the collection, C, has some desirable combinatorial properties then inverting the function is apparently a hard problem. Specifically, if C is an *expanding* collection i.e. for some k, every subset of C containing k subsets of $[n]$ be such that the union of these k subsets is of the form $k + \Theta(n)$, then the problem of inverting the function doesn't seem to have any efficient solution. We could thus use combinatorial constructions like expander graphs to construct the collection. Specifically, if we use the bipartile version of an expander, viz. $G(U,V)$ with $|U| = |V| = n$ with every vertex in $|U|$ and $|V|$ having a fixed degree d, a natural suggestion would be using the set $\{N(i)|i \in U\}$, where $N(i)$ denotes the set of neighbors of i in G as C. Such a choice of C satisfies the desired expansion property spoken of above.

It appears that the choice of the predicate used by the function is not so restrictive (we just need to avoid a few weaknesses mentioned in [1]). In practice, the suggestion implementation is to select a random predicate and to hardwire 2^d values in the form of a lookup table into the function (since d is small 2^d values do not increase the space complexity greatly). A few other predicates (suggested in [1]) which are relatively easier to hardwire and thus reduce the complexity of the function are -

- Using a random d-variant low-degree polynomial. Specifically, we think of such a polynomial as one having degree $k \in \{2, 3\}$ over the finite field of two elements. Such a polynomial can be described by $\binom{d}{k}$ bit values, which is significantly less than 2^d.

- Using a predicate that partitions its input into two equal length strings and takes their inner product modulo 2 i.e. $\pi(z_1, .., z_d) = (\sum_{i=1}^{d/2} z_i z_{i+d/2})\text{mod}2$. This predicate (we refer to it as π_{ip}) can be described even more concisely but it appears that it renders the function more vulnerable to attacks. We will further discuss the choice of this particular predicate in Section 4.

1.2 The basic attack

Before we present the more sophisticated attacks we tried on the function, it would be nice to introduce the reader to the basic suggestion for inversion of the function given in [1]. This basic attack works by examining the given output sequence bit by bit, in stages, and by maintaining a list of (incompletely known) candidate input strings which could cause the output bits (under application of the function) examined till a certain stage. For a set $S = \{i_1, i_2, .., i_k\} \subseteq [n]$ let $P(z, S) \stackrel{\text{def}}{=} z_{i_1} z_{i_2} .. z_{i_k}$ denote the projection function that projects z onto S. The the attack works as follows

Attack *Basic_Attack(y, C, n, d)*

1. Pick a random bit position, $i_1 \in [n]$, and make a list, L_1, of strings in $\{0, 1, *\}$ such that for every $z \in L_1$ it holds that $\pi(P(z, S_{i_1})) = y_{i_1}$ and $z_j = *$ if and only if $j \notin S_{i_1}$.

2. Put $U_1 \leftarrow S_1$. Put Rem $\leftarrow [n] \setminus \{i_1\}$.

3. For k in $\{2, 3, .., n\}$ do the following -

   - Choose a random $i_k \in$ Rem.
   - Put $U_k \leftarrow U_{k-1} \cup S_{i_k}$. Initialize list L_k.
   - For every $z \in L_{k-1}$, consider all possible strings $z' \in \{0, 1, *\}^n$ for which (a) $P(z', U_{k-1}) = P(z, U_{k-1})$, (b) $z'_j = *$ iff $j \notin U_k$. Append z' to L_k if $\pi(P(z', S_{i_k})) = y_{i_k}$.
   - Put Rem $\leftarrow$ Rem $\setminus \{i_k\}$.

4. Output all strings in L_n.

The running time of the attack would depend almost entirely on the size of the list of candidate strings with maximum size, $L_k^{(max)}$, which needs to be maintained during the process. It was shown in [1] that the expected size of $|L_k|$ is lower bounded by a quantity exponential in $E(k) = |\cup_{1 \le j \le k} S_j| - k$ and thus $L_k^{(max)}$ would have a lower bound corresponding to that k for which $E(k)$ is maximum. Thus, the basic attack (which proceeds by making no assumption on the nature of C) seems to suggest that a collection which is sufficiently expanding would be a good candidate for being used in the function since for such a collection there will exist a k such that $E(k)$ is of the order of $\Theta(n)$

We now go straight into describing the various experiments we performed on the function. For more details on the details of the function, the user is encouraged to read [1].

2 Collision Tests on the Iterated/Non-iterated Versions of the Function

2.1 The collision probability of a function

The collision probability of a function is defined as the probability that two elements chosen uniformly and independantly at random from the domain of the function are mapped to the same image under it's application. Thus, the collision probability can be written as -

$$p_c \quad = \quad \mathbf{Pr}[f(U_n^{(1)}) = f(U_n^{(2)})] \tag{2}$$

where $U_n^{(1)}$ and $U_n^{(2)}$ are independant random variables with uniform distribution over $\{0,1\}^n$. Clearly, a high collision probability (for a function, like ours, whose domain and co-domain coincide) implies that the function shrinks it's domain greatly.

2.2 Motivation for studying collision probability

As suggested in [1], an iterated version of the one-way function can be used for creating a pseudorandom function which can, in turn, be used to design cryptographic schemes. For the pseudorandom function to be utilizable in such a scheme, it's important that it (the pseudorandom function) and thus, the iterated version of the one-way function, doesn't have a very small range. In other words, it's important that the collision probability of the one-way function (and it's iterated version) be low.

2.3 Finding the collision probability (p_c) by exhaustive search

The easiest way to determine the collision probability is by exhaustively computing the function at all points in the domain ($\{0,1\}^n$) and by collecting information about the number of pre-images every point in the range, $R \subseteq \{0,1\}^n$, has. Consider the mapping induced by the function as a partition, P, of the domain where every element of any subset, S_j, in P maps to the same image, $j \in R$, under f. Let $N_f^{(S_j)}$ be the number of elements in S_j. Let C_{N} denote the number of subsets in P with cardinality N. (Note that the partition is well-defined since the one-way function itself is well-defined). Then the collision probability can be computed as follows (where $U_n^{(1)}$ and $U_n^{(2)}$ are again independant random variables with uniform distribution over $\{0,1\}^n$) -

$$\begin{aligned} p_c &= \mathbf{Pr}[f(U_n^{(1)}) = f(U_n^{(2)})] \\ &= \sum_{j \in R} \mathbf{Pr}[(f(U_n^{(1)}) = j) \wedge (f(U_n^{(2)}) = j)] \end{aligned}$$

Since $U_n^{(1)}$ and $U_n^{(2)}$, and hence $f(U_n^{(1)})$ and $f(U_n^{(2)})$, are independant, we have

$$\begin{aligned} p_c &= \sum_{j \in R} \mathbf{Pr}[f(U_n^{(1)}) = j] \cdot \mathbf{Pr}[f(U_n^{(2)}) = j] \\ &= \sum_{S_j \in P} \mathbf{Pr}[U_n^{(1)} \in S_j] \cdot [U_n^{(2)} \in S_j] \\ &= \sum_{N=1}^{\max_P\{|S|:S \in P\}} \quad \sum_{S_j : |S_j| = N} \mathbf{Pr}[U_n \in S_j]^2 \end{aligned}$$

$$= \sum_{N=1}^{\max_P\{|S|:S\in P\}} C_{\mathrm{N}} \cdot \left(\frac{N}{2^n}\right)^2$$

2.4 Experimental Observations for the exhaustive approach

We tried the above technique with n equal to 20 and 22 and random expanders. The results presented below are both for the iterated (upto 10 iterations) and non-iterated versions of the function.

