Practical Polynomial Time Known Plaintext Attacks on a Stream Cipher Proposed by John Nash

Adi Shamir and Eldad Zinger

Faculty of Mathematics and Computer Science
Weizmann Institute of Science
{adi.shamir,eldad.zinger}@weizmann.ac.il

Abstract. In this paper we present two known plaintext attacks on a stream cipher which was developed by John Nash in the early 1950's but whose design was declassified by the NSA only in 2012. The main attack reduces the claimed security of the scheme from $((n-1)! \cdot 2^n)^2$ to $O\left(n^2 \log^3 n\right)$, where n is a security parameter. This attack succeeds with high probability for randomly chosen keys even when the only thing we know about the plaintext is that a small fraction of isolated plaintext bits are slightly biased, but always fails for a certain well defined class of keys which is exponentially large but a negligibly small fraction of all the possible keys. To show that it would not suffice to simply restrict the choice of keys to this class, we develop a different attack which works best for the subset of keys which are hardest to find by the first attack. This attack reduces the security of the scheme from $2^{O(n)}$ to $O\left(n^2\right)$. Both attacks were verified with actual simulations, finding cryptographic keys which are thousands of bits long in just a few minutes on a single PC.

Keywords: Cryptanalysis, Stream cipher, Permutation, John Nash.

1 Introduction

John Nash is known today mostly for his seminal contributions to the field of Game Theory and as the winner of the 1994 Nobel Memorial Prize in Economic Sciences. What was not known for many years is that in the early 1950's he 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 the NSA but both the letters and their analysis by the NSA were kept classified. In 2012, the NSA declassified and published the correspondence [3], but did not reveal any details about its internal evaluation process. Naturally, this led to considerable speculation about the actual security of this scheme.

Nash's letters have a great historical significance, and here is what Noam Nisan wrote about them in February 2012 in his blog [2]:

"Nash goes on to put forward an amazingly prescient analysis anticipating computational complexity theory as well as modern cryptography. In the letter, Nash takes a step beyond Shannon's information-theoretic formalization of cryptography (without mentioning it) and proposes that security of encryption be based on computational hardness — this is exactly the transformation to modern cryptography made two decades later by the rest of the world (at least publicly...). He then goes on to explicitly focus on the distinction between polynomial time and exponential time computation, a crucial distinction which is the basis of computational complexity theory, but made only about a decade later by the rest of the world. He conjectures that almost all cipher functions (from some — not totally clear — class) are one-way. He is very well aware of the importance of this "conjecture" and that it implies an end to the game played between code-designers and code-breakers throughout history. Indeed, this is exactly the point of view of modern cryptography. He is very well aware that this is a conjecture and that he cannot prove it. Surprisingly, for a mathematician, he does not even expect

it to be solved. Even more surprisingly he seems quite comfortable designing his encryption system based on this unproven conjecture. This is quite eerily what modern cryptography does to this day: conjecture that some problem is computationally hard; not expect anyone to prove it; and yet base their cryptography on this unproven assumption. All in all, the letter anticipates computational complexity theory by a decade and modern cryptography by two decades."

In his letters, Nash proposed a particular class of stream ciphers with an unspecified security parameter n, and conjectured that the security of his cryptographic scheme is equivalent to the number of keys, which is $((n-1)! \cdot 2^n)^2$. His design shares many ideas with the earlier generation of Enigma-like designs (and in particular the idea of composing several fixed permutations in many possible ways), but goes beyond them by exploiting the transition from electro-mechanical to electronic implementations in order to greatly expand the number of internal states (from the $26^3 = 17576$ possible settings of three mechanical rotors to the 2^n possible values of a general n-bit register). The design is mathematically clean and intentionally minimalist, using only one register, two permutations, and the ability to move and to complement each state bit in one of two possible ways. It was described in the letter with the small hand drawn figure depicted in Fig. 1, which contained everything one has to know about the scheme.

In this paper, we first show in Section 3 that by using a different but equivalent description of the scheme, we can easily reduce the effective number of keys to $(n-1)! \cdot 2^{n+1}$. We then describe two new types of known plaintext attacks on such schemes, where each attack is applicable to a different key distribution.

In the standard model of uniformly chosen keys, we exploit the fact that when we compose two (secret) random permutations over a domain of n positions in multiple random orders, then for any $i, j \in [n]$[1] there exists with high probability (w.h.p) some (unknown) sequence of about $\log n$ permutation steps which move the bit occupying position i to the new position j. By exploiting this property, we develop in Section 4 a known plaintext attack with time and data complexities of just $O\left(n^2 \log^3 n\right)$. In fact, this is almost a pure known ciphertext attack in the sense that it suffices to know only that an arbitrarily small fraction of (possibly isolated) plaintext bits are slightly biased in order to recover the full key. We simulated the attack for many randomly chosen keys whose length exceeded 10,000 bits, and were always able to find the key within a few minutes on a single PC.

The first attack always fails when the two permutations can move each bit only a limited distance to the left or to the right in the register, since in this case it is impossible to move bits from one end of the register to the other end in a logarithmic number of steps. The hardest case for this type of attack is when in each step one of the permutations is limited to distance 1, and the other permutation is limited to distance 2. There are exponentially many possible choices of such pairs of permutations, and thus one can try to salvage the scheme by restricting the choice of permutations to this particular subset. Our second attack, described in Section 5, exploits a different vulnerability of the scheme in order to show that this choice of keys can also be broken using $O\left(n^2\right)$ time and $O\left(n^{1.5}\right)$ data. We will conclude this part with actual simulation results of our attack for various key sizes.

