A Practical Polynomial-Time Known-Plaintext Attack on a Cryptosystem Proposed by John Nash

Adi Shamir and Eldad Zinger
Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science

June 2012

Abstract

In this paper we present a known plaintext attack on a stream cipher proposed by John Nash in 1955, which reduces the claimed security of the scheme from $(n! \cdot 2^n)^2$ to $n^2 \log n$. This attack was verified with an actual simulation, finding a cryptographic key of 3800 bits in just a few minutes on a single PC.

1 Introduction

In 1955 John Nash wrote a series of letters to the National Security Agency (NSA) in which he described a new design for a stream cipher [1]. His design was not adopted by NSA but both the letters and the evaluation of the scheme by the NSA were kept classified. In 2012, NSA declassified and published the correspondence to the public [2], but did not reveal any details about its internal evaluation process. Naturally, this led to considerable speculation about the actual security of this scheme.

In his letters, Nash suggested that the security of the scheme is equivalent to the number of keys, which is $(n! \cdot 2^n)^2$. In this paper, we will first show that the effective number of keys can be easily reduced from $(n! \cdot 2^n)^2$ to $n! \cdot 2^{n+1}$. We will then describe a subtle weakness in the design, which requires the analysis of patterns of logarithmic number of consecutive bits in the data. This weakness is, briefly, the result of the interaction of two random permutations, in which for any pair $i \in [n]$ and $j \in \{2, \ldots n, \mathit{output}\}$ there exists a sequence of about $\log n$ invocations of those permutations which maps position i to position j. By exploiting this weakness, we will provide a known plaintext attack with time and data complexities of $O\left(n^2 \log n\right)$. We will conclude with simulation results of our attack for various key sizes.

A different attack on the Nash cipher, which has similar time complexity but requires chosen messages, was independently developed by Ron Rivest and his students (private communication).

2 The stream cipher

Nash introduced a stream cipher which is based on a permuter, which is a state that evolves by applying permutations on it. This technique used to be very common among cryptographers in the 40s and the 50s and was used in the design of the Enigma machine. Unlike the Enigma machine, Nash's cryptosystem design is fully electronic rather than electro-mechanical.

In his letter, Nash described a permuter that uses two single-cycle permutations $\left(P^0, P^1\right)$ over a state with n bits and two n bit vectors $\left(K^0 K^1\right)$. The permuter is controlled by a decider D. In each cycle, the permuter uses the bit in the decider to choose P^D and K^D. The permuter uses P^D to transfer all state bits, including the decider bit, across the state. When a bit is transferred, it might be flipped according to K^D.

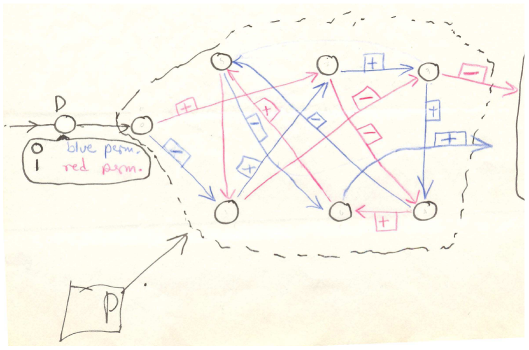

Figure 1:

At the beginning of each encryption step, the decider holds the previous generated ciphertext bit. In order to encrypt a plaintext bit, the permuter transfers and flips bits as described above and outputs a single bit. This bit is added to the plaintext bit (mod 2) and the result is the ciphertext bit. The ciphertext bit is transmitted, but also copied back into the decider bit.

The full operation of the permuter is formally described as follows:

Let $P^D(1)^i$ be the position of a bit that was in position 1 and traveled i jumps along the Hamiltonian path of permutation P^D. Notice that it must be that $P^D(1)^n = output$. Let K_i^D correspond to the i'th jump of the Hamiltonian path of P^D.

To be clear, given D, K_1^D is the bit that defines the flip of b_1 when moved to its new position by the permutation P^D and K_n^D is the bit that defines the flip of $b_{P^D(1)^{n-1}}$ when it is outputted from the permuter. The operations done in a single cycle are as follows:

$$output \leftarrow b_{P^D(1)^{n-1}} \oplus K_n^D$$

$$c \leftarrow p \oplus output$$

$$\forall i \in [n-1] \quad b_{P^D(1)^i} \leftarrow b_{P^D(1)^{i-1}} \oplus K_i^D$$

$$b_1 \leftarrow D$$

$$D \leftarrow c$$

An example, taken from Nash's original letter to NSA, is given in Fig 1.