2.4.1 Notations used

- f^i - The function iterated i times.
- $p_c^{(i)}$ - calculated collision probability for f^i
- α_i - shrinkage in domain size ($= |Range|/|Domain| = |Range|/2^n$) for f^i

In all the tables, every column tabulates values of C_N corresponding to f^i where i is the label of the column and N is the label of the row. Each experiment corresponds to a different instance of the function (with parameters as given) and in all the cases the predicate used is a random predicate. For brevity's sake, we've omitted the detailed statistics of all experiments except for one of them.

- **For** $n = 20, d = 8$

  Experiment 1 - For this experiment, we list only the values for $p_c^{(\alpha_i)}$ and α_i (and not the detailed statistics).

N	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9	i=10
$p_c^{(i)}$	$2{\cdot}2^{-20}$	$3{\cdot}2^{-20}$	$4{\cdot}2^{-20}$	$6{\cdot}2^{-20}$	$7{\cdot}2^{-20}$	$8{\cdot}2^{-20}$	$9{\cdot}2^{-20}$	$11{\cdot}2^{-20}$	$12{\cdot}2^{-20}$	$13{\cdot}2^{-20}$
α_i	0.5908	0.4235	0.3309	0.2720	0.2311	0.2009	0.1777	0.1593	0.1443	0.1319
α_i/α_{i-1}	–	0.7168	0.7813	0.8220	0.8500	0.8693	0.8845	0.8965	0.9058	0.9141

  Experiment 2 - For this particular experiment, we list the values of C_N for $N = 1, 2, .., 30$ for the varios f^i's with i in the range $[1, 10]$ in addition to $p_c^{(i)}$ and α_i.

N	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9	i=10
1	356745	182120	111405	75670	54732	41420	32569	26146	21521	18013
2	177858	124385	86565	62863	47732	37396	30132	24846	20800	17730
3	67746	72082	59155	46822	37596	30738	25428	21405	18159	15577
4	21756	38849	38618	33983	28960	24756	21080	18160	15779	13734
5	6157	20253	24508	24114	21829	19290	17018	14954	13225	11749
6	1790	10418	15536	16815	16430	15164	13862	12471	11179	10053
7	439	5189	9559	11605	12117	11821	11055	10232	9391	8688
8	103	2655	6031	8277	9076	9249	9008	8609	8101	7552
9	32	1268	3696	5646	6734	7201	7244	7039	6753	6355
10	12	624	2329	3997	5090	5688	5840	5784	5679	5473
11	1	292	1362	2645	3700	4328	4716	4840	4829	4703
12	1	153	870	1836	2807	3404	3772	3973	4045	4020
13	0	70	467	1213	1978	2573	3053	3355	3438	3489
14	0	41	341	924	1499	2084	2478	2743	2894	2932
15	0	21	210	559	1043	1504	1875	2164	2410	2490
16	0	5	114	403	779	1190	1546	1854	2057	2153
17	0	4	80	283	600	897	1217	1473	1625	1769
18	0	2	41	173	414	706	954	1188	1365	1564
19	0	0	28	129	326	550	794	982	1176	1333
20	0	0	14	84	247	450	675	871	1031	1157
21	0	0	6	53	157	337	486	670	843	972
22	0	0	5	47	140	266	440	587	747	847
23	0	0	4	30	81	185	317	438	570	688
24	0	0	3	18	56	151	255	345	429	557
25	0	1	0	13	50	112	209	333	401	499
26	0	0	0	4	25	68	147	231	337	442
27	0	0	2	4	26	66	147	230	319	387
28	0	0	1	9	30	55	100	166	224	283
29	0	0	0	4	25	54	91	138	226	298
30	0	0	0	1	12	41	59	119	168	218
$p_c^{(i)}$	$2{\cdot}2^{-20}$	$3{\cdot}2^{-20}$	$4{\cdot}2^{-20}$	$5{\cdot}2^{-20}$	$7{\cdot}2^{-20}$	$8{\cdot}2^{-20}$	$9{\cdot}2^{-20}$	$9{\cdot}2^{-20}$	$10{\cdot}2^{-20}$	$11{\cdot}2^{-20}$
α_i	0.6033	0.4372	0.3442	0.2844	0.2425	0.2116	0.1877	0.1687	0.1532	0.1403
α_i/α_{i-1}	–	0.7247	0.7873	0.8263	0.8527	0.8726	0.8871	0.8988	0.9081	0.9158

Note that the collision probability obtained for higher iterations is a bit inaccurate because the value for N was restricted to less than 30. (The collision probability was computed for N being in $\{1, 2, .., 30\}$ and higher values of N were ignored.)

- **For** $n = 22, d = 8$

  Experiment 1

N	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9	i=10
$p_c^{(i)}$	$1{\cdot}2^{-20}$	$1{\cdot}2^{-20}$	$1{\cdot}2^{-20}$	$1{\cdot}2^{-20}$	$2{\cdot}2^{-20}$	$2{\cdot}2^{-20}$	$2{\cdot}2^{-20}$	$2{\cdot}2^{-20}$	$2{\cdot}2^{-20}$	$3{\cdot}2^{-20}$
α_i	0.6087	0.4428	0.3495	0.2893	0.2470	0.2157	0.1915	0.1723	0.1566	0.1435
α_i/α_{i-1}	–	0.7274	0.7893	0.8278	0.8538	0.8733	0.8878	0.8997	0.9089	0.9163

Experiment 2

N	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9	i=10
$p_c^{(i)}$	$1{\cdot}2^{-20}$	$1{\cdot}2^{-20}$	$1{\cdot}2^{-20}$	$2{\cdot}2^{-20}$	$2{\cdot}2^{-20}$	$2{\cdot}2^{-20}$	$2{\cdot}2^{-20}$	$3{\cdot}2^{-20}$	$3{\cdot}2^{-20}$	$3{\cdot}2^{-20}$
α_i	0.5888	0.4189	0.3252	0.2655	0.2242	0.1940	0.1709	0.1527	0.1380	0.1259
α_i/α_{i-1}	–	0.7114	0.7763	0.8164	0.8444	0.8652	0.8809	0.8935	0.9037	0.9123

2.5 Finding p_c by random sampling of the domain

Unfortunately, the limitation of computational capacities does not allow us to obtain estimates of collision probability by the exhaustive method for values of n significantly larger than 20. For such values, we obtained approximate estimates to p_c by randomly sampling points in $\{0,1\}^n$ and computing f at these points only. The size of the sampling set was taken to be $\sqrt{2^n} = 2^{n/2}$, the expected number of samples after which a repetition in $\{0,1\}^n$ is observed (by the *birthday paradox*). Even though with this we couldn't reach very high values of n, it gave us some improvement above the limit we could reach with exhaustive search.

