Perfect Block Ciphers With Small Blocks

Louis Granboulan[1] and Thomas Pornin[2]

[1] École Normale Supérieure; EADS; `louis.granboulan@eads.net`
[2] Cryptolog International, Paris, France, `thomas.pornin@cryptolog.com`

Abstract. Existing symmetric encryption algorithms target messages consisting of elementary binary blocks of at least 64 bits. Some applications need a block cipher which operates over smaller and possibly non-binary blocks, which can be viewed as a pseudo-random permutation of n elements. We present an algorithm for selecting such a random permutation of n elements and evaluating efficiently the permutation and its inverse over arbitrary inputs. We use an underlying deterministic RNG (random number generator). Each evaluation of the permutation uses $O(\log n)$ space and $O((\log n)^3)$ RNG invocations. The selection process is "perfect": the permutation is uniformly selected among the $n!$ possibilities.

1 Introduction

Block ciphers such as AES[1] or DES[2] typically operate on large input data blocks, each consisting of 64 or more bits (128 or 256 bits are now preferred). Using smaller blocks leads to important security issues when encrypting large messages or using the block cipher for a MAC over such a large message. However, some applications need smaller blocks, and possibly non-binary blocks (i.e., a block space size which is not a power of two).

An example of such an application is the generation of unique unpredictable decimal numbers, destined to be typed in by users. Such numbers must be short; this precludes the use of a simple PRNG, which would imply collisions with a non-negligeable probability. A solution is to use a block cipher operating over the set of decimal numbers of the required length: simply encrypt successive values of a counter.

Building a secure block cipher is known to be a tricky task. Small blocks and non-binary alphabets are even more challenging in several ways:

- There are theoretical results on the construction of secure block ciphers, e.g. [3]; more recent examples include [4] and [5]. These constructions implicitely rely on huge block sizes to hide some biases which our security model (which we detail in section 5) cannot tolerate.

[0] This work has been supported in part by the French government through X-Crypt, in part by the European Commission through ECRYPT.

- Usual security analysis techniques try to model the attacker as having access to an encryption or decryption oracle, with some limitations on the number of queries. With a small block size, the complete code book is too small for this model to be adequate.
- Some classes of attacks (differential and linear cryptanalysis) are much more difficult to express on non-binary alphabets. For instance, several distinct ring structures can be applied on the set of decimal digits. Moreover, there is no field of size 10, only rings with some non-invertible elements, which makes things even more complex[6].

For these reasons, there have been only few attempts at designing such block ciphers, e.g. [7] which proposes to build ciphers of arbitrary domains by iterating a larger block cipher or by Feistel-like structures.

In this paper, we investigate another entirely different way of generating permutations, based of the following SHUFFLE algorithm. Let's consider a very small space of input data, for instance the two-digit decimal numbers. The input space has size 100, and there are 100! possible permutations. Each of them may be represented as an array of 100 numbers from 0 to 99, with no two array elements being equal: if the array element of index x contains y, then the permutation maps x to y (we number array elements from 0). If an appropriate source of randomness is available, then the following algorithm uniformly selects a random permutation of 100 elements:

```
SHUFFLE:
 1. fill the array with element $w$ containing the number $w$
 2. $i \leftarrow 0$
 3. select a random number $j$ between $i$ and 99 (inclusive)
 4. if $i \neq j$, swap array elements of index $i$ and $j$
 5. increment $i$
 6. if $i < 99$, return to step 3
 7. the array contents represent the permutation
```

This algorithm was first published by Fisher and Yates[8], then published again by Durstenfeld[9], and popularized by Knuth[10], to the point that this algorithm is usually known as the "Knuth shuffle". If the random number selection at step 3 is uniform, then it can be shown that all possible permutations have an equal probability of being generated. Thus, for very small blocks, we have a practical candidate for a block cipher: use the key in a cryptographically secure seeded PRNG, then generate the complete permutation using the Knuth shuffle, and apply it. A simple linear pass can be used to compute the inverse permutation, which is used for decryption. The uniform selection of the random number at step 3 is a bit tricky if the PRNG outputs bits. Here is an algorithm which outputs uniformly random numbers between 0 and $d - 1$ using a PRNG which outputs bits:

```
RandomNumber:
 1. chose an integer $r$ such that $2^r \geq d$
 2. obtain $r$ random bits and interpret them as a number $x$ such that
    $0 \leq x < 2^r$
 3. compute $t = \lfloor \frac{x}{d} \rfloor$
 4. if $d + td > 2^r$, return to step 2
 5. output $x - td$
```

Depending on r, the probability that the loop occurs can be made arbitrarily low, to the point of being impossible to achieve in practice. A practical implementation may decide to use $r = 256$ and ignore the possibility that the test in step 4 is verified. Ultimately, this would imply a slight selection bias which is sufficiently small to be neglected.