A different attack on the Nash cipher, which has a similar polynomial time complexity but requires the much stronger attack model of chosen messages, was independently developed by Ron Rivest and his students (private communication [4]). Such attacks are much easier to carry out, since the attacker can use his ability to fill the register at any desired moment with

[1] We will use the notation $[n] = \{1, \ldots, n\}$.

bits of his choice by carefully choosing the next sequence of n plaintext bits. In particular, he can repeatedly reset the register to the same state, and test the effect of any single bit change in it, which makes it very easy to recover the full key after a small number of experiments.

2 The Stream Cipher Proposed by Nash

The following description of Nash's cipher is based on Fig. 1 (copied directly from his letter) which provides a simple example of the scheme. It contains a *permuter*, which is a state consisting of n bits[2], that evolves by repeatedly permuting the order and flipping the value of these bits.

The permuter uses a secret key which consists of two single-cycle permutations $\left(P^0, P^1\right)$ over n positions and two n-bit vectors $\left(K^0, K^1\right)$. For example, in Fig. 1, P^0 is described by the blue arrows and P^1 is described by the red arrows. By labeling all 7 positions clockwise with 1 being the left-most position, P^0 is the permutation $(1, 7, 3, 4, 5, 2, 6)$ and P^1 is the permutation $(1, 3, 5, 6, 2, 7, 4)$. Each permutation consists of a single-cycle if we add an edge from the last position back to position 1 (along which it is XOR'ed with the next plaintext bit).

The permuter is controlled by a *decider*, which is a single bit register that contains the previously generated ciphertext bit. At each step, this bit d is used to choose P^d and K^d among the two possibilities. The permuter uses P^d to move all the state bits according to the corresponding arrows, setting the bit in position 1 to d and using the bit from the last position as the output of the bit generator. This bit is XOR'ed with the next plaintext bit and the result is used both as the ciphertext bit, and as the next value of the decider, as described in Fig. 2. Whenever bit number i is moved to a new position j, it may be flipped according to the i-th bit in K^d.

The decryption process uses the same permuter with the same key since the decider always uses the previous ciphertext bit which is known to both the sender and the receiver. From the point of view of the cryptanalyst, in known plaintext attacks he knows (but cannot influence) which symbolic sequence of operations were applied to the state, while in chosen plaintext attacks he can choose any sequence he wants.

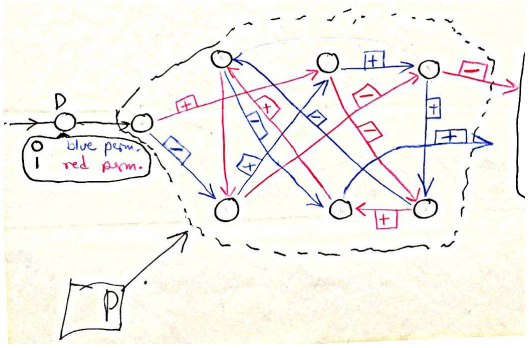

Fig. 1: An example of a permuter (marked with a dashed line) in Nash's own handwriting. The vectors K^0 and K^1 are described as the "+" and "-" markings on the arrows, "-" means that the bit is flipped and "+" means that the bit is not flipped.

[2] Nash originally defined n as the number of positions excluding the first position (which plays a special role in his design), but we prefer to denote by n the total number of positions including this first position.

3 A Simplified Representation of the Scheme

In his letter, Nash conjectured that the cost of breaking his scheme (presumably with a known plaintext attack, even though this is not clearly specified) is the cost of trying all the possible keys, which is $((n-1)! \cdot 2^n)^2$. In this section, we describe a simplified representation of the stream cipher, which uses only $(n-1)! \cdot 2^{n+1}$ possible keys. This is still exponential but only a square root of the conjectured complexity, which trivially disproves Nash's conjecture. In the rest of the paper, we will use this alternative representation to simplify the description of our polynomial time attacks.

Since the labels of the bits in the state could be chosen arbitrarily, we can relabel their positions according to the order of the arrows of P^0, and modify the definition of P^1 as described in the graph G in Fig. 2. The new key will be equivalent to the original key in the sense that it will encrypt the same plaintexts to the same ciphertexts. Since P^0 is now known, this immediately reduces the effective number of keys to $(n-1)! \cdot 2^{2n}$.

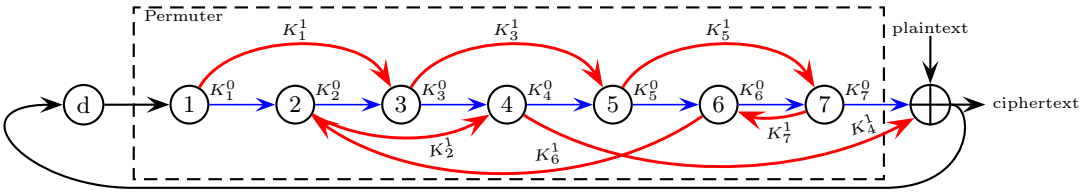

Fig. 2: An example of a graph G arranged by the order of the blue arrows. This is the same permuter as in Fig. 1.

Our next observation is that a permuter affects the encryption/decryption process only through the bits it outputs, and not through the bits it stores. Each such bit (except for the initial $n-1$ bits) was at some stage in position 1, and then a sequence of permutation steps moved it to the output. Each intermediate position $p \in \{2, \ldots, n\}$ in the register has two incoming and two outgoing edges as described in Fig. 3. If we simultaneously flip the 4 key-bits of these edges, then we will store the complement of the original bit, but operate in the same way. By choosing to flip the 4 key-bits in a left to right order according to the key-bit of the incoming blue edge, we can force all the blue key-bits from K^0 except the last one to be zero, and this simplified key will be equivalent to the original key. However, in this representation the register will have to be initialized into a key-dependent value rather than into the all zero value, which can slightly complicate some of our attacks.

