

BMGL: Synchronous Key-stream Generator with Provable Security (Revision 1)

Johan Håstad*

NADA, Royal Inst. of Technology
SE-10044 Stockholm, Sweden

Mats Näslund†

Communications Security Lab
Ericsson Research
SE-16480 Stockholm, Sweden

March 6, 2001

Abstract

We propose a construction of an efficient, synchronous keystream generator with provable security properties in response to the NESSIE call for primitives. The cryptographic core of the stream cipher is the block cipher Rijndael. We show that a non-trivial attack on the cipher reduces to an attack on Rijndael.

The construction uses an optimization of earlier work on pseudo-random generators by Blum and Micali, and Goldreich and Levin.

1 Introduction

For confidentiality, the strongest notion of security that we can hope to achieve in practice is so called *semantic security*: any computational information about a message m that can be non-trivially derived from an encryption, $E(m)$, can also be obtained without $E(m)$. This notion was put forward in the seminal work by Goldwasser and Micali [12].

Unfortunately, semantic security cannot be achieved by deterministic, public key algorithms, and existing design-techniques for (non number-theoretic) symmetric block ciphers, gives little hope of ever being able to prove semantic security for such. On the other hand, the security of a stream cipher is, from a theoretical point of view, in some sense “easy” to understand, as the property needed from the keystream to give a semantic security is precisely that of

*johanh@nada.kth.se

†mats.naslund@era-t.ericsson.se

pseudo randomness: it should be computationally infeasible to distinguish, with non trivial success rate, the keystream from a completely random stream of the same length. If we can construct a pseudo-random generator, we therefore also have a good stream cipher; the key is simply the seed of the generator. However, understanding what is to be meant by “pseudo-random” was until quite recently a very open-ended topic, full of imprecise statements, mainly focusing on defining fixed sets of “statistical tests” that would determine if a generator produced pseudo-random numbers or not, see e.g. [14].

A sound theory of pseudo randomness emerged in the seminal works Blum and Micali, [7], and Yao [28] in the early 80’s. In a theoretical sense the area was closed when, in the paper [13], it was shown that necessary and sufficient conditions for the existence of a pseudo-random generator is the existence of another fundamental primitive: the *one-way function*. This is a function easy to compute, but hard to invert. We do not know if such functions exist, but many strong candidates exist, such as cryptographic hash functions, or perhaps a good block cipher (the latter viewed as a function mapping keys to ciphertexts, keeping the plaintext fixed).

Still, the construction in [13] is far too complex to have any practical implication. Keysizes of thousands (if not millions) of bits are needed to get any security out of the generator. This is due to the fact that one-wayness is in itself not a very strong property. In fact, a function may be hard to invert but still have some very undesirable properties. For instance, even if f is one-way, almost all of x may be easily deduced from $f(x)$. Secondly, generating a pseudo-random keystream will typically require iteration of some function and even if some f is one-way, it may lose its one-wayness if iterated. Thus, basing pseudo-randomness on one-wayness alone is a delicate matter, appearing to require elaborate constructions.

However, if one assumes more than just one-wayness, e.g. that the function f is also a permutation, then the situation becomes much more favorable and much simpler constructions can be found. In fact, from the work of Blum and Micali mentioned above, and later work by Goldreich and Levin [11], a general construction that is “almost practical” can be obtained. Basically, [7] focuses on the two undesirable properties mentioned above: information “leakage” of x through $f(x)$ and loss of one-wayness when iterated. They show that if f is a permutation and has at least a single bit of information, $b(x)$, that does not leak via $f(x)$, then a pseudo-random generator can be built. In [11], then, it is shown that every one-way function, in particular ones being permutations, have such a hard bit $b(x)$. In this paper we start with a widely believed secure block cipher, Rijndael [23], as our f , and then proceed by optimizing the above theoretical works, bringing the necessary security parameters (key-sizes) down to practical values of just above 100 bits. As a concrete example, we can show that if we base our construction on the 256-bit version of Rijndael, and generate 1Gbit of keystream, then an attack that correctly distinguishes this keystream from a truly random 1Gbit string with success rate 2^{-32} , gives an attack on Rijndael that is at least 2^{21} times more successful than one would expect for a “perfect” cipher, see §2.4.3.

Admittedly, the stream-cipher constructed here is slower than algorithms such as (the alleged) RC4 or SEAL [26, 25]. However, practical experiments indicate that the performance is completely acceptable in cases where a stream cipher (or pseudo random generator) is needed and security is at the forefront. Another strength of the construction is that it can, if any weakness in Rijndael is found, be based on any other conjectured one-way function; another block cipher, or even a cryptographic hash function.