The permutation generated by Shuffle can be viewed as a block cipher, where the key is the PRNG output (or, equivalently, the seed used in the PRNG). Since a truly random generator implies a truly random permutation, any successful attack on that selection system would be a successful distinguisher on the PRNG. Cryptographically speaking, the "block" cipher thus obtained is as perfect as the PRNG used. Unfortunately, this procedure has a cost $O(n)$ (for a permutation over a set of size n), both in CPU and in space, which makes it impractical for values of n beyond, for instance, 10^4. Generating a full permutation means computing each of the n output values; but we do not need the full permutation. Our goal is to evaluate the block cipher (i.e. the permutation) over one given input block. Hence, what we need here is a deterministic procedure which, from the output of a PRNG, may generate a permutation ϕ over n elements such that:

- assuming a perfect random source instead of the PRNG, all possible permutations of n elements have an equal chance of being selected for ϕ;
- for any given x, $\phi(x)$ can be computed with an average computational cost much lower than $O(n)$.

The purpose of this article is to describe such a procedure. It computes $\phi(x)$ for any x in the input data set in time $O((\log n)^3)$, and may also compute $\phi^{-1}(y)$ for any y in the output data set in time $O((\log n)^3)$. The notion of time we use here is the number of requests to the PRNG; we assume a *seekable* PRNG output[3]. Our algorithms are described as recursive for simplicity, but they may easily be transformed into tail recursions; the real space requirements are for a fixed number of integer values lower than n, hence implying a $O(\log n)$ space complexity. We describe our notations more formally in the next section; then we proceed with the algorithm description. Finally, we specify our security model.

[3] Note that the overall number of requests to the PRNG that are needed to define the whole permutation is $O(n(\log n)^3)$ which is bigger than $\log(n!)$.

2 Notations

We consider the problem of selecting a random permutation ϕ which operates over a set of n elements. n is an integer greater than 1; the targeted values of n are between 10^3 and 10^{20}, although our construction may be used with smaller and greater blocks as well.

We need a deterministic PRNG, which constitutes the key for our cipher. We model the PRNG as a function:

$$\begin{aligned} \mathcal{R} : \mathbb{N}^r &\longrightarrow \{0, 1\} \\ (n_1, n_2, \dots n_r) &\longmapsto \mathcal{R}(n_1, n_2, \dots n_r) \end{aligned}$$

The PRNG is assumed to be an unbiased random oracle, in that the output for a given set of input integers cannot be computationally predicted from the knowledge of the output of $\mathcal{R}$ for all other input value sets. From such a PRNG $\mathcal{R}$ with r input parameters, one can easily build a PRNG $\mathcal{R}'$ with $r + 1$ input parameters, by definining:
$\mathcal{R}'(n_1, \dots n_r, n_{r+1}) = \mathcal{R}(n_1, \dots n_{r-1}, (n_r + n_{r+1} + 1)(n_r + n_{r+1} + 2)/2 - n_r - 1)$

In practice, we will need a keyed PRNG with three input parameters, for which we will provide workable bounds.

Our PRNG must be *seekable*, which means that computing the output of $\mathcal{R}$ for some given input parameters must be efficient, regardless of which other parameter values were previously input into $\mathcal{R}$. Unfortunately, this characteristic precludes the use of some of the "provably secure" PRNG such as BBS[11] or QUAD[12]: these PRNG output bits sequentially, and are not seekable.

We will use "log" for the base-2 logarithm, such that $\log 2^x = x$ for all x.

The algorithm description uses binomial terms; in order to simplify the equations, we use extended binomial coefficients, which are defined thus:

$$\binom{n}{p} = \frac{n!}{p!(n-p)!} \qquad \text{if } 0 \le p \le n, \qquad \text{and 0 otherwise.}$$

3 Random Permutation Selection

In this section, we describe our algorithm for computing $\phi(x)$ on input x, as well as the inverse algorithm which computes $\phi^{-1}(y)$ from y. The first algorithm is an extension of a random permutation selection algorithm described in [13] in the context of a massively parallel architecture; we add the ability to compute only $\phi(x)$ and not the whole description of ϕ, as well as an algorithm for ϕ^{-1}.

Our two main algorithms are called PERMUTATOR and INVPERMUTATOR; they compute, respectively, ϕ and ϕ^{-1} on a given input. PERMUTATOR implements ϕ by using a binary tree of "splits", where each split segregates the inputs into two groups. The SPLITTER algorithm implements the split; it itself uses a binary tree of repartitions that REPARTITOR computes. REPARTITOR is the algorithm which uses the PRNG output. INVPERMUTATOR uses INVSPLITTER, which itself relies on REPARTITOR. We describe all those algorithms in the following sections.