The secret key is the tuple $\left(P^0, P^1, K^0, K^1\right)$. The decryption process is based on the same permuter with the same key since the permuter uses only the previous ciphertext bit which is available to the receiver, so the receiver's state will evolve in the same way as the sender's state.

The conjectured cost of breaking the stream cipher is the cost of going over all possible keys and checking each key by decrypting the given ciphertext and comparing to the known plaintext. Ignoring the cost of decrypting and comparing the data, the cost is proportional to the number of keys:

$$(n! \cdot 2^n)^2$$

3 Reducing the effective number of keys to $n! \cdot 2^{n+1}$

In the following subsections, we will show that the number of effective keys can be easily reduced to $n! \cdot 2^{n+1}$. We will prove it in two steps.

In the first step we will show that the effect of the two secret single-cycle permutations P^0, P^1 can be obtained by one arbitrary known permutation and one secret permutation, regardless of K^0, K^1. This step will reduce the number of keys to $n! \cdot 2^{2n}$.

In the second step we will show that the effect of the two secret n-bits vectors K^0, K^1 can be obtained by just $n+1$ secret bits. This step will reduce the effective number of keys to $n! \cdot 2^{n+1}$.

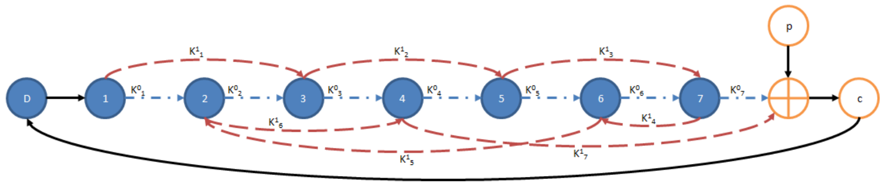

Figure 2:

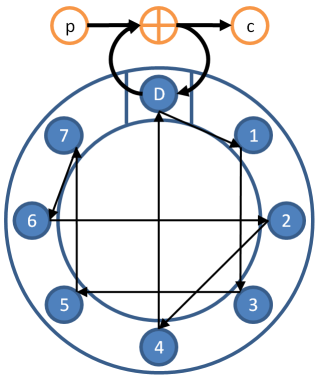

Figure 3:

3.1 A single secret permutation is sufficient to mimic the effect of two secret permutations

Our first observation is that the effect of any two secret permutations $\left(P^0, P^1\right)$ can be reduced to the effect of one arbitrary known permutation and one secret permutation. Generally, by labeling the positions in the state according to the order derived from P^0, for example, we will get that P^0 is the trivial permutation of shifting: $P^0\left(i\right) = i + 1$ (for $i \in [n-1]$) and $P^0\left(n\right) = \mathit{output}$. The second permutation will hold the composition of the original secret permutations.

This relabeling is formally explained by the following transformation. Let $i_{old} \in [n]$ be the label that a certain position has before the transformation. Let $i_{new} \in [n]$ be the label that the same position has after the transformation. Let $\left(P^0_{old}, P^1_{old}\right)$ and $\left(P^0_{new}, P^1_{new}\right)$ be the representation of the pair of permutations before/after the transformation. Let $revP^0_{old}\left(i_{old}\right)$ be the number of jumps along the Hamiltonian path of P^0_{old} needed in order to reach position i_{old}.

$$i_{old} = P^0_{old}\left(1\right)^{i_{new}-1}$$

$$i_{new} = revP^0_{old}\left(i_{old}\right) + 1$$

$$P^0_{new}\left(i_{new}\right) = revP^0_{old}\left(P^0_{old}\left(P^0_{old}\left(1\right)^{i_{new}-1}\right)\right) + 1 = i_{new} + 1$$

$$P^1_{new}\left(i_{new}\right) = revP^0_{old}\left(P^1_{old}\left(P^0_{old}\left(1\right)^{i_{new}-1}\right)\right) + 1$$

Since the transformation is only symbolic, the new key $\left(P^0_{new}, P^1_{new}\right)$ will be indistinguishable from the original key $\left(P^0_{old}, P^1_{old}\right)$. Notice that only P^1_{new} is secret. From now on, we will replace P^0, P^1 by P^0_{new}, P^1_{new} and all the described positions will be transformed accordingly.