The paper is organized as follows. We first give some preliminary background in §2, describing in more detail the theoretical results upon which the actual construction (presented in §2.4) is built. We then give a formal proof of security, §2.4.2. Some estimates of efficiency in practical implementations appear in §3.1.5, and a few “non-mathematical” attacks are discussed in §3.2.

2 Description

2.1 Notation

The length of binary string x is denoted $|x|$, and by $\{0, 1\}^n$ we denote the set of x such that $|x| = n$. We write \mathcal{U}_n for the uniform probability distribution on $\{0, 1\}^n$. Except otherwise noted, \log refers to logarithm in base 2 (and \ln is the natural logarithm).

We remind the reader of the two most fundamental concepts in cryptography and the theory of pseudo-randomness.

A *one-way function* is a function f so that f is polynomial time computable, but for any probabilistic, polynomial time algorithm, I , any $c > 0$ and sufficiently large n ,

$$\Pr[f(I(f(x))) = f(x)] \leq \frac{1}{n^c},$$

probability over $x \in \mathcal{U}_n$ and I 's random choices.

Secondly, we say that two probability distributions D_1, D_2 , on strings of length n are (computationally) $\delta(n)$ -*distinguishable*, if there is an efficient probabilistic algorithm, A , such that

$$\left| \Pr_{y \in D_1} [A(y) = 1] - \Pr_{y \in D_2} [A(y) = 1] \right| \geq \delta(n).$$

Here, $\delta(n)$ is called the *advantage* of A . If no such A exist, D_1, D_2 are called $\delta(n)$ -indistinguishable. The reader familiar with statistical tests may think of A as such, trying to decide which distribution it sees as input. Note though that what A actually does is completely unspecified—any feasible test is permitted. Finally, a sequence of distributions $\{D_n\}_{n \geq 1}$, where D_n has support on $\{0, 1\}^n$, is said to be *pseudo-random* if for any $c > 0$ and sufficiently large n , D_n is n^{-c} -indistinguishable from \mathcal{U}_n .

2.2 Pseudo-random Generators from One-way Function

Suppose we have a one-way function, that in addition is a permutation, i.e. for each n , f is a one-to-one correspondence $\{0, 1\}^n \rightarrow \{0, 1\}^n$. Furthermore, suppose that we have a family of 0/1-functions, $B = \{b_r\}$, $b_r(x) \in \{0, 1\}$, with the property that given $f(x)$ and a randomly chosen b_r , $b_r(x)$ is computationally indistinguishable from a random 0/1 coin toss. (I.e. predicting $b_r(x)$, non-negligibly exceeding the trivial guessing strategy's success rate of $1/2$ is infeasible.) Then, the following construction, due to Blum and Micali [7], shows how to construct a pseudo-random generator (PRG).

Choose x_0, r (the seed), let $x_{i+1} = f(x_i)$, output $g(x_0, r) = b_r(x_1), b_r(x_2), \dots$ as the generator output.

Theorem 1 (Blum-Micali, '86). *Suppose there is an efficient algorithm D that distinguishes (with non-negligible advantage) $g(x, r)$ from a completely random string. Then, there is an efficient algorithm P that given $r, f(x)$ predicts $b_r(x)$ with non-negligible advantage.*

We shortly give proof of this theorem for the specific case of the generator constructed in this paper, see §2.4.2. A set of functions, B as above, is called a (family of) *hard-core functions* for f . Notice that for the theorem to say anything, it is really necessary that f is a one-way function. If not, we *can* predict $b_r(x)$ from $r, f(x)$ by first inverting f and we would not get any contradiction.

For the above theorem it is important that f is a permutation. The problem if it is not, is that x_i might then after a few iterations of f become restricted to a small subset of the domain of f . Possibly, f could be easy to invert on that small subset. If f is not a permutation, but f behaves like a random function, however, then it can be shown (see Theorem 3) that after a moderate number of iterations, f is still one-way even when restricted to this set of inputs. Assumptions along these lines have been proposed by Levin [15] and are in fact necessary and sufficient of the existence of pseudorandom generators.

In summary, from a theoretical standpoint, this leaves us with the question: which one-way functions (if any) have hard-cores, and if so, what do these hard-cores look like?