3.1 Repartitor

The REPARTITOR has the following inputs:

- integers n and p such that $0 \leq p \leq n$ and $n \neq 0$;
- a PRNG index i.

We define $a = \lfloor n/2 \rfloor$.

Using only $O(\log n)$ PRNG requests $\mathcal{R}(i, \ldots)$, all with the value i as first index, REPARTITOR outputs an integer value u following this distribution:

$$P(u = k) = \frac{\binom{a}{k}\binom{n-a}{p-k}}{\binom{n}{p}}$$

This distribution is known as the hypergeometric distribution. It can easily be shown that values u which have a non-zero probability of being returned must be such that:

$$0 \leq u \leq a$$
$$(a + p) - n \leq u \leq p$$

Note that in the degenerate cases where $p = 0$ or $p = n$, then only one return value is possible ($u = 0$ or $u = a$, respectively), allowing for a fast return.

When $p > a$, one may implement REPARTITOR by using:

$$\text{REPARTITOR}(n, p, i) = a - \text{REPARTITOR}(n, n - p, i)$$

which means that one may always assume that $p \leq a$ when implementing REPARTITOR.

REPARTITOR models the following experiment: from a set of n elements, split into two subsets of size a and $n - a$, p distinct elements are randomly selected. REPARTITOR computes and returns how many of those p elements come from the first subset (of size a). Implementing REPARTITOR efficiently, with a sufficiently unbiased output is tricky; we give a slow but precise algorithm in section 4.

3.2 Splitter

The SPLITTER algorithm has the following inputs:

- integers n and p such that $0 \leq p \leq n$ and $n \neq 0$;
- an input index x ($0 \leq x < n$);
- a PRNG index i.

Using only requests to REPARTITOR$(*, *, j)$ for indexes j such that $i \leq j < i + n - 1$, SPLITTER returns an integer $y = \text{SPLITTER}(n, p, x)$ such that:

$$0 \leq \text{SPLITTER}(n, p, x) < n$$
$$x_1 \neq x_2 \Rightarrow \text{SPLITTER}(n, p, x_1) \neq \text{SPLITTER}(n, p, x_2)$$
$$\#\{x | \text{SPLITTER}(n, p, x) < p\} = p$$
$$\text{SPLITTER}(n, p, x_1) < \text{SPLITTER}(n, p, x_2) \leq p \Rightarrow x_1 < x_2$$
$$p \leq \text{SPLITTER}(n, p, x_1) < \text{SPLITTER}(n, p, x_2) \Rightarrow x_1 < x_2$$

These equations mean that SPLITTER extracts p elements from the set of integer values from 0 to $n-1$; these p elements are given the output indexes 0 to $p-1$, but their order is preserved. The $n-p$ unextracted elements are given the indexes p to $n-1$, and their order is also preserved. There are $\binom{n}{p}$ such "splits" and we require that SPLITTER selects one of those with uniform probability.

SPLITTER is implemented with a "distribution tree". This is a binary tree whose leaves are the n elements; each leaf is thus "extracted" (part of the p elements) or "not extracted" (part of the $n-p$ other elements). Each node in the tree records how many elements among those descending from this node are "extracted". Hence, the root is the ancestor for all n elements, and records the value p. The tree is built from the root: each node manages k elements, and gets f "extractions" to distribute evenly among those k elements; it breaks the k elements into two halves and distributes the f extractions between the two halves (using REPARTITOR to follow the right probability distribution); the same process is invoked recursively on both halves. Computing the whole tree costs $O(n)$ calls to REPARTITOR, but knowing the status (extracted or not) of a given input element x, and computing its final index, can be done by exploring only the tree path from the root to the leaf corresponding to x. If the tree is balanced this costs $O(\log n)$ calls to REPARTITOR, hence $O((\log n)^2)$ PRNG invocations.

Here is an implementation of SPLITTER:

```
SPLITTER:
input: $n$, $p$, $x$ and $i$
output: $y$
 1. if $n = 1$, then return $x$
 2. compute $a \leftarrow \lfloor n/2 \rfloor$
 3. compute $u \leftarrow$ REPARTITOR$(n, p, i)$
 4. if $x \geq a$, go to step 7
 5. compute $t \leftarrow$ SPLITTER$(a, u, x, i+1)$
 6. if $t < u$, return $t$, else return $p + (t - u)$
 7. compute $t \leftarrow$ SPLITTER$(n-a, p-u, x-a, i+a)$
 8. if $t < p - u$, return $t + u$, else return $a + t$
```

Since a is computed such that the tree is well balanced, then the maximum recursion depth is $\lceil \log n \rceil$. It is easily shown that SPLITTER uses the PRNG only for indexes j such that $i \leq j \leq i + n - 2$: this is true for $n = 1$ (no PRNG invocation at all), and is extended by recursion (the first recursive SPLITTER invocation may use only indexes from $i+1$ to $i+a-1$, and the second invocation may use only indexes from $i+a$ to $i+n-2$). Moreover, for a given x value, the same PRNG index is never used twice.