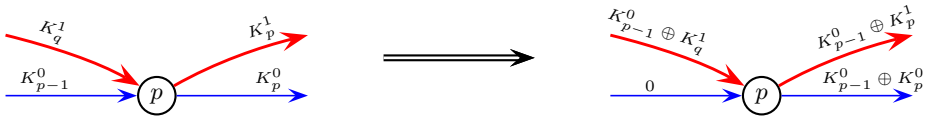

Fig. 3: An illustration of the process of flipping 4 key-bits.

4 A Known Plaintext Attack with Time and Data Complexities of $O\left(n^2 \log^3 n\right)$

For any $i, j \in [n+1]$, let a path from i to j in G be a sequence of edges which connects node i to node j. Since each edge is either a blue or a red arrow, every path can be represented by

a sequence of permutation steps that maps position i to position j. Each permutation step is defined by the value of the decider bit during the specific encryption step, so a path can be represented by a contiguous substring of the ciphertext.

The attack will focus on finding the secret random permutation P^1 while K^0 and K^1 are unknown. After discovering P^1, a set of linear equations in GF(2) can be formulated and solved to find K^0 and K^1. Each such linear equation is derived from a different path in G which is fully identified when P^1 is known. By noticing that a path moves a bit from the input of the permuter to its output and that the input and output values of this bit can be determined from the known data, an equation in the key-bits of K^0 and K^1 that correspond to the arrows along the path can be formulated. Focusing only on short paths of length up to $2\log n + 2\log\log n + c$, each such equation will contain only this many variables, as will be explained in Sect. 4.1. The complexity of solving a sparse system of linear equations over GF(2) is essentially proportional to the product of the number of variables, the number of equations, and the number of ones per row, [6]. Since in our case this product is $O\left(n^2 \log n\right)$, 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 particular attack will make two assumptions about the distribution of the keys and plaintexts. Later we will discuss what happens when these assumptions do not hold.

1. Uniform key distribution: P^1 is chosen uniformly from the set of all single-cycle permutations over n positions. K^0 and K^1 are chosen uniformly from the set of all n-bit vectors. A different known plaintext attack which assumes a different choice of P^1 will be described in Sect. 5.
2. Uniform plaintext distribution: Each plaintext bit p_i is chosen uniformly and independently in $\{0, 1\}$. Notice that since each ciphertext bit c_i is computed as $p_i \oplus \mathit{output}$, and output depends only on previous choices of p_i, c_i (which is used in the decider) is also distributed uniformly. Simulations show that this assumption can be relaxed, as explained in Sect. 4.4 and 4.5.

The next two subsections explain the two main ideas used in this attack.

4.1 The Length of the Shortest Input to Output Path in G Is at Most $\log n$ Bits w.h.p for a Random P^1

Each path in G is defined by a sequence of invocations of P^0 and P^1 such that a bit from position 1 is transferred (with possible complementations) to the output by the corresponding sequence of blue and red arrows. Such a path is uniquely represented by a sequence of decider bits $x_1, \ldots, x_m$. Notice that this sequence appears as a contiguous substring in the ciphertext, and the previous bit in the ciphertext determines the input bit for that path.

The length of the shortest path from position 1 to the output is a random variable over the distribution of P^1. We can provide a simple intuitive argument why the expected length of the shortest path is about $\log n$. This estimate was verified in actual simulations which demonstrated that at least 85% of the tested random permutations had such paths with fewer than $\log n$ edges.

Consider all the strings of length up to $\log n$ bits. Each string can be interpreted as a path from position 1. Some paths reach the output at some point, while others reach only inner positions. We can consider all these strings as the root-to-leaf paths in a full binary tree of depth $\log n$. Each node in this tree represents some state position, except for nodes that one of their ancestors was already the output. Since we are looking for the probability that

the output does not occur in the tree, we can ignore such cases. The number of nodes in a full binary tree of depth $\log n$, including the inner nodes, is about $2n$, so about $2n$ random positions are sampled out of n available positions $\{2, \ldots, n, output\}$. If we make the (non-rigorous) assumption that they are independent and uniformly distributed, then w.h.p there is a string of length up to $\log n$ bits that reaches the output. The actual probability can be estimated by considering the balls and bins problem, in which we throw $2n$ balls into n bins and we count how many bins are covered, that is have at least one ball in them. The expected number of covered bins is $n\left(1 - \frac{1}{e^2}\right) \approx 0.864 \cdot n$. The ratio of 0.864 represents the probability of reaching the particular output bin at least once after following at most $\log n$ edges, and is in excellent agreement with our simulations.

A generalized argument can be used to claim the same bound on 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. However, as will be explained in the actual attack, we would like to have all these paths simultaneously, and thus we have to consider slightly longer paths with lengths of up to $\log n + \log\log n$ edges. Since the number of paths starting at or ending at any intermediate position is about $n \log n$, we expect all these paths to exist w.h.p. This generalized claim was also verified by actual simulations.

4.2 Checking the Validity of Paths

A path in G (defined by a subsequence of ciphertext bits used as deciders) is said to be *valid* if it is a sequence of invocations of P^0 and P^1 that moves a bit from position 1 to the output. A random path is valid with probability of about $\frac{1}{n}$ (with some exceptions such as an all 0's or an all 1's, which reach the output after exactly n jumps regardless of the choice of key). We would like to be able to answer queries of the form "does the following string represents a valid path?" even when we know nothing about K^0 and K^1. If the substring is short, it is likely to occur many times in the ciphertext stream. In all these occurrences, the bit follows the same path, and is thus flipped the same number of times by the same subset of bits in K^0 and K^1. In known plaintext attacks, we know the input bit b and the output bit o that corresponds to each path which happens to be valid, and thus we can verify the path's validity by checking that in all these occurrences $b \oplus o$ is the same value (which we call the *path parity*). Such a decision procedure can introduce one-sided errors, and we call a subsequence of bits *k-positive* if it occurs k times in the ciphertext and in all the occurrences $b \oplus o$ is the same. Clearly, a valid path with k occurrences will always appear k-positive. A non valid path with k occurrences will appear k-positive with probability about 2^{-k+1}, since such an event occurs only when the k values of $b \oplus o$ happen to be the same by chance.