The actual estimation involved the following algorithm. Let the number of points sampled be denoted by M (in our case, we used $M = 2^{n/2}$). Let x_1, x_2,..,x_M denote the values of the samples. Apply the function at each of the x_i's and compute the set $V = \{(i,j)|f(x_i) = f(x_j) \wedge i < j\}$. This computation can be carried out by maintaining a data structure for all the points, J, in the range obtained by applying f on the samples, which also includes information about the number of times, C_j, each point, $j \in J$, is obtained. The size of V will be $\sum_{j\in J}\binom{C_j}{2}$. Output $\frac{|V|}{|\{(i,j)|i<j\}|} = \frac{|V|}{\binom{M}{2}}$.

Analysis : There are two important conditions that the above algorithm should satisfy to be able to serve as an effective technique for estimating the collision probability -

- *The expected value of the output should be p_c or very close to p_c.*
- *With high probability, the deviation of the output from it's expectation should not be large.*

The first of the above conditions can be seen to hold in the following manner. Let η_{ij} denote the indicator random variable for two independant samples, x_i and x_j, chosen uniformly at random from $\{0,1\}^n$ to have the same image under f i.e. $\eta_{ij} = \chi[f(x_i) = f(x_j)]$. Thus, η_{ij} is 1 with probability p_c (i.e. when $f(x_i) = f(x_j)$) and 0 with probability $1 - p_c$ (i.e. when $f(x_i) \neq f(x_j)$). Clearly, the expected size of V in the algorithm is the expectation value of the sum of these η_{ij}'s taken over all possible pairs (i,j). Thus,

$$\begin{aligned} E[Output] &= E\left[\frac{|V|}{\binom{M}{2}}\right] \\ &= \frac{1}{\binom{M}{2}} \cdot E[|V|] \end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{\binom{M}{2}} \cdot E\left[\sum_{i,j\in[M];i<j} \eta_{ij}\right] \\
&= \frac{1}{\binom{M}{2}} \cdot \sum_{i,j\in[M];i<j} E[\eta_{ij}] \\
&= \frac{1}{\binom{M}{2}} \cdot \binom{M}{2} \cdot p_c \\
&= p_c
\end{aligned}
$$

To prove the second part, we'll use Chebyshev's inequality. Since η_{ij} is an indicator random variable for two independant random samples, x_i and x_j, to have the same image under f, η_{ij} and η_{kl} (which has the same distribution as η_{ij}) will be independant for every $1 \leq i,j,k,l \leq M$ ($i \neq k$ or $j \neq l$). Hence the numerator of the output is a sum of $m = \binom{M}{2}$ pairwise independant and identically distributed random variables (say β_k's) with the same expectation p_c and the same variance, say $Var(\beta)$. That is, our output is equal to the following quantity.

$$\frac{\sum_{k=1}^{m} \beta_k}{m}$$

Using Chebyshev's inequality, we get that for every $\epsilon > 0$

$$
\begin{aligned}
\mathbf{Pr}\left[\left|\frac{\sum_{k=1}^{m} \beta_k}{m} - p_c\right| \geq \epsilon\right] &\leq \frac{Var(\beta)}{\epsilon^2 m} \\
&= \frac{p_c(1-p_c)}{\epsilon^2 m}
\end{aligned}
$$

Hence, the probability that the output deviates from its expected value, p_c, is bounded above by $\frac{p_c - p_c^2}{\epsilon^2 m}$, which becomes smaller as M, and thus, m, becomes larger. Thus, if ϵ is of the order of p_c and we sample $M = \sqrt{2^n}$ points in our domain, the probability of our output deviating from p_c by a value greater than ϵ will be bounded above by a small constant.

2.6 Results obtained for the random sampling technique

In all the tables that follow, we tabulate the estimated collision probability for two cases, $n = 40, d = 10$ and $n = 20, d = 8$. The quantity c refers to the difference in the number of points sampled and the number of images obtained for a paricular sampling set. Recall that the number of sampled points is $2^{n/2}$.

- **For n=40, d=10**
  - **Experiment 1**

N	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9	i=10
c	0	0	1	1	2	2	3	4	4	5
$p_c^{(i)}$	$1 \cdot 2^{-40}$	$1 \cdot 2^{-40}$	$3 \cdot 2^{-40}$	$3 \cdot 2^{-40}$	$5 \cdot 2^{-40}$	$5 \cdot 2^{-40}$	$7 \cdot 2^{-40}$	$9 \cdot 2^{-40}$	$9 \cdot 2^{-40}$	$11 \cdot 2^{-40}$

- **For n=20, d=8**

– **Experiment 1** - In the following 2 tables, we make a comparison of the random sampling technique with the exhaustive search technique. The function (rather instance of the function) used was same in both cases.

(i) Result for the random sampling technique :-

N	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9	i=10
c	0	0	1	2	2	2	3	3	3	3
$p_c^{(i)}$	$1{\cdot}2^{-20}$	$1{\cdot}2^{-20}$	$3{\cdot}2^{-20}$	$5{\cdot}2^{-20}$	$5{\cdot}2^{-20}$	$5{\cdot}2^{-20}$	$7{\cdot}2^{-20}$	$7{\cdot}2^{-20}$	$7{\cdot}2^{-20}$	$7{\cdot}2^{-20}$

(ii) Result for the exhaustive search technique :-

N	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9	i=10
$p_c^{(i)}$	$2{\cdot}2^{-20}$	$3{\cdot}2^{-20}$	$4{\cdot}2^{-20}$	$6{\cdot}2^{-20}$	$7{\cdot}2^{-20}$	$8{\cdot}2^{-20}$	$9{\cdot}2^{-20}$	$10{\cdot}2^{-20}$	$11{\cdot}2^{-20}$	$12{\cdot}2^{-20}$
α_i	0.6007	0.4330	0.3397	0.2796	0.2376	0.2066	0.1827	0.1600	0.1482	0.1354

2.7 Observations

Some observations we can make from these results are

- The collision probability for the non-iterated version of the function with (i.e. f^i) is consistently of the order of $2 \cdot 2^{-n} = 2 \cdot (|Domain|)^{-1}$, which means that on an average every point in the range has two preimages. Thus, the size of the range of the function is of the order $\Theta(|Domain|)$, which is good enough for effective utilization of the function (in cryptographic schemes).
- The shrinkage in size of the domain (for the values of n for which we could perform exhaustive search) reduces in value with increase in the number of iterations, as one would expect to happen.

3 Comparisons between Inversion Time and Expansion

In this section we will discuss the behaviour of the function with respect to the inversion algorithm given in [1] and make observations about the relation between the running time of this algorithm and the expansion of the collection used by the function. The time complexity of the algorithm is lower bounded by a quantity exponential in the expansion of the collection and our purpose here is to determine the seperation of the actual running time from this bound.

We stress that when we refer to expansion of a collection, $C \equiv \{S_i | S_i \subseteq [n]; |S_i| = d; 1 \leq i \leq n\}$, we mean the following quantity (defined in [1]) -

$$\max_{1 \leq k \leq n} \min_{I:|I|=k} \{|\cup_{i \in I} S_i| - k\} \tag{3}$$

which is somewhat different from the notion of expansion expressed in [2] and [3]. We use the properties of expander graphs to bound the above quantity i.e. using the fact that the collection has been generated from an expander graph, we can derive a lower bound on the expansion (as defined above) in terms of certain parameters of the graph. These parameters are the size of the graph, n, which is the same as the size of C, the degree of each vertex, d, which is the same as the size of each set in C and the seperation between the first and second eigenvalues of the normalised adjacency matrix of the graph [2].