The hypergeometric distribution implemented by REPARTITOR ensures that SPLITTER selects uniformly the split among the possible splits of n elements into sets of length p and $n - p$.

3.3 InvSplitter

The InvSplitter implements a reversal of Splitter: InvSplitter(n, p, y, i) computes x such that Splitter$(n, p, x, i) = y$. Since Splitter implements a permutation of n elements, a unique solution always exists for all y from 0 to $n - 1$.

InvSplitter first checks whether the targeted y is part of the p extracted elements, or not. Then it invokes Repartitor to know how the elements, at that tree level, are split between the left and right parts, and it follows the correct node, depending on whether y was extracted, and at which rank. When the leaf x is reached, then x is the value looked for. Here is the InvSplitter algorithm:

```
InvSplitter:
input: $n$, $p$, $y$ and $i$
output: $x$
 1. if $n = 1$, then return $y$
 2. compute $a \leftarrow \lfloor n/2 \rfloor$
 3. if $y \geq p$, then go to step 5
 4. if $y < u$, return InvSplitter$(a, u, y, i + 1)$;
    otherwise, return $a +$ InvSplitter$(n - a, p - u, y - u, i + a)$
 5. if $y < a + p - u$, return InvSplitter$(a, u, y - (p - u), i + 1)$;
    otherwise, return $a +$ InvSplitter$(n - a, p - u, y - a, i + a)$
```

Note that if $y < p$ (respectively $y \geq p$) then all nested invocations of InvSplitter will also have $y < p$ (respectively $y \geq p$): this is because "$y < p$" is equivalent to "y is from the set of p extracted elements". InvSplitter has cost $O((\log n)^2)$ ($\lceil \log n \rceil$ calls to Repartitor).

3.4 Permutator

The Permutator algorithm selects ϕ and computes $\phi(x)$. ϕ is defined as a binary tree of splits, as defined by Splitter: each tree node splits its input into two halves, and uses Splitter to decide in which half each input element goes. It then invokes itself recursively on the half where the actual input x has gone.

The inputs are n (the input space size), x (the actual input element, between 0 and $n-1$) and i (the base index for PRNG access). For the complete permutation, these parameters are, respectively, n (the total input space size), x and 0.

```
Permutator:
input: $n$, $x$ and $i$
output: $\phi(x)$
 1. if $n = 1$, then return $x$
 2. compute $a \leftarrow \lfloor n/2 \rfloor$
 3. compute $t \leftarrow$ Splitter$(n, a, x, i)$
 4. if $t < a$, return Permutator$(a, t, i + n - 1)$
 5. otherwise, return $a +$ Permutator$(n - a, t - a, i + n - 1 + \gamma(a))$
```

In this description, γ is a function which computes how many PRNG input indexes are used by PERMUTATOR: PERMUTATOR may use indexes from i to $i + \gamma(n) - 1$ (inclusive). γ must be such that:

$$\begin{aligned}
&\gamma(1) \geq 0 \\
&\gamma(n) \geq n - 1 + \gamma(a) + \gamma(n-a) \text{ for } n > 1 \text{ and } a = \lfloor n/2 \rfloor
\end{aligned}$$

The "optimal" γ (where these inequalities are equalities) is achieved with the following formula for $n > 1$:

$$\gamma(n) = nf(n) - 2^{f(n)} + 1$$

where:

$$f(n) = 1 + \lfloor \log(n-1) \rfloor$$

($f(n)$ is the bit length of the binary representation of $n - 1$.)

The depth of the binary tree of splits that PERMUTATOR implements is $\lceil \log n \rceil$, because a is computed to be as close as possible to $n/2$. The number of invocations of REPARTITOR is $O((\log n)^2)$, hence the cost of $O((\log n)^3)$ PRNG invocations.