Taking $k = c \log n$ for a sufficiently large constant c, will result in a polynomially small error probability. As will be described in the next subsection, the total number of subsequences we query about is $n^2 \log n$, so we can make the expected number of errors smaller than 1, and thus with a good probability no query is expected to err.

4.3 The Actual Attack

We will define an order over the strings in which, for a and b strings of bits, $a < b$ iff a is shorter than b or if both lengths are equal but $a < b$ lexicographically. The attack will create two mappings. The first mapping is between any position $p \in \{2, \ldots, n\}$ to the first path in this order in G that moves a bit from p to the output. The second mapping is between the end position of the i-th arrow of P^1 to the first path in this order that moves a bit from that

position to the output, for $i \in [n-1]$. Clearly, these $n-1$ end positions are the positions $\{2,\ldots,n\}$, but each end position is unknown. Since every position is uniquely defined by the first path in this order from it to the output, and all the paths in the mappings are chosen first in this order, then every path in the mappings appears twice. By setting every end position of an arrow of P^1 to be the position that shares the same first path in this order as described above, P^1 is discovered. We will divide the attack into three phases: data processing, path mapping, and discovering P^1.

Data processing and query specification In order to efficiently answer queries about the validity of suggested paths, we represent the known ciphertext stream as a suffix tree [5]. For each suffix (a leaf in the suffix tree) we save the index in the ciphertext stream that the suffix began at. Given a suggested path s, a query will use the suffix tree to decide if s is a valid path or not. The query will travel along the suffix tree twice in order to verify that all occurrences of s agree on the same path parity value, as described in Sect. 4.2. The first tree traversal will find all the suffixes that start with $0 \circ s$[3]. Each such suffix is an occurrence of the suggested path in the data. For each occurrence we calculate the index of the output bit in the plaintext and ciphertext streams in order to find the output bit value. If not all the output bits are the same, return false. Else, remember this output bit as x. The second tree traversal is similar but will consider all the suffixes that start with $1 \circ s$. As before, if not all the output bits are the same, return false. Else, remember the output bit as y. Return true only if $0 \oplus x$ is equal to $1 \oplus y$ and the number of occurrences is at least k (as defined in Sect. 4.2).

Paths mapping We build a dictionary of $n-1$ paths of length at most $\log n + \log\log n + c$ bits. This dictionary holds for each position $p \in \{2,\ldots,n\}$, the first path in the order defined above in G from p to the output, t_p. An example of such a dictionary is given in Table 1. We start by finding t_2. We query iteratively the validity of $0 \circ s$ for an increasing order of the string s, by the order defined above, until a valid path is identified. This valid path begins in position 1, moves to position 2 by a single step of P^0 and then moves to the output by s. Therefore, s is the first path in that order from position 2 to the output (with length at most $\log n + \log\log n + c$ bits as was explained in Sect. 4.1), $t_2 = s$. We store t_2 in a dictionary with t_2 as the search-key and 2 as the data. Intuitively, in order to map other positions to their first paths in that order, we have to query for the validity of paths with the structure: $0^i \circ s$ for $i \in [n-1]$ and some string s. Since each such path might have a length of $\Omega(n)$ bits, fewer than k occurrences of it are expected to be in the data so these queries will always fail. In order to work around this difficulty, after finding t_p we find h_p, which is the first path in that order from position 1 to position p, for each $p \in \{2,\ldots,n\}$. Given t_p, the method we use to find h_p is very similar to the method we use to find t_2: querying iteratively the validity of $s \circ t_p$ for an increasing order of the string s, by that order, until a valid path is identified. This valid path begins in position 1, moves to position p by s and moves to the output by t_p, therefore $h_p = s$. In the next iteration, we use h_p to find t_{p+1} by querying iteratively the validity of $h_p \circ 0 \circ s$ for an increasing order of the string s, by that order, until a valid path is identified. This technique ensures that the queries are always up to $2\log n + 2\log\log n + c$ bits long while we keep track of p. Store each t_p in the dictionary with t_p as the search-key and p as the data. At the end of this phase, a dictionary with the first path in that order from every position $p \in \{2,\ldots,n\}$ to the output is available.

[3] We will use the notation $a \circ b$ as the concatenation of the strings a and b.

Table 1: An example of a dictionary of first paths in the order defined in Sect. 4.3, for the permuter illustrated in Fig. 2. The search-keys are the stored paths and the data is the positions.

position p	2	3	4	5	6	7
t_p	11	01	1	10	00	0

Discovering P^1 This phase is very similar to the previous phase but instead of querying paths of the form $0^i \circ s$ we query paths of the form $1^i \circ s$, for $i \in [n-1]$. Another difference is that we do not write to the dictionary but we use the dictionary to find position values. The first step in discovering P^1 is discovering the end position of the first red arrow. As explained above, we query iteratively the validity of $1 \circ s$ for an increasing order of the string s, by the order defined above, until a valid path is identified. This valid path begins in position 1, moves to some unknown position $P^1(1) \in \{2, \ldots, n\}$ by a single step of P^1 and then moves to the output by s. Therefore, s is the first path in that order from position $P^1(1)$ to the output. Since the dictionary already contains the mapping of each position $p \in \{2, \ldots, n\}$ to its first path in that order to the output, we can discover $P^1(1)$ by finding it in the dictionary using s as the search-key. We continue this attacking phase using the same method we used in the previous phase until all the end positions of the red arrows are discovered, which means that P^1 is discovered.