As an example of what a permuter might look like after such a transformation, figure 2 describes the same permuter that Nash described in his original letter. Figure 3 is a more intuitive representation of the same permuter, where the trivial P^0 is a simple clockwise rotation of the ring and P^1 is described by the arrows inside the ring.

3.2 $n+1$ secret flipping bits are sufficient to mimic the effect of $2n$ secret flipping bits

First, we will notice that a distinguisher can look only at output bits. Hence, we need to mimic the behavior of the original secret $2n$ flipping bits only on full paths, which means paths that an inserted bit, b, traveled inside the permuter and was outputted from the permuter as bit o. These paths are defined by the sequence of bits inserted to the permuter after b was inserted and until b was outputted. Let this sequence of input bits be $x_1, \ldots, x_m$ where x_1 is the bit that was inserted right after b was inserted, and the bit x_m is the bit that was inserted and caused the extraction of o, therefore $P^{x_m}(\cdots(P^{x_1}(1))) = output$. Let a path parity behavior a be the accumulated flips that an inserted bit b will be subjected to until b is outputted as o.

$$a = K_1^{x_1} \oplus \cdots \oplus K^{x_i}_{revP^{x_i}(P^{x_{i-1}}(\cdots(P^{x_1}(1))))+1} \oplus \cdots \oplus K_n^{x_m}$$

$$o = b \oplus a$$

As b and o are known, this is a linear equation with up to $2n$ variables.

Let π^i be $P^1(1)^i$. Define the following $n+1$ variables:

$$z_0 = \bigoplus_{j=1}^{n} K_j^0$$

$$\forall i \in [n] \quad z_i = K_i^1 \oplus \bigoplus_{j=\min\{\pi^{i-1},\pi^i\}}^{\max\{\pi^{i-1},\pi^i\}-1} K_j^0$$

The first variable, z_0, is the parity of K^0. Any other variable z_i is a sum of one unique bit from K^1 added with the sum of consecutive bits from K^0 that are needed by the trivial permutation P^0 to complete the same jump that K_i^1 corresponds to in P^1 or the opposite jump if the i'th jump of P^1 is to a smaller position.

In order to write the above linear equation as an equation of only up to $n+1$ variables, we will use an iterative process in which we will hold a permuter path and in each step we will change this path until the path matches the path that is defined by $x_1, \ldots, x_m$.

Formally, we would like that for every m, $x_1, \ldots, x_m$ find a sequence $y_1, \ldots, y_{m'} \in [n] \cup \{0\}$, $m' \leq m+1$, such that

$$a = \bigoplus_{i=1}^{m'} z_{y_i}$$

The iterative process will start with $a' = z_0$. The iterations are as follows: for every $i \in [m]$, if $x_i = 1$ then add $z_{revP^{x_i}(P^{x_{i-1}}(\cdots(P^{x_1}(1))))+1}$ to a'.

Following is a proof for the correctness of the above transformation. It is advised to follow the example in the appendix.

Claim: For any $i \in [m]$, before the i'th iteration, a' is the path parity behavior of the required path up to position $P^{x_{i-1}}(\cdots(P^{x_1}(1)))$ added with the path parity behavior of the Hamiltonian path of P^0 from position $P^{x_{i-1}}(\cdots(P^{x_1}(1)))$ to the output. Notice that from this claim, before each iteration, a' represents a full path.

Proof: By induction on i, the iteration index. The base case is the initial value $a' = z_0$ and the represented path is the Hamiltonian path of the trivial permutation P^0, n zeros. Assuming that for iteration $i-1$ the claim holds, lets prove for iteration i. At the beginning of the iteration, the current path suggested by a' goes from position 1 to position $P^{x_{i-1}}(\cdots(P^{x_1}(1)))$ with the same path parity behavior as required and then the path continues to the output by $n+1-P^{x_{i-1}}\left(\cdots\left(P^{x_1}(1)\right)\right)$ steps of P^0.