3.5 InvPermutator

The INVPERMUTATOR algorithm must select the same ϕ than PERMUTATOR with the same PRNG, and then compute $\phi^{-1}(y)$ for a given value y. INVPERMUTATOR explores the same binary tree of splits than PERMUTATOR, but upwards, from the leaves to the root, instead of downwards. At each level, PERMUTATOR uses SPLITTER to know where its current input value goes. For INVPERMUTATOR, we want to know from where a given output comes; we thus use INVSPLITTER.

```
INVPERMUTATOR:
input: $n$, $y$ and $i$
output: $\phi^{-1}(y)$
 1. if $n = 1$, then return $y$
 2. compute $a \leftarrow \lfloor n/2 \rfloor$
 3. if $y < a$, compute $t \leftarrow$ INVPERMUTATOR$(a, y, i + n - 1)$; otherwise,
    compute $t \leftarrow a +$ INVPERMUTATOR$(n - a, y - a, i + n - 1 + \gamma(a))$
 4. return INVSPLITTER$(n, a, t, i)$
```

INVPERMUTATOR first goes recursively to the relevant leaf, and then works upwards, back to the root, where $\phi^{-1}(y)$ will be known.

4 Sampling following the Hypergeometric Distribution

The hypergeometric distribution models the selection of p distinct elements among n, and returns how many of these p elements are taken from the first $a = \lfloor n/2 \rfloor$ elements. For small values of p, this exact procedure can be simulated. When p increases, this becomes impractical, and we need a more efficient sampling method. As outlined in section 3.1, we may assume that $p \leq a$.

4.1 Acceptance-rejection method aka. rejection sampling

This classical method, due to von Neumann, generates sampling values from an arbitrary probability distribution function $f(k)$ by using an instrumental distribution $g(k)$, under the only restriction that $f(k) < Mg(k)$ where $M > 1$ is an appropriate bound on $f(k)/g(k)$.

The algorithm runs as follows: k is sampled following the distribution g. Then $f(k)$ is computed and a random integer $y \in [0, 1]$ is generated. If $y \leq \frac{f(k)}{Mg(k)}$, then k is output, else nothing is output and a new k must be sampled.

If g can be perfectly sampled, and if $\frac{f(k)}{Mg(k)}$ and y can be computed with arbitrary precision, the output of the algorithm follows exactly the distribution f. In practice, since one only needs to decide whether $y \leq \frac{f(k)}{Mg(k)}$ or not, the average precision needed is only a few bits.

We want to select according to the hypergeometric distribution $HG_{n,\alpha,p}$ where $\alpha = \lfloor n/2 \rfloor / n$, which we bound with the binomial distribution $B_{\alpha,p}$, itself being bounded by the Cauchy-Lorentz distribution $CL_{\mu,\nu}$ where $\mu = \alpha p$ and $\nu = 2\alpha(1-\alpha)p$:

$$HG_{n,\alpha,p}(k) = \frac{\binom{\alpha n}{k}\binom{(1-\alpha)n}{p-k}}{\binom{n}{p}}$$

$$B_{\alpha,p}(k) = \binom{p}{k}\alpha^k(1-\alpha)p-k$$

$$CL_{\mu,\nu}(x) = \frac{1}{\pi}\left(\frac{\sqrt{\nu}}{(x-\mu)^2+\nu}\right)$$

HG and B are discrete, whereas CL is continuous.

4.2 Upper-bounding distributions

Theorem 1. *If $\alpha \leq 1/2$, then:*

$$\frac{HG_{n,\alpha,p}(k)}{B_{\alpha,p}(k)} < \frac{2^{(1-2\alpha)n}}{(2\alpha)^p}\sqrt{1+\frac{p}{n-p}}.$$

Theorem 2. *If $k \in \mathbb{N}$ and $\alpha = 1/2$, then:*

$$\frac{B_{\alpha,p}(k)}{CL_{\alpha p,2\alpha(1-\alpha)p}(k)} \leq \sqrt{\pi}.$$

Proofs for these theorems are provided in appendix A and B, respectively.

The values of interest to us are $\alpha = \lfloor n/2 \rfloor / n$ for large n and sufficiently large p. For these values, and for any x, there exists a constant c such that:

$$\frac{B_{\alpha,p}(\lfloor x + 1/2 \rfloor)}{CL_{\alpha p, 2\alpha(1-\alpha)p}(x)} \leq c\sqrt{\pi}.$$

We decide that if $p \leq 10$ or $n \leq 10$, then we will use another method for Repartitor, namely performing p random selections. Under those conditions, $c = 1.2$ is an appropriate constant.

4.3 Cauchy-Lorentz distribution by inverse method

If $u \in [0, 1[$ is selected uniformly (and $u \neq 1/2$), then the value $\mu + \sqrt{\nu}\tan(\pi u)$ follows the $CL_{\mu,\nu}$ distribution. If $x = \sqrt{\nu}\tan(\pi u)$ then we need, for Repartitor, the integer closest to $x + \mu$, and we have $CL_{\mu,\nu} = \frac{\sqrt{\nu}}{\pi(x^2+\nu)}$.