Complexities The length of each path is at most $len = 2\log n + 2\log\log n + c$ bits and we would like each string of this length to have at least $k = O(\log n)$ occurrences to get a low error probability. A stream of $k \cdot 2^{len}$ bits is expected to contain k occurrences for each string of length len bits. Setting the stream longer will increase the probability for such an event. The probability that a string of length len bits will not occur in a stream of length 2^{len} bits is about 37%. Intuitively, the probability that this string will occur fewer than k times in a stream of length $O\left(k \cdot 2^{len}\right)$ is less than $2^{-O(k)}$ (by Chernoff's inequality) which is polynomially decreasing. Therefore, it is enough to have a stream of that length, so we need a total of $O\left(n^2 \log^3 n\right)$ bits of data.

Building the suffix tree is done in linear time in the size of the data, which is $O\left(n^2 \log^3 n\right)$. Each query to the suffix tree costs $O(\log n)$ clock cycles 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[4]. For $p \in [n-1]$, finding t_p and h_p in the two attacking phases costs $O(n \log n)$ suffix tree queries, so the total number of suffix tree queries processed during the attack is $O\left(n^2 \log n\right)$. The dictionary can be implemented with a simple table which requires constant time insertion and lookup. The total time complexity of the attack is thus $O\left(n^2 \log^3 n\right)$.

4.4 Experimental Results

We implemented the full attack on a PC using Intel Xeon X5355 CPU and a single thread simulator written in the C programming language.

Table 2 describes the actual time complexity of the attack for various key sizes (without solving the resultant sparse system of linear equations whose programming is tedious and whose time complexity is well understood). The first column is the size of the permuter. The second column is the full key size $\lceil \log((n-1)!) \rceil + 2n$ in bits. The third column is the partial key size $\lceil \log((n-1)!) \rceil$ in bits, without the additional key-bits needed to define K^0 and K^1.

[4] Even if more leaves happen to be available, there is no need to consider them.

The forth column is the attack time in seconds on a single PC. The fifth column is the value of $\frac{n^2 \log^3 n}{205000}$, which is the expected complexity with an appropriate chosen constant. For each key size, we repeated the attack 100 times and did not encounter any failures. For each test, we uniformly chose a random key and a random plaintext.

Known plaintexts which contain no inherent randomness, such as alternating bits 01010... were also tested in combination with random keys with success probability of 97%. The small decrease in the success probability is mainly due to the fact that some paths were not available in the ciphertext, and thus the paths that the attack uses are not always the shortest and longer paths have fewer occurrences in the data. Consequently, many long paths fail to pass the threshold of k occurrences. This resulted in some end positions of red arrows which were left unknown, and thus the attack failed to discover the full key.

Table 2: Results for uniform key and plaintext distribution.

n	full key size [bits]	size of P^1 [bits]	attack time [sec]	$\frac{n^2 \log^3 n}{205000}$
2^4	73	41	0.115	0.08
2^5	177	113	0.7	0.62
2^6	418	290	4.85	4.32
2^7	966	710	24.7	27.4
2^8	2188	1676	160	163
2^9	4891	3867	917	932
2^{10}	10808	8760	5125	5115

4.5 Extension to Weaker Attack Models

So far, we described the key recovery process in the standard model of known plaintext attack, in which the attacker knows a single long (or several short) sequences of corresponding plaintext and ciphertext bits. However, we can easily extend the attack to weaker models in which the attacker knows all the ciphertext bits but only a small fraction of plaintext bits (which may be isolated, as in the case of arbitrary ASCII characters, where the only known property of the plaintext is that every 8-th bit in it is zero). The crucial observation in this case is that all the decider bits (and thus also all the inputs to the register) are determined by the known ciphertext bits. The attacker uses each available plaintext bit only in order to determine the validity of the short subsequence of ciphertext bits that preceded it, and thus he can use any constant fraction of known plaintext bits (regardless of their locations) in order to carry out the attack with the same asymptotic time and data complexities. Since the test of validity is statistical in nature, it can be based on a majority vote rather than on unanimity and thus it suffices to know that some plaintext bits are slightly biased (without knowing their actual values). For example, it should suffice to know that every hundredth plaintext bit is zero with probability 0.51 in order to carry out the attack. This is an extremely weak attack model, which is almost a pure known ciphertext attack.

5 A Known Plaintext Attack for Jump-Bounded Keys

The previous attack assumed that P^1 is chosen uniformly from all the single-cycle permutations over n positions. Using this assumption we showed that there are short paths from position 1

to the output, as described in Sect. 4. Since each such path is very short, we could find many occurrences of it in the data. However, a cryptographer could try to salvage this scheme by using only a subset of keys in which short paths cannot exist. In this section we assume that P^1 is chosen from a particular distribution that assures that all the paths have length of $\Omega(n)$. In particular, we would like to consider cases in which P^1 is chosen from the set of single-cycle permutations over n positions, where the length of any red arrow is bounded from above by 2. This choice of P^1 is the hardest for the previous attack since it creates the longest possible input to output paths. For example, the permutation: $1 \to 3 \to 2 \to 4 \to 5 \to 6 \to 8 \to 7 \to 9$ maps every position to a position that is up to two positions away. We will not look for short paths in the stream like we did in the previous attack, since they do not exist. Instead, we will describe a certain characteristic in such keys that allows us to split the key into $O(n)$ independent parts and discover these parts iteratively. Another difference from the previous attack is that we will use several short streams, while in the previous attack we could use one long stream. The next two subsections explain the two main ideas used by this attack.