If $x_i = 0$, a' will stay the same. In this case a' will be considered as the representation of the path from position 1 to position $P^0\left(P^{x_{i-1}}\left(\cdots\left(P^{x_1}(1)\right)\right)\right)$ followed by $n+1-P^0\left(P^{x_{i-1}}\left(\cdots\left(P^{x_1}(1)\right)\right)\right)$ steps of P^0. Trivially, the claim holds.

Consider the case of $x_i = 1$. Notice that the current represented path from position $P^{x_{i-1}}\left(\cdots\left(P^{x_1}(1)\right)\right)$ to position $P^1\left(P^{x_{i-1}}\left(\cdots\left(P^{x_1}(1)\right)\right)\right)$ is by steps of P^0. These steps need to be replaced with one step of P^1. Let q be the number of jumps in the Hamiltonian path of P^1 that are needed to reach position $P^{x_{i-1}}\left(\cdots\left(P^{x_1}(1)\right)\right)$ from position 1, $q = revP^1\left(P^{x_{i-1}}\left(\cdots\left(P^{x_1}(1)\right)\right)\right)$. The contribution of the required jump of P^1 to the path parity is K^1_{q+1} and the contribution of the redundant steps of P^0 is $\bigoplus_{j=\min\{\pi^q,\pi^{q+1}\}}^{\max\left\{\pi^q,\pi^{q+1}\right\}-1} K^0_j$. In order to adjust the path parity behavior, canceling the contributed behavior of the steps of P^0 is made by subtracting the contribution of the redundant steps of P^0 from a' and adding the contribution of the required jump of P^1. Subtraction and addition mod 2 are the same, so:

$$a' \leftarrow a' \oplus K^1_{q+1} \oplus \bigoplus_{j=\min\{\pi^q,\pi^{q+1}\}}^{\max\left\{\pi^q,\pi^{q+1}\right\}-1} K^0_j$$

Notice that $K^1_{q+1} \oplus \bigoplus_{j=\min\{\pi^q,\pi^{q+1}\}}^{\max\left\{\pi^q,\pi^{q+1}\right\}-1} K^0_j = z_{q+1} = z_{revP^1\left(P^{x_{i-1}}\left(\cdots\left(P^{x_1}(1)\right)\right)\right)+1}$. The claim follows.

Claim: After the last step, a' represents the same path as described by $x_1, \ldots, x_m$, so $a' = a$.

Proof: By the previous claim, after the last iteration, meaning before iteration $m+1$, a' is the path parity behavior of the required path up to position $P^{x_m}\left(\cdots\left(P^{x_1}(1)\right)\right) = output$ added with the path parity behavior of the Hamiltonian path of P^0 from position $P^{x_m}\left(\cdots\left(P^{x_1}(1)\right)\right)$ to the output, 0 steps. The claim follows.

Consequences: a' is added with variables from $\{z_i\}_{i\in\{0,\ldots,n\}}$ until $a' = a$. Therefore, every path parity equation can be regarded as an equation over just $n+1$ variables. If we know the secret permutation P^1, then we will need to find $n+1$ independent linear equations which are different permuter paths along with a sampled bit $o \oplus b$. Solving these linear equations will give us the missing variables.

Since any distinguisher can test only the paths parity behavior, and the set of $n+1$ variables $\{z_i\}$ mimics this behavior perfectly, we can define any transformation between the original key to the reduced key arbitrarily as long as the values of $\{z_i\}$ are the same. As an example, such transformation can be:

$$\forall i \in [n-1] \quad newK^0_i = 0$$

$$newK^0_n = \bigoplus_{i=1}^{n} oldK^0_i$$

$$\forall i \in [n] \quad newK^1_i = oldK^1_i \oplus \bigoplus_{j=\min\{\pi^{i-1},\pi^i\}}^{\max\left\{\pi^{i-1},\pi^i\right\}-1} oldK^0_j$$

$$z_0 = newK^0_n$$

$$\forall i \in [n] \quad z_i = newK^1_i$$

4 A known plaintext attack with time and data complexities of $O\left(n^2 \log n\right)$

This attack will focus on finding the secret random permutation P^1 while K^0, K^1 are unknown. After this attack, a set of linear equations can be formulated and solved to find K^0, K^1. Each such linear equation is a different path in the permuter. This path is the result of the interaction between the random permutation and the trivial permutation. Focusing only on short paths, each such equation will hold up to $2\log n$ variables, as will be explained in the following subsection. Ignoring logarithmic factors, the problem of solving a sparse system of linear equations over GF(2) is proportional to the product of the number of variables, the number of equations, and the number of ones per row. Since in our case this product is $n^2 \log n$, the additive complexity of solving the final system of linear equations is comparable to that of applying the actual attack, and thus we will ignore it in the rest of our analysis.