3.1 Computing the lower bound on expansion

We use the bipartile construction of expander graphs for generating C. The construction is given in [1]. For deriving the lower bound on the expansion we will make use of Tanner's Theorem [2].

Let G be a bipartile graph with classes of vertices U and V, where $|U| = |V| = n$ and the degree of every vertex in U and V is d. Let A be the adjacency matrix of G (A is of dimension $n \times n$). Let λ_1 and λ_2 be the two largest eigenvalues of AA^T such that $\lambda_1 > \lambda_2$. (λ_1 is known to be always equal to d^2). Then, Tanner's Theorem states that for any set of vertices, $X \subseteq U$, if $N(X)$ denotes the set of all neighbors of X in G, then

$$\begin{aligned} |N(X)| &\geq \frac{\lambda_1 |X|}{(\lambda_1 - \lambda_2)|X|/n + \lambda_2} \\ &= \frac{d^2 |X|}{(d^2 - \lambda_2)|X|/n + \lambda_2} \end{aligned}$$

For the normalised adjacency matrix, A_{norm}, of G, the largest eigenvalue is 1 and if we denote its second largest eigenvalue by λ, then the following relation can be seen to hold

$$\lambda_2 = d^2 \lambda^2 \tag{4}$$

Using the above two equations, we get the following

$$\begin{aligned} |N(X)| &\geq \frac{d^2 |X|}{d^2(1 - \lambda^2)|X|/n + d^2 \lambda^2} \\ &= \frac{|X|}{(1 - \lambda^2)|X|/n + \lambda^2} \\ &= \frac{|X|}{1 - (1 - \lambda^2)\left[1 - \frac{|X|}{n}\right]} \end{aligned}$$

The above relation suggests that the further λ, and, thus λ^2, is from 1, the largest eigenvalue of A_{norm}, the larger is the neighboring set for every subset in U and thus, the better expander G is. Indeed, we can obtain a relationship between the expansion co-efficient of G and the seperation between the two largest eigenvalues of A_{norm} (or even of AA^T) using Tanner's theorem. This even gives us an algebraic definition of an expander graph.[1]

We will use the above obtained relation for deriving the expansion bound in terms of the second largest eigenvalue of the normalised adjacency matrix. Let $c = \lambda^2$. Let $G(U,V)$ be the expander graph from which the collection, C, is derived. Then the expansion of C, say $E(C)$, can be written as

$$\begin{aligned} E(C) &= \max_{x\in[n]} \min_{X:X\subseteq U;|X|=x} \{|N(X)| - |X|\} \\ &= \max_{x\in[n]} \{\{\min_{X:X\subseteq U;|X|=x} |N(X)|\} - x\} \\ &\geq \max_{x\in[n]} \{\frac{x}{(1-c)x/n + c} - x\} \\ &= \max_{x\in[n]} \{\frac{nx}{(1-c)x + cn} - x\} \end{aligned}$$

Assuming the expression being maximized to be a continuous function of x and the expansion to be continuosly related to x as well, we get that $E(C)$ is maximum for $x = x_{(max)}$ where

$$\begin{aligned} \frac{x_{(max)}}{n} &= \frac{\sqrt{c} - c}{1-c} \\ &= \frac{\lambda - \lambda^2}{1-\lambda^2} \end{aligned}$$

and the lower bound on $E(C)$ can be obtained by substituting this value in the expression for x in $E(C)$ as follows

$$\begin{aligned} E(C) &\geq (\frac{\lambda-\lambda^2}{1-\lambda^2})(\frac{1}{\lambda-\lambda^2+\lambda^2}) - (\frac{\lambda-\lambda^2}{1-\lambda^2}) \\ &= \frac{\lambda(1+\lambda^2) - 2\lambda^2}{\lambda(1-\lambda^2)} \end{aligned}$$

It is important to note that we don't obtain an exact expression for the expansion but only a lower bound on it. This is because Tanner's Theorem itself provides only a bound and indeed this bound is quite loose. Thus, even our bound for expansion is very loose and experiments reveal that the actual expansion (in the cases we could manage to compute it) is much greater than this bound.

[1] The advantage of using eigenvalues to obtain a lower bound on the expansion of a graph, as is usually done, is that they are much easier to compute than the co-efficient of expansion. They give us an easy-to-prepare certificate for an expander, which is a lower bound on the expansion co-efficient. It is for the same reason that we make use of the expansion bound in our analysis. But the utility derived is not much since the bound on expansion we get using them is way too loose.

3.2 Experimental results

In the table given below, we illustrate the results of our experiments with the above mentioned inversion algorithm and compare its running time with the actual expansion (for low values of n) and the expansion bound (for higher values of n). The entries in the column labelled *Running Time* are the powers to the base 2 which equal the length of the list of maximum length during the course of the attack for that particular case. For each case (i.e. for each choice of parameter values), we performed the inversion on five differnt strings and the value of running time tabulated is the median of the values obtained for all the runs. We have listed the values of λ for each case and the correspondingly derived lower bound on expansion (*Expansion Bound*), too. For the cases in which we could compute the expansion directly (by exhaustively computing $|N(X)| - |X|$ for every possible subset, $X \subseteq [n]$ and then applying equation (2)), we have also listed the respective values, $E(C)$. The predicate used in all the cases was random.

Parameters	Expander Used	λ	Expansion Bound	$E(G)$	Running Time(t)
$n = 20, d = 8$	Random	0.51	6.49	10	15.03
$n = 25, d = 8$	Random	0.53	7.70	12	18.02
$n = 25, d = 10$	Random	0.41	10.71	14	17.96
$n = 30, d = 8$	Random	0.54	9.00	–*	22.07
$n = 31, d = 6$	Alon's	0.37	14.25	–*	23.62
$n = 31, d = 6$	Random	0.64	6.80	–*	18.20

* - *Could not compute*

It is important to see that the actual value of expansion is greatly seperated from the computed lower bound in all of the cases where we could compute the former. Also, the running time has value much more that what even the exact expansion takes. The latter difference in values could be reduced greatly by implementing an improved attack on the function and we'll move in this direction in the following section.

At this point it is important to highlight the fact that in the above mentioned experiments and in the ones we'll talk about in the next section, the collection and the predicate for a particular pair of values of n and d were kept constant. For example, wherever we talk of an instance of the function that uses $n = 20$ and $d = 8$, we refer to the same instance (the one which we've used in the first case above). The observations could be very different for instances with the same parameters and we don't claim that the instances we have used are representative of the typical behaviour of the function for the respective values. This was done only for the sake of consistency and ease of comparison.

4 Further Cryptanalysis of the One-Way Function

In this section we discuss some attacks that we tried out on the one-way function and it's variations. Most of our experiments revolved around the version of the function which uses the predicate, π_{ip} defined by $\pi_{ip}(z_1, z_2, ..., z_d) = (\sum_{i=1}^{d/2} z_i \cdot z_{i+d/2}) \bmod 2$, where $z_k \cdot z_l$ is the product of the 2 bit values z_k and z_l. It appears that the usage of this predicate renders the function easier to invert and we were indeed able to obtain an inverting algorithm that runs in time exponential in a quantity significantly less than the expansion of the collection of sets used by the function. Our results further establish the fact that the choice of the predicate is indeed important in the behaviour of the function.