5.1 Analyzing the Structure of Any Distance-2 Permutation

We would like to show that any choice of a single-cycle permutation P^1, as described above, has a certain characteristic which we will exploit in the actual attack in Sect. 5.3.

Definition of a primitive We will regard P^1 as a sequence of jumps along n positions where each jump is described in a relative notion (i.e. a jump from position 4 to position 6 is a "+2 jump"). Let a *primitive* be a minimal sequence of relative jumps such that the sequence travels through all the positions greater than the starting position, i, and less than the ending position, $j > i$, and visits these positions only. For example, the primitive $(+1)$ is a single relative jump of $+1$. Given that this primitive starts in position i, its ending position is $j = i + 1$. Since all relative jumps are bounded to ± 1 and ± 2, only two primitives are possible: $(+1)$ and $(+2, -1, +2)$ as demonstrated in Fig. 4. Note that (-1) and $(-2, +1, -2)$ are not primitives, since they move in the reverse direction.

The structure of P^1 By considering all the possible ways to generate a single-cycle permutation over n positions with each jump bounded by 2, we show that P^1 must consist of a sequence of the primitives $(+1)$ and $(+2, -1, +2)$. It is easy to see that the jump from position 1 is either a $+1$ jump, which is a complete primitive, or a $+2$ jump, in which the next two jumps of P^1 must be $(-1, +2)$ (in order to reach position 2 and be able to continue to other positions), so either way P^1 starts with a primitive. The same argument can be claimed for the next possible jumps, since P^1 cannot visit the same position twice. An exceptional case in which the primitive $(+2, -1, +2)$ can not be used because the starting position is too close to the output, is solved by using the primitive $(+1)$ until reaching the output. Therefore, P^1 is built by alternating between the two primitives, as illustrated in Fig. 4. Notice that while building P^1, each time a primitive reaches its ending position, all smaller positions are covered and all larger positions are free. By exploiting this characteristic we will develop a known plaintext attack.

The number of such permutations follows the recurrence relation $f(n) = f(n-1) + f(n-3)$ with initial values $f(1) = f(2) = 1$ and $f(3) = 2$. This expression behaves asymptotically as $O(\alpha^n)$ for[5] $\alpha \approx 1.4656$. The effective size of the key $\left(P^1, K^0, K^1\right)$ is thus linear in the state size n, if we use an efficient encoding procedure.

[5] $\alpha = \frac{1}{3}\left[1 + \sqrt[3]{\frac{1}{2}\left(29 - 3\sqrt{93}\right)} + \sqrt[3]{\frac{1}{2}\left(29 + 3\sqrt{93}\right)}\right]$.

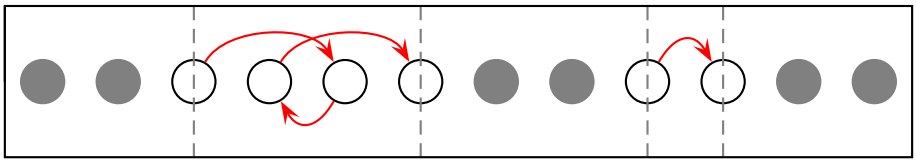

Fig. 4: The two primitives as building blocks for P^1.

5.2 The Number of Required Plaintext Streams

We will use the notion of a state as a particular assignment of values to the bits in the permuter. Let b_i be the bit in position i. As will be explained in Sect. 5.3, the actual attack will try to find a state in which $b_i = b_{i+1}$ and another state in which $b_i \neq b_{i+1}$, for each $i \in [n-1]$. Intuitively, each invocation of a permutation results in another state, so a long stream, in which each ciphertext bit triggers a permutation invocation, would generate many states. However, as will be explained in Sect. 5.3, each sequence of permutation invocations that we want to apply to the initial state should be available in two different streams. Therefore, each such sequence cannot be too long. In order to work around this limitation, we will use S streams, all encrypted by using the same initial value (of zeros) for the state and the decider. In the following analysis, we will give an intuitive argument for the value of S such that enough states will be available for the actual attack, which was verified successfully in actual simulations.

Let k be some integer whose value will be specified later. Let an available state be a state that is generated during the encryption of the first k bits in each stream. Although such a state is unknown (since it depends on the secret key), the actual attack will manage to process it. Intuitively, by setting S to be $O\left(2^k\right)$, all the states that result from any sequence of $k' \leq k$ permutation invocations[6] applied to the initial state are available. The requirement to find a state in which $b_i \neq b_{i+1}$, for some $i \in [n-1]$, will not be satisfied if every sequence q of up to k permutation invocations applied to the initial state results in a state in which $b_i = b_{i+1}$. Since the initial state is all zeros, such an event happens when q applies the same complementation when moving a bit from its initial position to position i and when moving another bit from its initial position to position $i+1$. Intuitively, this event happens when certain bits in K^0 and K^1 are equal. Let us consider which bits in K^0 and K^1 define these complementations by considering a path that is defined by q. A path that starts with an initial jump of P^0 and then applies k jumps can take a state-bit and transfer it $k+1$ positions ahead if the path uses only $(+1)$ primitives (or steps of P^0), $\lceil 1.5k \rceil + 1$ positions ahead if the path uses only $(+2, -1, +2)$ primitives, or $\lceil \lambda k \rceil + 1$ positions ahead for a general permutation P^1, where $1 \leq \lambda \leq 1.5$ depending on P^1 and on the target position i. Therefore, about λk bits from K^0 and about λk bits from K^1 can define the complementations. Since K^0 and K^1 are random, the probability that these bits are the same decreases like $2^{-2\lambda k}$. A similar failure probability can be explained for the case that there is no available state in which $b_i \neq b_{i+1}$.