This attack will assume two independent distributions:

1. Uniform key distribution. P^1 is a random permutation (Hamiltonian path) and so are K^0, K^1.
2. Uniform plaintext distribution. The known ciphertext is generated out of a uniform plaintext. Notice that since each ciphertext bit is equal to $p_i \oplus output = c_i$, and p_i is a random (uniformly chosen) bit, c_i is distributed uniformly.

The next two subsections explain the main two ideas that this attack uses.

4.1 The length of the shortest path is $\leq \log n$ with high probability

Lets consider len, the random variable of the length of the shortest path from position 1 to the output. By considering the interaction of a random permutation with an arbitrary permutation, we would like to upper bound this random variable by $\log n$ w.h.p.

One intuitive argument is to consider all the strings of length up to $\log n$ bits. About $2n$ random positions are sampled out of n available positions $\{2, \ldots, n, output\}$. Therefore, w.h.p there is a string of length up to $\log n$ bits that reaches the output, as was verified in simulations.

Another argument can be made by using the birthday paradox. In order to reach the output after $\log n$ steps, one of the $\frac{n}{2}$ positions reached after $\log n - 1$ steps must be either position n or position $P^1\left(output\right)^{-1}$. This is possible if and only if one of the $\frac{n}{4}$ positions reached after $\log n - 2$ steps is in $\left\{n-1, P^1\left(output\right)^{-1} - 1, P^1\left(n\right)^{-1}, P^1\left(output\right)^{-2}\right\}$. Repeating this argument $\frac{1}{2}\log n$ steps we will have a set S_{needed} of $\sqrt{n}$ positions that are needed in order to ensure that after $\frac{1}{2}\log n$ steps the position will be the output. Notice that S_{needed} might hold less than $\sqrt{n}$ positions due to duplicates, but by the uniformity of P^1 and the birthday paradox, the set will hold $O\left(\sqrt{n}\right)$ distinct random positions. On the other hand, the set $S_{available}$ of values that are available after $\frac{1}{2}\log n$ steps from position 1 has $\sqrt{n}$ random positions. With the same argument as before, $S_{available}$ holds $O\left(\sqrt{n}\right)$ distinct random positions. Arguing for the birthday paradox again, w.h.p there is a position in $S_{needed} \cap S_{available}$ so w.h.p some string of length $\log n$ bits is a full path.

A similar argument can be used to claim the same for the length of the shortest path from position 1 to any position and for the length of the shortest path from any position to the output.

4.2 A path is valid if it is $c\log n$-positive

We would like to be able to answer queries of the form "is the following string of up to $2\log n + c$ bits a valid path from the first position to the output of the permuter?". A suggested path is k-positive if k different instances were found in the known ciphertext (including overlaps) and all instances apply the same path parity behavior a. That is, when a bit b is inserted just before the bits representing a path are inserted, the output is always $b \oplus a$ for any b. Larger k means that the suggested path is valid with higher probability. By setting the required known ciphertext length to be $k \cdot 2^{2\log n + c}$, the expected number of instances of an arbitrary string of length $2\log n + c$ bits it at least k.

The validity of a path depends only on the random permutation P^1. Taking the probability over the key distribution, a suggested path is valid with probability about $\frac{1}{n}$ given that no prefix of it is a valid path. Note that the answer of a query depends on the random key but also on the random data (the given plaintext/ciphertext).

A valid path with k instances will always be considered k-positive. A non valid path with k instances will be considered k-positive with probability about 2^{-k+1}. Such an event occurs when for each instance, the output bit o is correlated to the input bit b. Since it is given that the path is not valid, it means that o is not related to b, but is the result of the travel of another input bit b'. The first instance defines the correlation a so the probability is only over the other $k-1$ instances.

We will answer the validity of a suggested path according to the grade the path received: negative or k-positive. Unfortunately, the answers may be erroneous, and we have to consider the error probability.

Given that a query answer is negative, the path is definitely non valid so the query can not err.