2.3 The Goldreich-Levin Theorem

In 1989, Goldreich and Levin [11], proved that *any* one-way function (not only permutations) have hard-cores¹. Perhaps surprisingly, the hard-cores they found are also extremely simple to describe. If r, x are binary strings of length n , let r_i (and x_i) denote the i th bit of r (and x), fixing an order left-to-right, or right-to-left. The set $B = \{b_r\}$ consists of 2^n functions, each indexed by such a string

¹We again stress that this does not automatically imply that a PRG can be built from any one-way function, as the construction by Blum and Micali only works for one-way permutations. It is true that PRGs *can* be built from any one-way function, but without the permutation assumptions, the construction no longer becomes practical, [13].

r and $b_r(x)$ is defined as

$$b_r(x) \triangleq r_1 \cdot x_1 + r_2 \cdot x_2 + \cdots + r_n \cdot x_n \pmod{2},$$

that is, the inner product mod 2.

Theorem 2 (Goldreich-Levin, '89). *Suppose there is an efficient algorithm A , that given $r, f(x)$ for randomly chosen r, x , distinguishes (with non-negligible advantage) $b_r(x)$ from a completely random bit. Then there exists an efficient algorithm B , that inverts $f(x)$ on random x with non-negligible probability.*

(A proof is given in §2.4.2.) So, if f is believed to be a one-way function, the existence of such A would be a contradiction. Before proceeding we note that there are also other functions, known to be hard-core for any one-way function, e.g. [21], but for the purpose of this paper, there is no need to look beyond the Goldreich-Levin construction.

So far, the construction above gives one pseudo-random output bit, $b_r(x)$, per application of the function f . Even if f is fairly easy to compute, it is obvious that computing b_r will in practice be negligible compared to computing f . As established already in [11], one way to improve efficiency would therefore be to extract more than one bit at a time.

Indeed, it turns out that instead of outputting a single inner product mod 2, it is possible to output as many as $m \in O(\log n)$ (where $n = |x|$) b_r s, corresponding to multiplying the binary vector x by a random $m \times n$ binary matrix, R . We denote the set of all such matrices \mathfrak{M}_m , and the corresponding functions $\{B_R^m \mid R \in \mathfrak{M}_m\}$. That is,

$$B_R^m(x) = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,n} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,n} \\ \vdots & \vdots & & \vdots \\ r_{m,1} & r_{m,2} & \cdots & r_{m,n} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \pmod{2}.$$

It is clear that from a theoretical point of view, outputting $\sim \log n$ bits is, in general, the best we can hope for. To give a security proof in practice with a real function f we need an exact analysis keeping track of all constants and that analysis is the bulk of this paper. To do such an exact analysis, we need to fix a computational model.

Except otherwise noted, the following model is used. For a given security parameter n (e.g. the key/block size of a block cipher), we shall treat operations on n -bit strings (such as taking their bit-wise XOR), as elementary, taking constant time.

2.4 The Construction

As described above we have a general construction given any one-way function.

Definition 1. Let n , and m, L, λ be integers such that $L = \lambda m$ and let $f : \{0, 1\}^n \rightarrow \{0, 1\}^n$. The generator $BMGL_{n,m,L}(f)$ stretches $n + nm$ bits to L bits as follows. The input is interpreted as x_0 and $R \in \mathfrak{R}_m$. Let $x_i = f(x_{i-1})$, $i = 1, 2, \dots, \lambda$ and let the output be $\{B_R^m(x_i)\}_{i=1}^\lambda$.

Our goal is to see how secure this generator is given an assumption about the difficulty of inverting f (i.e. Rijndael). We first define security in terms of a distinguisher.

Definition 2. Let G be a generator, stretching n bit seeds to $L(n)$ bits, and let D be a probabilistic algorithm computing a function $\{0, 1\}^{L(n)} \rightarrow \{0, 1\}$. Let $p_r \triangleq \Pr_{z \in \mathcal{U}_{L(n)}}[D(z) = 1]$ and $p_G \triangleq \Pr_{s \in \mathcal{U}_n}[D(G(s)) = 1]$. We then say that D is a $(L(n), T(n), \delta(n))$ -distinguisher for G if D runs in time $T(n)$ and $p_r \geq p_G + \delta(n)$. If there is no $(L(n), T(n), \delta(n))$ -distinguisher for G then it is said to be $(L(n), T(n), \delta(n))$ -secure.

Our definition of a distinguisher is only stated for the case of distinguishing random bits from the output of a generator. We note that there are more general situations where the distinguisher is trying to distinguish two distributions A and B and also that it can be given an element of a third distribution C , which might be coupled with A and/or B , as an aid. We do not give the formal definition of this concept at this moment.