4.4 Description of Repartitor

The full Repartitor handles the situation where $p > a$, and where $p \leq 10$ (if $n \leq 10$ then $p \leq 10$):

```
Repartitor:
input: $n$, $p$ and $i$
output: $k$
 1. $a \leftarrow \lfloor n/2 \rfloor$
 2. if $p > a$ then return $a - $ Repartitor$(n, n-p, i)$
 3. if $p > 10$ then return RepartitorReject$(n, p, i)$
 4. $n_1 \leftarrow a$, $n_2 \leftarrow n - a$
 5. if $p = 0$, then return $a - n_1$
 6. select randomly $r$ between 0 and $n_1 + n_2 - 1$ (inclusive)
 7. if $r < n_1$, then $n_1 \leftarrow n_1 - 1$, else $n_2 \leftarrow n_2 - 1$
 8. $p \leftarrow p - 1$
 9. go to step 5
```

The random selection of r is done using an algorithm similar to RandomNumber, working with the stream of random bits output by $\mathcal{R}(i, j, 0)$ for values of j beginning with 0. The average number of invocations of $\mathcal{R}$ for this process will be less than $p \log n$; it is extremely improbable that more than 10 times this value is actually needed.

RepartitorReject implements the general selection with the rejection method. We have shown that, for the parameters used with Repartitor:

$$\frac{HG_{n,\alpha,p}(\lfloor x + 1/2 \rfloor)}{CL_{\alpha p, 2\alpha(1-\alpha)p}(x)} < 1.2\sqrt{\pi}\frac{2^{(1-2\alpha)n}}{(2\alpha)^p}\sqrt{1 + \frac{p}{n-p}}$$

For positive integers i and j, we define the value $u_{i,j}$: $u_{i,j} = \sum_{k\geq 0} \mathcal{R}(i,j,k)2^{-k-1}$ (the conceptually infinite stream $\mathcal{R}(i,j,*)$ defines the binary digits of $u_{i,j}$).

```
RepartitorReject:
input: $n$, $p$ and $i$
output: $k$
 1. define $a = \lfloor n/2 \rfloor$, $\mu = ap/n$ and $\nu = 2(a/n)(1-a/n)p$
 2. define $M = 1.2\sqrt{\frac{\nu}{\pi}}\frac{2^{(1-2\alpha)n}}{(2\alpha)^p}\sqrt{1+\frac{p}{n-p}}$
 3. $l \leftarrow 0$
 4. $l \leftarrow l+1$
 5. $x \leftarrow \sqrt{\nu}\tan(\pi u_{i,2l-1})$
 6. $k \leftarrow \lfloor x + \mu + 1/2 \rfloor$
 7. if $k < 0$ or $k > p$, then go to step 4
 8. $H \leftarrow \binom{a}{k}\binom{n-a}{p-k}/\binom{n}{p}$
 9. if $u_{i,2l} \leq (x^2+\nu)H/M$, then return $k$
10. go to step 4
```

The value of k needs to be determined exactly, which means that x will have to be computed with enough precision. Required precision increases linearly with the derivative of tan over the range of accepted values. Here, the maximum tangent value is on the order of $\sqrt{\mu}$, with a derivative of μ, whose size is that of n. Therefore, we need about $2\log n$ random bits (from $u_{i,2l-1}$) to compute k.

The other computations need be precise enough only for the final test against $u_{i,2l}$ to be meaningful; precision must be raised only if the test is not clear cut, which happens with a probability on the order of 2^{-r} if r bits of precision are used. On the average, only $O(1)$ random bits are used for $u_{i,2l}$.

5 Security Model

We use the following security model: the attacker is given access to a black box implementing Permutator for an internal secret key, with a message space of size n. The attacker may perform up to $n-2$ encryption or decryption (adaptive) queries. The attacker must then predict the output of Permutator for the encryption of an hitherto unseen input. His probability of success is at most 0.5 if the black box implements a truly random permutation.

Using the terminology of [3], our attacker is an $(n-2)$-*limited adaptive distinguisher* in the model of *super-pseudorandomness* (the oracle accepts both encrypt and decrypt queries). However, this terminology is here quite stretched. As an illustration, the Luby-Rackoff construction offers *no* security against our attacker: since a Feistel network can only implement an even permutation, knowing the outputs for $n-2$ inputs implies only a single possible permutation, and our attacker has a probability 1 of success.