Given that a query answer is k-positive, the probability that the path is actually non-valid and thus the query errs is as following:

$$\Pr[\text{non-valid} \mid k\text{-positive}] \approx 2^{-(k-1-\log n)}$$

Taking $k = c\log n$ will result in an error probability of $\frac{1}{poly(n)}$. Since the total number of such queries is n^2, with a good choice of k, no query is expected to err.

4.3 The Actual Attack

We will divide the attack into 3 phases: data processing, path-suffixes mapping, and revealing P^1.

In the first phase we will process the given data. In order to answer queries for the validity of suggested paths quickly, we would use a suffix tree over the known ciphertext bits. Any such query will travel along the suffix tree twice. Let $b \in \{0,1\}$ be the tested input bit, $\langle path\rangle$ be the bits representing the suggested path and o_b the output bit that was outputted after the insertion of $b \circ \langle path\rangle$. A query will check for the consistency of $b \oplus o_b$. The first tree travel will find all the suffixes that start with $0 \circ \langle path\rangle$. For each suffix calculate the output bit. If not all the output bits are the same, return false. Else, remember this output bit as o_0. The second tree travel is similar but will consider all the suffixes that start with $1 \circ \langle path\rangle$ and if all the output bits are the same, remember the output bit as o_1. Return the boolean expression $0 \oplus o_0 = 1 \oplus o_1$.

In the second phase we will build a dictionary of n path-suffixes, where the length of each path-suffix is about $\log n$ bits. We will start by finding the shortest path-suffix $A_{\{0\}^1}$ such that $0 \circ A_{\{0\}^1}$ is a path. The method for finding this path checks for $A_{\{0\}^1}$ all strings in lexicographic order until a positive path is found.

Having $A_{\{0\}^1}$ with expected length $\log n$ bits, store it in a dictionary with $A_{\{0\}^1}$ as the key and 1 as the data.

Do the same for any path-prefix of i zeros, $i \in [n-1]$. The data for each new path-suffix $A_{\{0\}^i}$ is i. For every $i \in [n-1]$, after finding $A_{\{0\}^i}$, find the shortest path-prefix R such that $R \circ A_{\{0\}^i}$ is a valid path. In the next iteration, use R to find $A_{\{0\}^{i+1}}$ for the path-prefix $R \circ 0$. This compression technique makes sure that the queries are always up to $2\log n + c$ bits long while we keep track of the starting positions of the path-suffixes.

After the second phase, a dictionary with path-suffixes from any position in $\{2,\ldots,n\}$ is available.

The third phase is very similar to the second phase but instead of path-prefixes of zeros, run path-prefixes of ones. Instead of storing $A_{\{1\}^i}$, find it in the dictionary. Since any jump but the last one has an end-index in $\{2,\ldots,n\}$, the order of the generated path-suffixes guarantees that we find it as a key in the dictionary. For each $j \in [n-1]$, we find $i \in [n-1]$ such that $A_{\{1\}^j} = A_{\{0\}^i}$ and it means that the j'th jump of P^1 reaches position $i+1$. Having the end-indices of all the jumps in P^1, we revealed P^1.

Complexities The length of each path is at most $2\log n + c$ and we would like that each string of $2\log n + c$ bits will have at least $k = c\log n$ instances for low error probability. Therefore, we need $O\left(n^2 \log n\right)$ bits of data.

Building the suffix tree is done in linear time in the size of the data, $O\left(n^2 \log n\right)$. Each query to the suffix tree costs $O(\log n)$ since the query length is $O(\log n)$ and the number of leaves to consider for each query is not more than $O(\log n)$ leaves. Even if more leaves are available, there is no need to consider them.

For $i \in [n-1]$, finding $A_{\{0\}^i}, A_{\{1\}^i}, R$ costs n suffix tree queries, so total n^2 suffix tree queries are processed. The dictionary can be implemented with a simple table with constant time insertion and look up. The total cost of the attack is $O\left(n^2 \log n\right)$.
The following table describes the complexity of the attack for various key sizes (without actually solving the resultant sparse system of linear equations). The first column is the size of the permuter. The second column is the key size $\lceil \log(n!) \rceil$ in bits, without the additional $n+1$ key-bits needed to define K^0, K^1. The third column is the attack time in seconds on a single PC. For each key size, we repeated the attack 100 times and did not encounter any failures.

n	key size	attack time [sec]
2^4	45	0.115
2^5	118	0.7
2^6	296	4.85
2^7	717	24.7
2^8	1684	160
2^9	3876	917
2^{10}	8770	5125

References

[1] John Forbes Nash Jr. Cryptosystem proposal letters. http://www.nsa.gov/public_info/_files/nash_letters/nash_letters1.pdf, January 1955.