4.1 The original attack revisited

We first look at the algorithm suggested in [1] for inversion of the function and discuss an improvement on this algorithm that uses a good choice of the order in which we examine the bits in the output string. Consider a string $y \in \{0,1\}^n$ that we are trying to invert for a version of the function, $f : \{0,1\}^n \to \{0,1\}^n$. Let $C \equiv \{S_i | S_i \subseteq [n]; |S_i| = d; 1 \leq i \leq n\}$ be the collection of sets and $\pi : \{0,1\}^d \to \{0,1\}$ the predicate used by f. For any set $S = \{i_1, i_2, .., i_k\} \subseteq [n]$ of size k and $z \in \{0,1\}^n$ let $P(z, S) \stackrel{\text{def}}{=} z_{i_1} z_{i_2} .. z_{i_k}$ denote the projection function that projects z onto S. Our attack proceeds in the following manner.

Attack *Invert_in_good_order(y, C, n, d)*

1. Pick a random bit position, $i_1 \in [n]$, and make a list, L_1, of strings in $\{0, 1, *\}$ such that for every $z \in L_1$ it holds that $\pi(P(z, S_{i_1})) = y_{i_1}$ and $z_j = *$ if and only if $j \notin S_{i_1}$.
2. Put $U_1 \leftarrow S_1$. Put Rem $\leftarrow [n] \setminus \{i_1\}$.
3. For k in $\{2, 3, .., n\}$ do the following -
   - Choose $i_k \in$ Rem such that $|S_{i_k} \cap U_{k-1}| = \max_{j \in \text{Rem}}\{|S_j \cap U_{k-1}|\}$.
   - Put $U_k \leftarrow U_{k-1} \cup S_{i_k}$. Initialize list L_k.
   - For every $z \in L_{k-1}$, consider all possible strings $z' \in \{0, 1, *\}^n$ for which (a) $P(z', U_{k-1}) = P(z, U_{k-1})$, (b) $z'_j = *$ iff $j \notin U_k$. Append z' to L_k if $\pi(P(z', S_{i_k})) = y_{i_k}$.
   - Put Rem $\leftarrow$ Rem $\setminus \{i_k\}$.
4. Output all strings in L_n.

The only difference between this algorithm and the one desribed in [1] is that the current algorithm looks for a specific order in which the bits in the output string are examined depending on the nature of C. In contrast, the previous one chose the given order for inversion oblivious of C (i.e. $i_1 = 1, i_2 = 2, .., i_n = n$). It was shown in [1] that the length of the list at step k is lower bounded by a quantity exponential in the expansion of the set $\{i_1, .., i_k\}$, i.e. $|\cup_{1 \leq j \leq k} S_{i_j}| - k$. But experimentation (see section 4.1.1) shows that the algorithm actually behaves much worse that that. Thus, an obvious choice for the bit position at the kth step would be one that keeps the expansion smallest, which is what the above algorithm does. Indeed, the running time of the attack is still lower bounded by a quantity exponential in the quantity $\max_{1 \leq k \leq n} \min_{I:|I|=k}\{|\cup_{i \in I} S_i| - k\}$, which is $\Theta(n)$ for an expanding collection.

4.1.1 Evaluation of the improved version

Our results show that the introduction of the idea of ordering of bit positions improves the actual running time quite significantly. The following table enumerates a few cases of comparison between the two versions of the attack. In all the cases the predicate used was random.

The entries in the last two columns (t_{Basic_Attack} and $t_{Invert\_in\_good\_order}$) are the powers to the base 2 which equal the length of the list, $L_k^{(max)}$, of maximum length during the course of the respective attack on a particular string. Each pair of entries in the third and fourth columns is such that for every row (i.e. every set of parameters) the difference between these entries is the median of the differences between five pairs of such values, each obtained for the same set of parameters (i.e. the same collection and the same predicate) but a different output string being inverted.

Parameters	Type of Expander	Expansion of C	t_{Basic_Attack}	$t_{Invert\_in\_good\_order}$
$n = 20, d = 8$	Random	10	15.03	10.90
$n = 25, d = 8$	Random	12	18.02	12.90
$n = 31, d = 6$	Alon's	14.25*	23.62	17.35
$n = 31, d = 6$	Random	6.8*	18.20	13.70

$*$ - *These entries are not the actual values of the expansion but the bounds on the expansion (section 3.1).*

4.2 The predicate π_{ip} seems to be weaker than a random predicate

We now turn our attention to the π_{ip}-version of the function and focus entirely on tests with this version. Our ultimate aim is to establish the fact that using $\pi_{ip}$the function's inversion can be attained in time significantly less than that for the ordinary version. Our conjecture is that the π_{ip}-version of the function is a weaker version and, thus, should not be used in practice.

Intuitively speaking, π_{ip}, even though not perfectly linear, is very close to being a linear predicate. To see this, observe that if the values of all the input bits in one half of the positions in a particular set are known to be either 0 or 1, then the corresponding output bit is linearly related to the remaining input bits. Thus, a natural suggestion to invert the π_{ip}-version would be to fix values at bit positions (in the input string to the function) one by one and at some stage (after having fixed around $n/2$ bits) every bit in the output would become a linear combination of the remaining bits. Thus, the problem would then reduce to checking for all possible combinations of the fixed bits and solving a linear system for every fixation.

Another crucial observation is that the predicate, π_{ip}, is more biased than any random predicate would be expected to be. The fraction of inputs for which the output is 1 is always less than that for which it is 0 (Specifically, for every π_{ip}: $\{0,1\}^d \to \{0,1\}$ exactly $2^{d-1} + \epsilon$ inputs yield a 0 and $2^{d-1} - \epsilon$ yield a 1 where $\epsilon = 2^{d/2-1}$). Thus, if f uses π_{ip}as its predicate, then the distribution of $f(X^{(n)})$ where $X^{(n)}$ is uniformly distributed over $\{0,1\}^n$ will have lower entropy than what we would expect for any random predicate (that is, a predicate for which the 2^d possible output values are each chosen uniformly at random from the set $\{0,1\}$).

Also, experimentation reveals that *in most cases* and using the same inversion algorithm, *Invert_in_good_order* and a particular string $x \in \{0,1\}^n$, an instance of the function $f_{weak} : \{0,1\}^n \to \{0,1\}^n$ that uses a given collection of sets and π_{ip}as its parameters can be inverted faster (when inverting $f_{weak}(x)$) than an instance, $f_{normal} : \{0,1\}^n \to \{0,1\}^n$ which uses the same collection but a random predicate (when inverting $f_{normal}(x)$). We obtained such a result in almost all the cases that we tried and this greatly supports our conjecture that $\pi_{ip}$is a weaker predicate than a truely random one.