We consider that we achieve a security level of r bits if the attacker cannot succeed with probability greater than $0.5 + 2^{-r}$, except by demonstrating that the PRNG output is not truly random. With the REPARTITOR implementation described in section 4, we achieve an "infinite" level of security, i.e. PERMUTATOR is as robust as the PRNG used: any successful attack is mechanically a proof that the PRNG is not a truly random source, i.e. a distinguisher attack on the PRNG.

However, PERMUTATOR may tolerate a slightly biased REPARTITOR. PERMUTATOR is, basically, an acyclic graph of REPARTITOR invocations, containing about $n \log n$ nodes. For a given permutation ϕ, there is a single set of output values for these nodes which yields the ϕ permutation. Each REPARTITOR should follow the hypergeometric distribution $H_{m,p}(u)$ to ensure uniform permutations selection ($H_{m,p}(u)$ is the probability that the unbiased REPARTITOR outputs u on inputs m and p). We now consider a biased REPARTITOR with the distribution $H'_{m,p}$ such that:

$$\sum_{H'_{m,p}(u)=0} H_{m,p}(u) \leq \epsilon$$
$$H'_{m,p}(u) \neq 0 \Rightarrow \left| \frac{H_{m,p}(u)}{H'_{m,p}(u)} - 1 \right| \leq \epsilon$$

for a value $\epsilon \ll 1/n$. Then, for a given permutation ϕ:

– either one of the $H'_{m,p}(u)$ is 0 for ϕ, which means that ϕ cannot be selected; this happens with probability at most $\epsilon n \log n$;
– or ϕ is selected with probability $P = \prod H'_{m,p}(u)$, which is close to the theoretical unbiased probability: $|P/n! - 1| \leq \epsilon n \log n$.

To achieve a security level of r bits, it is sufficient that these two probabilities be lower than 2^{-r}, i.e. $\log \epsilon \leq -(r + \log n + \log \log n)$. Practically, for 128 bits of security and $n = 2^{32}$, ϵ is 2^{-165}. Any implementation of REPARTITOR which achieves that level of unbiasness will fulfill our security requirements.

6 Conclusion

We have presented a new algorithm for constructing a pseudo-random permutation over a space of size n from a seekable PRNG. The PRNG must take three numerical indexes as parameters, the first never exceeding $\gamma(n) \approx n \log n$, while the other two parameters remain below $100 \log n$ with an overwhelming probability. This is a "perfect" block cipher, up to the computational indistinguishability of the PRNG.

We implemented PERMUTATOR in C, using the MPFR[14] library for arbitrary precision computations. Values of n up to $2^{32} - 1$ are supported. The underlying PRNG uses the AES[1], by encrypting the concatenation of the three PRNG indexes to produce 128 bits of random. Performance is quite inadequate: for $n = 10^9$, a 2 GHz PC requires about 0.48 seconds to compute the image of a single value by PERMUTATOR. Profiling shows that more than 98% of the time is spent in the floating-point computations.

We believe that a much simpler implementation of REPARTITOR can be designed, for much improved performance. As outlined in section 5, a slightly biased REPARTITOR can be tolerated, which opens the way to all kinds of approximations.

References

1. *Advanced Encryption Standard*, National Institute of Standards and Technology (NIST), FIPS 197, 2001.
2. *Data Encryption Standard*, National Institute of Standards and Technology (NIST), FIPS 46-3, 1999.
3. *How to construct pseudo-random permutations from pseudo-random functions*, M. Luby and C. Rackoff, Lecture Notes in Computer Science, Proceedings of Crypto'85, 1985.
4. *Dial C for Cipher*, T. Baignères and M. Finiasz, Lecture Notes in Computing Science, Lecture Notes in Computing Science, Proceedings of SAC 2006, 2006.
5. *KFC - the Krazy Feistel Cipher*, T. Baignères and M. Finiasz, Lecture Notes in Computing Science, Proceedings of ASIACRYPT 2006, 2006, pp. 380–395.
6. *Pseudo random Permutation Families over Abelian Groups*, L. Granboulan, E. Levieil and G. Piret, Preproceedings of FSE 2006, Austria, March 15-17, 2006.
7. *Ciphers with Arbitrary Finite Domains*, J. Black and P. Rogaway, Proceedings of RSA CT'02, LNCS, vol. 2271, Springer, 2002, pp. 114–130.
8. *Statistical Tables*, R. A. Fisher and F. Yates, London, 1938, example 12.
9. *CACM*, R. Durstenfeld, **7**, 1964, pp. 420.
10. *The Art of Computer Programming*, D. Knuth, Volume 2, Third edition, 1997, pp 145.
11. *A simple unpredictable pseudorandom number generator*, L. Blum, M. Blum and M. Shub, SIAM Journal on Computing, 15, 1986, pp. 364–383.
12. *QUAD: A Practical Stream Cipher with Provable Security*, C. Berbain, H. Gilbert and J. Patarin, Lecture Notes in Computing Science, Proceedings of EUROCRYPT 2006, 2006, pp. 109–128.
13. *Fast Generation of Random Permutations via Networks Simulation*, A. Czumaj, P. Kanarek, M. Kutylowski and K. Lorys, European Symposium on Algorithms, 1996, pp. 246–260.
14. *The MPFR Library*, `http://www.mpfr.org/`