[2] NSA Press Release. National Cryptologic Museum Opens New Exhibit on Dr. John Nash. http://www.nsa.gov/public_info/press_room/2012/nash_exhibit.shtml, January 2012.

A An example of secret bits reduction

We will use the permuter described in figures 2 and 3 to give an example of the process described in section 3.2. The bits $z_0, \ldots, z_7$ are as following:

bit	mark
$\bigoplus_{j=1}^{7} K_j^0$	z_0
$K_1^1 \oplus \bigoplus_{j=1}^{2} K_j^0$	z_1
$K_2^1 \oplus \bigoplus_{j=3}^{4} K_j^0$	z_2
$K_3^1 \oplus \bigoplus_{j=5}^{6} K_j^0$	z_3
$K_4^1 \oplus K_6^0$	z_4
$K_5^1 \oplus \bigoplus_{j=2}^{5} K_j^0$	z_5
$K_6^1 \oplus \bigoplus_{j=2}^{3} K_j^0$	z_6
$K_7^1 \oplus \bigoplus_{j=4}^{7} K_j^0$	z_7

Consider an output bit o. It started as a known bit b, then went through a series of XORs with bits from K^0 and K^1 and eventually was outputted. Assume that the permutations are known. Lets say that the output equation is:

$$o = b \oplus K_1^1 \oplus K_3^0 \oplus K_4^0 \oplus K_5^0 \oplus K_6^0 \oplus K_4^1 \oplus K_5^1 \oplus K_2^0 \oplus K_3^0 \oplus K_7^1$$

Now we will show how to express $o \oplus b$ as a function of the above $n + 1 = 8$ bits $\{z_i\}_{i=0}^7$.

Start with the expression $a' = z_0$. We would like that the expression will represent the path that the bit b traveled in the permuter. Currently, it looks like if the bit started from the first position and moved right all the way with P^0 until the output. This is not true and we will fix it. The first move was the first jump of P^1. This means that b jumped from position 1 to position 3 by P^1. Therefore, we need to replace the current path from position 1 to position 3 by a single jump of P^1. We will replace $K_1^0 \oplus K_2^0$ with K_1^1. Adding z_1 to a' will do that.

The expression now is $a' = z_0 \oplus z_1$. The second jump is from position 3 to position 4 by a step of P^0. This step is already accounted for in a' so there is no need to update a'. The same goes for the third, forth and fifth jumps because they are by steps of P^0 from position 4 to position 7.

The current path represented in a' is from position 1 to position 3 by a single jump of P^1 and from position 3 to the output by 5 steps of P^0.

The sixth jump is by P^1 from position 7 to position 6. Therefore, we need to add to the represented path a jump backward from position 7 to position 6 by P^1 and then a step forward from position 6 to position 7 by P^0. It is intuitive to consider this change as if we cut the represented path in position 7 and insert a new subpath $7 \to 6 \to 7$. The change is made by adding z_4 to a'.

The expression now is $a' = z_0 \oplus z_1 \oplus z_4$ and it matches the required path only up to the seventh jump.

The seventh jump is from position 6 to position 2 by a jump of P^1. As before, we will cut the path in position 6 and insert the subpath $6 \to 2 \to 3 \to 4 \to 5 \to 6$. The change is made by adding z_5 to a'.

The expression now is $a' = z_0 \oplus z_1 \oplus z_4 \oplus z_5$ and it matches the required path only up to the eighth jump.

The eighth and ninth jumps are from position 2 to position 4 by steps of P^0 so no need to change a'.

The tenth jump is from position 4 to the output by P^1 so we need to replace the represented subpath $4 \to 5 \to 6 \to 7 \to$ *output* with a single jump of P^1 $4 \to$ *output*. This is done by adding z_7 to a'.

We iterated over all the jumps from position 1 to the output and our expression equals to the same path parity behavior as the original expression.

$$o \oplus b = z_0 \oplus z_1 \oplus z_4 \oplus z_5 \oplus z_7$$