The total success probability is about $\left(1 - 2^{-2\lambda k}\right)^n > \left(1 - 2^{-2k}\right)^n$. Setting $k = \frac{1}{2} \log n$ will provide a constant success probability as was verified in actual simulations. In the rest of this section we assume that k has this value, and thus we use $S = O\left(\sqrt{n}\right)$ plaintext streams.

[6] The actual number of permutation invocations to the initial state is $k' + 1$ because there is a mandatory initial invocation of P^0 that is caused by the initial value of the decider.

5.3 The Actual Attack

For this attack, we define, for $d \in \{0,1\}$ and $j \in [n]$, the value K_j^d as the complementation of the j-th arrow by the order derived from the permutation P^d. The attack will discover the primitives of P^1 iteratively, from the last primitive to the first. In each iteration we will discover all the key-bits related to a specific primitive. Each iteration consists of up to 4 phases: discovering bits of K^0, discovering the last bit of K^1 in a primitive, discovering the last jump in a primitive, and if we discover the primitive $(+2,-1,+2)$ then the fourth phase is revealing the rest of the bits of K^1 for this primitive. We will describe the i-th iteration, assuming that the end position of the current attacked primitive is x, so all the jumps with their complementation bits from any position $x \leq y \leq n$ were already discovered. The first iteration will start with $x = n+1$, which is the output of the permuter.

Discovering the bits K^0_{x-3}, K^0_{x-2} and K^0_{x-1} Assuming that one of the ciphertext streams begins with 00, we consider 3 sequential invocations of P^0 on the initial state (including the mandatory initial invocation due to the initial value in the decider). These 3 invocations will set b_x, b_{x+1} and b_{x+2} to be $K^0_{x-3} \oplus K^0_{x-2} \oplus K^0_{x-1}$, $K^0_{x-2} \oplus K^0_{x-1} \oplus K^0_x$ and $K^0_{x-1} \oplus K^0_x \oplus K^0_{x+1}$ respectively. Since all the jumps and their complementations are known from previous iterations, we can apply any sequence of permutations to the state and be able to track the position of these 3 state-bits, along with their complementations, until they are outputted from the permuter. Therefore, we can use the sequence of invocations of the current stream (after the first two bits 00) to propagate these state-bits to the output, where we are able to compute the values of the secret key-bits K^0_{x-3}, K^0_{x-2} and K^0_{x-1}.

Discovering the bit K^1_{x-1} We would like to find a string α of up to k bits such that by applying the sequence of invocations of P^0 and P^1 defined by α immediately after the initial mandatory invocation of P^0, the state develops into a state in which $b_{x-1} = b_{x-2}$. In order to check if a specific string results in such a state and in order to use this string in case that the state is suitable, we choose the string α only if there are two ciphertext streams, C_{00} and C_1, that begins with $\alpha \circ 00$ and with $\alpha \circ 1$ respectively. By choosing S large enough, we can assume that such streams exist for any string of up to k bits. The values of b_{x-1} and b_{x-2} can be read by using the technique mentioned above for C_{00}. That is, after the string α was encrypted, the next two invocations of P^0 will set the bits in positions x and $x+1$ to be $b_{x-2} \oplus K^0_{x-2} \oplus K^0_{x-1}$ and $b_{x-1} \oplus K^0_{x-1} \oplus K^0_x$ respectively. We propagate these bits to the output as described above. Since we already know K^0_{x-2} and K^0_{x-1} from the previous phase and K^0_x from the previous iteration, we can compute b_{x-2} and b_{x-1} of the state generated by α. We go over all strings of length up to k and check in each string if $b_{x-1} = b_{x-2}$, until we find such a string. Once we find this string, the value of K^1_{x-1} can be computed using C_1. Since C_1 develops the same state as C_{00} does after inserting the bits of α, then applying P^1 to the state in which $b_{x-1} = b_{x-2}$ will set the bit in position x to either $b_{x-1} \oplus K^1_{x-1}$ or $b_{x-2} \oplus K^1_{x-1}$. We will propagate this bit to the output. Since $b_{x-1} = b_{x-2}$ and their values are known (from using C_{00}), we can compute K^1_{x-1}.

Discovering the last jump of P^1 in the i-th primitive We would like to discover the starting position of the last jump of P^1 in the current primitive. This jump is either a $+2$ jump from position $x-2$ to position x or a $+1$ jump from position $x-1$ to position x. An example for an attack phase is illustrated in Fig. 5, in which all the right side was already discovered and all the left side is still unknown. Once a bit is moved to a position greater or equal to x, we

can track its position and complementation while applying any sequence of permutation steps until the bit is outputted. Intuitively, once we discover a primitive, the gray area expands to the left and we begin to discover the next primitive. We find a string β of up to k bits such that by applying the sequence of invocations of P^0 and P^1 defined by β immediately after the initial mandatory invocation of P^0, the state develops into a state in which $b_{x-1} \neq b_{x-2}$. In order to check and use this, we add another requirement that there are two ciphertext streams, C_{00} and C_1, that begins with $\beta \circ 00$ and with $\beta \circ 1$ respectively. The values of b_{x-1} and b_{x-2} can be read by using the same technique as described in the previous phase. Once we find this string, the starting position of the jump of P^1 to position x can be computed using C_1. Since C_1 develops the same state as C_{00} does after inserting the bits of β, then applying P^1 to the state will set the bit in position x to either $b_{x-1} \oplus K^1_{x-1}$ or $b_{x-2} \oplus K^1_{x-1}$. We will propagate this bit to the output. Since $b_{x-1} \neq b_{x-2}$ and their values are known (from using C_{00}) and also K^1_{x-1} is known, we can identify the starting position of the $x-1$-th jump of P^1: if this jump is $+1$ then the output bit is $b_{x-1} \oplus K^1_{x-1}$ and if the last jump is $+2$ then the output bit is $b_{x-2} \oplus K^1_{x-1}$.