A Proof of theorem 1

We now prove theorem 1. Let's compute this:

$$\frac{HG_{n,\alpha,p}(k)}{B_{\alpha,p}(k)} = \frac{\binom{n-p}{\alpha n-k}}{\binom{n}{\alpha n}} \frac{1}{\alpha^k(1-\alpha)^{p-k}}.$$

If $\alpha \leq 1/2$, then $\alpha \leq 1-\alpha$, therefore $\frac{1}{\alpha^k(1-\alpha)^{p-k}}$ reaches its maximum for $k = p$. Hence $\frac{1}{\alpha^k(1-\alpha)^{p-k}} \leq \alpha^{-p}$.

Define $m = \alpha n$ and $\beta = n - 2m = (1-2\alpha)n$. We have three possibilities:

- $p \geq \beta$ and $p - \beta$ is even;
- $p \geq \beta$ and $p - \beta$ is odd;
- $p < \beta$.

If $p = 2q+\beta$, the maximum of $\binom{n-p}{\alpha n-k} = \binom{2m-2q}{m-k}$ is achieved for $k = q$; hence:

$$\frac{HG_{n,\alpha,p}(k)}{B_{\alpha,p}(k)} \leq \alpha^{-p} \frac{\binom{2m-2q}{m-q}}{\binom{2m}{m}} = \frac{2^{-2q}}{\alpha^p} \frac{F(m-q)\sqrt{m}}{F(m)\sqrt{m-q}} < \frac{2^\beta}{(2\alpha)^p} \sqrt{1+\frac{p}{n-p}}.$$

If $p = 2q+\beta+1$, the maximum of $\binom{n-p}{\alpha n-k} = \binom{2m-2q-1}{m-k}$ is achieved for $k = q$ or $k = q+1$ (same value); hence:

$$\frac{HG_{n,\alpha,p}(k)}{B_{\alpha,p}(k)} \leq \alpha^{-p} \frac{\binom{2m-2q-1}{m-q}}{\binom{2m}{m}} = \frac{2^{-2q-1}}{\alpha^p} \frac{F(m-q)\sqrt{m}}{F(m)\sqrt{m-q}} < \frac{2^\beta}{(2\alpha)^p} \sqrt{1+\frac{p}{n-p}}.$$

If $p < \beta$, the maximum of $\binom{n-p}{\alpha n-k} = \binom{2m+\beta-p}{m-k}$ is achieved for $k = 0$; hence:

$$\frac{HG_{n,\alpha,p}(k)}{B_{\alpha,p}(k)} \leq \alpha^{-p} \frac{\binom{2m+(\beta-p)}{m}}{\binom{2m}{m}} = \alpha^{-p} 2^{\beta-p} \prod_{i=1}^{\beta-p} \frac{1+i/(2m)}{1+i/m} < \frac{2^\beta}{(2\alpha)^p}.$$

B Proof of theorem 2

We need the following lemma:

Lemma 1. *The function $F(k) = \binom{2k}{k} 2^{-2k}\sqrt{k}$ is strictly increasing, with limit $\pi^{-1/2} = 0.56418\ldots$*

Proof. The limit of $F(k)$ when $k \to \infty$ is easily deduced from Stirling's formula:

$$F(k) = \frac{(2k)!}{(k!)^2} 2^{-2k}\sqrt{k} \approx \frac{(2k)^{2k} e^{-2k} \sqrt{4\pi k}\sqrt{k}}{k^{2k} e^{-2k} 2\pi k 2^{2k}} = \frac{1}{\sqrt{\pi}}.$$

To prove that F is strictly increasing, it suffices to remark that:

$$\frac{F(k+1)^2}{F(k)^2} = 1 + \frac{1}{4k(1+k)} > 1.$$

We may now see that:

$$\frac{B_{1/2,p}(k)}{CL_{p/2,p/2}(k)} = \pi F(p/2) \frac{\binom{p}{k}}{\binom{p}{p/2}} \left(\left(\frac{k}{p/2} - 1 \right)^2 + 1 \right)$$

which reaches its maximum for $k = p/2$. This maximum is $\pi F(p/2) < \sqrt{\pi}$.