4.3 An attack that breaks the expansion bound for the π_{ip}-version

We now present the main result of this section, namely an inversion algorithm on the π_{ip}-version of our function which defeats the lower bound in terms of expansion as given in [1]. Our result also helps us in establishing the significance of the predicate in the overall structure of the function. The attack we present here has been tried and tested for various parameters of the function (and also for various input strings) and in all the cases drastic differences between the expansion bound and the running time of the attack were observed. In the discussion that follows we refer to the π_{ip}-version of the function (rather any instance of the function which uses π_{ip}) as f_{ip}.

4.3.1 Capitalization on the simple structure of π_{ip}: Motivating Discussion

To begin with, let us see and analyze what happens when our original attack trying to invert f_{ip}on a string fixes a few of the bits in the input string being attempted to find. We refer to the bits in the input string which have been guessed (whether correctly or incorrectly) by the attack as the *known* bits and those that are yet to be guessed as the *undiscovered* bits. The list of strings that it maintains are referred to as *candidate* inputs (for the given output) and each candidate input has some known bits, with the rest being undiscovered. Specifically, consider the case for a particular set, S_i, in C for which some (but not all) of its elements (say a subset S_i') are bit positions corresponding to known bits. Even though there may be quite a few bit positions in S_i for which the values are undiscovered, we *can* obtain vital information about these values using S_i'. Our aim here is to capitalize on this vital information.

The structure of π_{ip}ordains some restrictions on the values that the undiscovered bits' positions in S_i can have if the known values are from a particular class. The restrictions on theses values are there for any general predicate but what's important is that *for some cases the values are much easier to discover with π_{ip}than with other predicates.* For example, if all the bit positions in the first half of S_i (i.e. the positions with indices $1, 2, .., d/2$ in the vector that S_i represents) have values 0 in a particular candidate string then the output bit position, y_i, corresponding to S_i, must have value 0. If such is not the case, the candidate string can be ruled out immediately.

Let's put things more formally now. For any candidate input, z, and with $S_i = \{i_1, i_2, .., i_d\}$, we try to evaluate the predicate on the bit positions in S_i i.e. we try and find $e_i = (\sum_{j=1}^{d/2} z_{i_j} \cdot z_{i_{j+d/2}})\text{mod}2$. We use variables, i.e. z_{i_k}'s for the undiscovered bits' positions and substitute values for the known bits' positions. The expression, e_i, we equate to the bit value y_i which corresponds to S_i in the output string being inverted. We may end up in either of the following four classes by doing this

1. e_i may turn out to be a constant, with this constant not being equal to y_i. In such a case, our candidate gets disqualified.

2. e_i may turn out to be linear in exactly one variable, z_{i_k}, in which case the resulting equation will be something like $z_{i_k} = c$ where c is a constant. The vital information we get here is that if our candidate has to remain a candidate then z_{i_k} must have value c. Thus, we benifit by *discovering* exactly one additional bit in the candidate.

3. e_i may be quadratic in exactly two variables, z_{i_k} and z_{i_l} and the resulting equation is of the form $z_{i_k} \cdot z_{i_l} = 1$. This can only happen if both the variables have values 1 and, thus, we can obtain a benifit of *discovering* 2 additional bits here.

4. Any other form that the equation takes will not give us immediately utilizable benifit and we don't look at such forms.

It is important to observe that we can derive such benifits owing only to the oversimplified structure of π_{ip}. This is because $\pi_{ip}$is essentially a sum over $d/2$ pairs of values and if all but one of these pairs get zeroed out or some of them yield a 1, we are in good luck. Furthermore, our luck greatly improves with an increase in the number of known bits.

What increases the utility of the benifits spoken of above is the fact that *the discovered values of a few undiscovered bits can, in turn, cause other undiscovered bits to be discovered.* In other words, once we know the values at some bit positions over and above the ones for which the values were already known, cases 1, 2 and 3 above may become applicable to sets in the collection for which they weren't applicable before knowing these values. Intuitively speaking, this triggers a **chain process** of discovering bits and eliminating candidates and in some suitably triggered cases, we may end up discovering all the bits in the input. A good triggering set of known bits would be one which is (a) reasonably large and (b) has a sufficiently large fraction of 0's. It is not clear how we could quantify *reasonably/sufficiently large* here - these are just intuitions which come to the mind. It is not even clear what the fixpoint of the chain process would be and whether it can be expressed directly in terms of the fraction of 1's in the known bits, the collection, C, and the output string, y. Indeed, the analysis of this chain process is quite difficult because it is likely to behave very differently for even a small variation in the values of the known bits or the output string.

4.3.2 Some definitions and notations

We will make some preliminary definitions with respect to our attack before we actually present it. Let f_{ip}, π_{ip}, C and P be as defined earlier in this section. Specifically, from now on we consider $f_{ip}$and $\pi_{ip}$to be fixed instances with respect to some given values of n and d. Let $y \in \{0,1\}^n$ be the string being attempted to invert. We'll use $z \in \{0,1,*\}^n$ to denote candidate inputs with every $*$ representing an undiscovered bit. Define -

- S_z as the set of bit positions in z corresponding to discovered bits i.e. $S_z \stackrel{\text{def}}{=} \{i | i \in [n]; z_i \neq *\}$.
- $discoverable(z, y)$ as the string $z'_{max} \in \{0,1,*\}^n$ which is the string with the maximum possible value of $S_{z'}$ that can be obtained by applying the chain process spoken of above i.e. by repeatedly applying cases 2 and 3 as long as they are applicable. $discoverable(z, y)$ is *null* if at any stage of this repeated application, the string gets disqualified (case 1).

4.3.3 The inversion algorithm $Invert\_f_{ip}$

We are now set to present the entire algorithm with implementation details. We hope that the idea is clear to the reader by now and that we can skip detailed explanation of the algorithm. An important difference between the candidates in this attack and those in the original one is that here at every stage the known bits in the candidates are different for all of them and thus need to be derived seperately for each candidate (Recall that for the previous case, the known bits at stage k were only the ones in the positions of $\cup_{j \in \{i_1,..,i_k\}} S_j$). This makes one part of the algorithm a bit more intensive than the previous one but the advantage we get from discovering undiscovered bits using the known bits makes up for more than this loss.

Attack $Invert\_f_{ip}(y,\ C,\ n,\ d)$

1. Choose a random bit position, i_1, in the output sequence to invert on. (Alternatively, we can begin with the first bit in y).

2. Form a list L_1 of candidate inputs as in *Invert_in_good_order*.

3. For every $z \in L_1$ -

   (a) Put $z' \leftarrow discover(z, y, C, n, d)$.

   (b) If $z' = null$, delete z from L_1.

   (c) If $z' \neq null$, replace z with z' in L_1.

4. For k in $\{2, 3, .., n\}$ do the following -

   (a) Initialize list L_k.