Discovering the rest of a primitive We would like to end each iteration after discovering a complete primitive. If the last jump is $+1$ then we can continue to the next iteration. If the last jump is $+2$, then it must be the last jump of the primitive $(+2, -1, +2)$. In this case the jumps are known, but we still have to discover K^1_{x-2} and K^1_{x-3}. K^1_{x-2} can be identified by using any ciphertext stream that starts with 11. The initial invocation of P^0 will set the value of b_{x-1} to K^0_{x-2}. The next invocation of P^1 will move the bit from position $x-1$ to position $x-2$, setting the value of b_{x-2} to $K^0_{x-2} \oplus K^1_{x-2}$. The second invocation of P^1 will move the bit from position $x-2$ to position x with a value of $K^0_{x-2} \oplus K^1_{x-2} \oplus K^1_{x-1}$. We propagate this bit to the output. Since K^0_{x-2} and K^1_{x-1} are already known, we can compute K^1_{x-2}. K^1_{x-3} can be identified by using two ciphertext streams C_{01} and C_{10}. The ciphertext stream C_{01} starts with 01 and will set the bit in position x to be $K^0_{x-4} \oplus K^0_{x-3} \oplus K^1_{x-1}$, revealing K^0_{x-4} (after propagating the bit to the output). The ciphertext stream C_{10} starts with 10 and will set the bit in position x to be $K^0_{x-4} \oplus K^1_{x-3} \oplus K^0_{x-1}$, revealing K^1_{x-3} (after propagating the bit to the output).

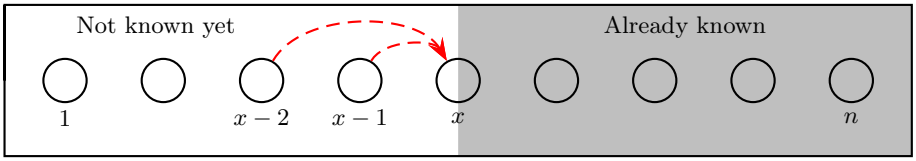

Fig. 5: An illustration of an attack phase.

Complexities The time complexity is the number of primitives times the average work done to discover each primitive. In order to identify a primitive we follow bits across $O(n)$ positions[7]. The number of such bits depends on the number of prefixes of length up to k that we test. In order to discover some primitives out of the $O(n)$ primitives that construct P^1, we may have to check many prefixes. Such an event happens when some bits in K^0 and K^1 are correlated

[7] Unless loops happen. Such loops will occur for example when the ciphertext is 010101... and a state bit is stuck in the sequence of jumps +1,-1,+1,-1,.... Such loops should be rare so we won't regard them.

as explained in Sect. 5.2. While we cannot provide an absolute upper bound for the number of prefixes needed to discover a primitive in P^1, simulations show that the average number of prefixes we test over all the primitives in a single randomly chosen permutation of this type is about 4. The number of primitives is $O(n)$ so the total time complexity is $O\left(n^2\right)$.
The data complexity is the number of streams times the length of each stream. The first k bits of a stream are used to generate a random state, as explained above. The rest of the stream is used to move a bit from some position to the output. Since it takes at most n jumps to output a bit from any position, n additional bits are sufficient for the rest of the stream. Therefore, each stream will be about $k+n$ bits long. The number of streams is $S = O(\sqrt{n})$ so the total data required is expected to be $O\left(n^{1.5}\right)$ bits.

5.4 Experimental Results

We implemented the full attack in python programming language on a PC using Intel Xeon X5472 CPU.

Table 3 describes the actual time complexity of the attack for various key sizes. The first column is the size of the permuter. The second column is the effective key size in bits, defined as $\lceil \log(f(n)) \rceil + 2n$ where $f(n)$ is the number of single-cycle permutations as described above. The third column is the attack time in seconds. The fourth column is the value of $\frac{n^2}{551000}$, which is the expected complexity with an appropriate chosen constant. For each key size, we repeated the attack 100 times where in each test we uniformly chose a random key and $4\sqrt{n}$ random plaintexts of length $1.5n$. None of our experiments resulted in a failure.

Table 3: Results for limited jumps permutations.

n	key size [bits]	attack time [sec]	$\frac{n^2}{551000}$
2^4	41	0.002	0.0005
2^5	81	0.004	0.002
2^6	163	0.009	0.007
2^7	326	0.031	0.03
2^8	653	0.123	0.119
2^9	1306	0.477	0.475
2^{10}	2612	1.897	1.903
2^{11}	5225	7.593	7.612
2^{12}	10451	30.461	30.448

6 Conclusions and Open Problems

In this paper we presented several attacks on the stream cipher proposed by John Nash which showed how to break it even in extremely weak attack models. The main remaining open problem is to extend the attack in Sect. 5 to cases in which P^1 can jump a bounded distance greater than 2, since our technique will require a very tedious analysis of how P^0 and P^1 can interact in this case.

References

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

2. Noam Nisan. John Nash's Letter to the nsa. http://agtb.wordpress.com/2012/02/17/john-nashs-letter-to-the-nsa/, February 2012.
3. 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.
4. Ron Rivest and his students. A chosen plaintext attack on nash's cipher. private communication.
5. Peter Weiner. Linear pattern matching algorithms. In *SWAT (FOCS)*, pages 1–11, 1973.
6. Douglas H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32(1):54–62, 1986.