   (b) For every $z \in L_{k-1}$ do the following -

      i. Put Known $\leftarrow known\_bits(z, n)$. Compute $Known\_sets = [n] \setminus \{l | \forall j \in S_l, z_j \in Known\}$.
      `(* Known is the set of bit positions for which the values are known in the current candidate *)`
      `(* Known_sets is the set of sets for which the values at bit positions corresponding to all their elements are known *)`

      ii. Put Rem $\leftarrow [n] \setminus Known\_sets$.

      iii. Choose $i_k^{(z)}$ such that $|S_{i_k^{(z)}} \cap \text{Known}| = \max_{j \in \text{Rem}} |S_j \cap \text{Known}|$.

      iv. Put Known_new $\leftarrow$ Known $\cup S_{i_k^{(z)}}$.

      v. Consider all possible strings $z' \in \{0, 1, *\}^n$ for which (a) $P(z', \text{Known}) = P(z, \text{Known})$, (b) $z'_j = *$ iff $j \notin$ Known_new.

      vi. If $\pi_{ip}(P(z', S_{i_k^{(z)}})) = y_{i_k^{(z)}}$ then
         Put $z'' \leftarrow discover(z', y, C, n, d)$.
         If $z'' = null$ then
            Discard z''.
         Else if $known\_bits(z'', n) = n$ then
            Output z''.
         Else
            Append z'' to L_k.

5. Output all strings, if any, in L_n.

The above presented algorithm uses, as subroutines two important procedures, namely

- $discover(z, y, C, n, d)$ - This procedure returns $discoverable(z, y)$, as defined previously. The idea is to keep performing iterations over all the sets in the collection and checking if any of the cases 1, 2 or 3 (section 4.3.1) hold for any of the collections at every iteration. If case 1 is found to hold, we return *null*. If cases 2 or 3 hold we modify z as the case demands and carry on iterating. We keep performing iterations till a stage is reached where z cannot be modified any further. This is the desired fixpoint of the procedure.

  Procedure *discover(z, y, C, n, d)*

  1. Put $t \leftarrow z$.
  2. Do the following -

(a) Put $extrabit \leftarrow 0$.

(b) For all $k \in \{1, 2, .., n\}$ do the following -

i. Let $S_k = \{j_1, .., j_d\}$

ii. $e_k \leftarrow (\sum_{i=1}^{d/2} z_{j_1} \cdot t_{j_1+d/2}) \bmod 2$.

`(*` e_k `will be in terms of constants and variables *)`

iii. Consider equation $e_k = y_k$ and see if it satisfies any of the conditions in cases 1, 2, or 3.

iv. If case 1 holds then

· Return *null*.

v. If cases 2 or 3 hold then

· Modify t as the respective case demands.

· Put $extrabit \leftarrow 1$

3. Repeat until $extrabit$ is 0

4. Return t.

- *known_bits*(z, n) - This procedure returns S_z. We simply iterate over all the bits in z and maintain a counter to count the bits which are not $*$.

Procedure *known_bits(z, n)*

1. Put $counter \leftarrow 0$.
2. For $k \in \{1, .., n\}$ do the following -

(a) If $z_k \neq *$ then

$counter \leftarrow counter + 1$

3. Return *counter*

We repeat that the number and nature of known bits could be drastically different for different candidates even during the same stage of the attack (i.e. when k has a fixed value in the loop beginning in step 4 of *Invert_*f_{ip}). This is because the success of the procedure *discover* depends largely on the specific values of the bits (i.e. the known bits) in z and y, besides depending on the nature of C. Indeed, the procedure *discover* is the core of the entire algorithm and without it the algorithm is absolutely the same as *Invert_in_good_order*.

It should be noted that we need to carry on running *Invert_*f_{ip} till the very end and that we cannot afford to stop it the moment we discover the first candidate string, z_1, with $S_{z_1} = n$. This is because our function is not a permutation and as already established in section 2, typically every output string would have more than one preimages.

4.3.4 Experimental results and comparison with the original attack

Our results reveal that the running time of *Invert_*f_{ip} is less than that of *Invert_in_good_order* by a factor which is exponential in 35% of what the logarithm to the base 2 of the running time of *Invert_in_good_order* is. Since the actual expansion of the collection (we refer to the definition in Section 3) is very close to the latter quantity (for the cases where we could compute expansion), this means that the algorithm is defeating the originally defined lower bound on inversion time (in [1]) by a significant quantity.

The following table illustrates a comparison between the two attacks for various choices of parameters. (In the discussion that follows, when we refer to running time of either of the inversion algorithms being considered, we mean the logarithms to the base 2 of the maximum list sizes maintained in the respective algorithm). The values presented in the columns labelled $t_{Invert\_in\_good\_order}$ and $t_{Invert\_f_{ip}}$, as before, are such that for every choice of parameters (i.e. every row in the table) the difference between the running times for the listed pairs is the median of the differences between the running times for five different pairs corresponding to runs of the attacks with five different output strings but the same choice of parameters. We also tabulate (in the last column) the value of k (as used in the listing of the algorithm i.e. k is the variable on which the *for* loop in step 4 iterates), $k_{complete}$, at which *Invert_*f_{ip} manages to complete outputting all the preimages of the given output and the listed value is again the median of the values obtained in five different runs of the attack.

Parameters	Expander Used	Expansion Bound	$t_{Invert\_in\_good\_order}$	$t_{Invert\_f_{ip}}$	$k_{complete}$
$n = 20, d = 8$	Random	6.49*	10.90	4.91	2
$n = 25, d = 8$	Random	7.70*	12.90	7.71	3
$n = 30, d = 8$	Random	9.0	14.96	9.03	4
$n = 40, d = 8$	Random	10.00	18.32	11.99	4
$n = 46, d = 8$	Random	12.73	21.72	14.06	5
$n = 31, d = 6$	Alon's	14.25	15.56	7.55	4
$n = 31, d = 6$	Random	6.80	12.48	6.44	5
$n = 57, d = 8$	Alon's	22.71	*c.n.c.***	20.53	4
$n = 57, d = 8$	Random	12.48	*c.n.c.***	17.53	8

$*$ *For these cases we could actually compute the expansion and it was found to be 10 and 12 for* $n = 20$ *and* $n = 25$ *respectively, which is much greater (for both cases) than what the bound gives us and is almost equal to the running time of Invert_in_good_order. Based on these observations, we could hope that even for the rest of the cases the running time of Invert_in_good_order is a close approximation to the expansion.*
$**$ *c.n.c* $\equiv$ *Could not compute (due to memory limitations)*

A crucial observation to make from the table is that the value of $k_{complete}$ is extremely small (compared to n) in all the cases. This means that we were able to determine the complete input string much early on than one would typically expect to, which further implies that our *chain process* works extremely fast in practice. Also, we could observe that after running the *for* loop (in step 4 of the attack) only twice we were able to obtain at least one preimage in almost all the cases (even for some cases using $n = 57$).

Another important point is that the difference between the actual running times of the two algorithms is less than the difference between the maximum list sizes for them. This is because *Invert_*f_{ip} is relatively more intensive in execution and also because we did not try to optimize on the implementation of the algorithm. One could, however, come up with an implementation for which the actual running times are of the order of close powers, to the base 2, of the quantities we've tabulated above.

It should be kept in mind that the values tabulated for a particular choice of parameters do not represent the average performance of the function for that choice. They just represent the performance of the instance of the function we used. Of course, for the case of Alon's expanders there can be only one instance of the function with the same choice of parameters but this does not hold for the random expanders.

4.3.5 Dependance of the attack on the value of d

As one would expect, the performance of *Invert_* f_{ip} deteriorates with increase in the value of d used by the function. With larger values of d, we would expect fewer candidates in every stage of the attack to be such that for a sufficient number of sets in the collection, C, we can apply either case 1, 2 or 3 of Section 4.3.1. To put it in another way, our function becomes more secure against this attack when the value of d is increased and thus, we could (rather should) use larger values for d when using $\pi_{ip}$to trade off the insecurity introduced by it. Increasing the value of d does not affect the complexity of the function (in terms of the substance of information hard-wired into it) since using π_{ip}requires no lookup table to be maintained inside the function. In the case of a random predicate we are forced to maintain a lookup table and the size of this table is exponential in the value of d used. Thus, for a random predicate, we would prefer to keep d small (of the order of $\log n$) so that the internally maintained information is of space complexity $O(n)$.

The following table illustrates results of a few tests carried out to distinguish between the cases of $d = 8$ and $d = 12$. We have also tabulated results obtained for the same cases (i.e. the same choice of parameters) when working with a random predicate (π_{random}) and using *Invert_in_good_order* as the inversion algorithm. The purpose is to illustrate the tradoff between the insecurity introduced by $\pi_{ip}$and the complexity introduced by a random predicate. As before, the entries are running times in terms of the maximum list length and the values enterred in every row correspond to the median observation with five different strings for the same choice of parameters. $t_{\pi_{random}}$ denotes the running time of *Invert_in_good_order* on the π_{random}-version of the function and $t_{\pi_{ip}}$ denotes the running time of *Invert_* f_{ip} on the π_{ip}-version.

Value of n	$t_{\pi_{random}}$		$t_{\pi_{ip}}$	
	$d = 8$	$d = 12$	$d = 8$	$d = 12$
25	12.98	17.01	7.71	10.61
31	15.06	19.03	11.59	14.06
40	18.19	23.03	14.46	17.28

Points worth observing are

- The increase in value of d improves the security of the π_{random}-version of the function much more than of the π_{ip}-version (the gap between the running times for $d = 8$ and $d = 12$ is much less for the π_{ip}-version than for the π_{random}-version).

- The running times for $d = 8$ in the π_{random}-version are close to the corresponding running times for $d = 12$ (in the π_{ip}-version). Thus, by increasing d for the π_{ip}-version, we could reach the security attained by lower values of d in the π_{random}-version.

4.3.6 An alternate implementation

We have presented *Invert_* f_{ip} in a form that closely resembles *Invert_in_good_order* for the sake of consistency. In general, there can be many more tricks one could try using the ideas presented in 4.3.1. One specific variation would be to begin by guessing a particular number, say n/k, of bits in the input string and then invoking *discover* on the resulting string. In case we still end up with some undiscovered bits, we continue (a) by guessing more bits *or* (b) by reverting to examining bits in the output string and following the course of *Invert_* f_{ip} after that. We carry this on till the stage where we know the input completely. The time complexity of such an attack would depend largely on the number of bits guessed initially (if k is small enough) and would be lower bounded by $O(2^{n/k})$ since we would need to try all possible guesses on the input bits previously guessed.

We tried this version of the attack (for subversions (a) and (b) and $k = 3, 4$) but it didn't provide us any improvement over *Invert_*f_{ip}. The running time of both versions were roughly the same in almost all the cases (that is, while making comparisons with a fixed output string and a fixed collection) and in most cases *Invert_*f_{ip} worked better than the current version.

4.3.7 Candidates for improvement on *Invert_*f_{ip}

In section 4.2, we gave an intuition for why π_{ip}could be weaker than any random predicate and that it is *almost linear* in some sense. Based upon this intuition one improvement on *Invert_*f_{ip} that appears natural is to check every candidate input for not only the possibility of *any one set* causing an easy-to-capitalize-upon situation with it (i.e. giving us either of cases 1, 2 or 3 spoken of before) but also for the possibility of *a set of sets* causing some other similarly-easy-to-capitalize-upon situation. What we mean here is that for every candidate input we could check (after invoking *discover* and discovering as many undiscovered bits as possible) if the set of equations (each equation being of the form $e_i = y_i$) we obtain by applying the predicate for some subset of C and equating these values to the corresponding output bits turns out to be *a linear system of equations* in |Rem| variables, where Rem is the set of undiscovered bits in the candidate being considered. If this linear system has enough number of equations (specifically, if the number of equations is greater than or equal to |Rem| itself) then we have a chance of solving it and obtaining a unique solution (if there exists one) or otherwise declaring the candidate invalid (if the system turns out to be unsolvable). It is of no use to look at a case where the number of equations is less that |Rem| because even if in such a case the system is solvable, it'll have more than one solutions.

In terms of the listing of the algorithm, what we could do is to introduce a new procedure *checklinear*(z, y, C, n, d) which checks if the system of equations obtained by applying the predicate for all the sets in C over the candidate, z, is linear in the |Rem| variables representing the undiscovered bits with the required minimum number of equations and if so returns the unique solution, if there exists one and *null* if there exist none. The procedure would return *nonlinear* otherwise. It would be invoked right at the beginning of the *for* loop at line number $4(b)$ in the listing.

Experiments reveal that this candidate for improving the attack, though pretty attractive, gives us absolutely no benifit with the reason being that at no stage do we get a candidate for which the set of equations (spoken of above) is linear *and* has size greater than or equal to |Rem|. Furthermore, the number of such candidates is too small to be of any great use. We experimented with various strings and different parameters to the function but in all the cases we saw exactly the same behaviour. Intuitively, we could explain this as occurring because we don't get too many linear equations, *each being linear in more than one variable* resulting from this procedure. The clause *"being linear in more than one variable"* is important - when looking for a linear system we will be able to find only such equations. Any equation linear in only one variable is equivalent to case 2 of section 4.3.1 and, thus, would be taken care of by invoking *discover* itself. It is in keeping with all this that we do not illustrate results with this inclusion.

In general, one could come up with more sophisticated ideas to take advantage of the partial linearity of π_{ip}(in the intuitive sense as we've spoken of it before) but we feel that they can be defeated by increasing the value for d substantially.

5 Conclusion and Acknowledgements

The experimental analysis of the function seems to suggest that it is indeed a good candidate for a one-way function. The fact that it appears to become more vulnerable to attacks for one particular version does not make us loose hope in it's utility because for the more generalise description (that using the random predicate and a lookup table) there seems to be no efficient way to invert it. We stress that besides using a good collection (i.e. one with a sufficiently large expansion) it is also important to make sure that the predicate is not too weakly designed. In particular, the more unrelated the output of the predicate is to it's input bits, the better it is, for we could (in a few cases, like the π_{ip}-version) capitalize upon such relations and improve the original attack. An absolutely random predicate seems to be the best choice.

We are grateful to Oded Goldreich for useful guidance and fruitful discussions offered during this work. We would also like to thank Adi Shamir for suggestions and pointers on the cryptanalysis of the function.

References

[1] O. Goldreich. Candidate One-Way Functions Based on Expander Graphs.

[2] N. Alon. Eigenvalues, Geometric Expanders, Sorting in Rounds, and Ramsey Theory. *Combinatorica*, Vol. 6, pages 207–219, 1986.

[3] N. Alon and V.D. Milman. λ_1, Isoperimetric Inequalities for Graphs and Superconcentrators, *J. Combinatorial Theory, Ser. B*, Vol. 38, pages 73–88, 1985.