Though we will not here make any particular assumption on the running time of the distinguisher, it is interesting to note that “practical” distinguishers are almost always faster than the generator undergoing the test (excluding poor generators such as linear congruential ones). This can be verified by anyone having experience with tests similar to Diehard, [18], or those proposed by Knuth [14].

The next section exactly relates the difficulty of inverting an iterated function f to the possibility of distinguishing the output of $BMGL_{n,m,L}(f)$ from random bits. We first define our measure of success for the inverter.

Definition 3. For a function $f : \{0, 1\}^n \rightarrow \{0, 1\}^n$, let $f^{(i)}(x)$ denote f iterated i times, $f^{(i)}(x) \triangleq f(f^{(i-1)}(x))$, $f^{(0)}(x) \triangleq x$.

Let A be a probabilistic algorithm which takes an input from $\{0, 1\}^n$ and has output in the same range. We then say that A is a (T, δ, i) -inverter for f if when given $y = f^{(i)}(x)$ for an x chosen uniformly at random, in time T with probability δ it produces z such that $f(z) = y$.

Note the the number z might be on the form $f^{(i-1)}(x')$ but this is not required. It is interesting to investigate what happens for a random function.

Theorem 3. Let A be an algorithm that tries to invert a black box function $f : \{0, 1\}^n \rightarrow \{0, 1\}^n$, and makes T calls to the oracle for f . If A is given $y = f^{(i)}(x)$ for a random x , then the probability (over the choice of f and x) that A finds a z such that $f(z) = y$ is bounded by $T(i+1)2^{-n}$. On the other hand, there is an algorithm that using at most T oracle calls outputs a correct z except with probability at least $(1 - (i+1)2^{-n})^{T-i} + i^2 2^{-(n+1)}$

Proof sketch. For the lower bound on the required number of oracle calls, consider the process of computing $f^{(i)}(x)$ and let $W_i = \{f^{(j)}(x) | 0 \leq j \leq i\}$ be the at most $i + 1$ distinct values involved in that process. If an inverter does not obtain a value $f(w) \in W_i$, there is no correlation between the inverter and the evaluation process. If the inverter makes T calls to $f(w)$, the probability of doing this for a $w \in W_i$ is at most $(i + 1)T2^{-n}$, and this can be formalized.

To construct an inverter, first assume that the $i + 1$ values seen under the evaluation of $f^{(i)}(x)$ are distinct. This happens except with probability (over random f) $\binom{i}{2}2^{-n} \leq i^22^{-(n+1)}$ and if it does not happen we simply give up. Now consider the following inverter. It is given $y = f^{(i)}(x)$. Start by setting $x_0 = 0^n$ and $x_j = f(x_{j-1})$ for $j = 1, 2, \dots$. Continue this process until either $x_j = y$ (and it is done) or x_j is a value it has seen previously. In the latter case it changes x_j to a random value it has not seen previously and continues. Each value it sees is a random value and if it ever gets one of the $i + 1$ values in W_i , it finds the y within at most i additional evaluations of f . The probability of not finding such a good value in the $T - i$ first steps is at most $(1 - (i + 1)2^{-n})^{T - i}$. \square

We would therefore expect that the best achievable time over success ratio to invert a randomly chosen, iterated function is about $2^n/i$. If one thinks of i as relatively small (fixed), then an asymptotic (in n) argument that this is the “correct” complexity can be seen from [8]. Basically, our concern is the size of the image, $\text{Im}(f^{(i)}(x))$. In [8] it is shown that if f is a randomly chosen function, then the expected size of $\text{Im}(f^{(i)}(x))$ is $(1 - \tau_i)2^n$, where $\tau_0 = 0$, $\tau_i = e^{-1 + \tau_{i-1}}$. A Taylor expansion shows that $1 - \tau_i$ is $\sim i^{-1}$.

As mentioned, the one-way function we will consider is the Rijndael block cipher (viewed as encryption of a fixed message and thus the input that is used is the key and the output is the cipher-text).

Definition 4. A σ -secure one-way function is an efficiently computable function f that maps $\{0, 1\}^n \rightarrow \{0, 1\}^n$, such that the average time over success ratio for inverting the i th iterate is $\sigma 2^n/i$. That is, f cannot be (T, δ, i) -inverted for any $T/\delta < \sigma 2^n/i$.

A block cipher, $f(p, k)$, $|p| = |k| = n$, is called σ -secure if the function $f_p(k)$, for fixed, known plaintext p , is a σ -secure one-way function of the key k .

The security assumption we now will make is that $f = \text{Rijndael}$ [23, 22] is hard to invert, even when iterated. This is of course stronger than assuming that Rijndael itself is hard to invert, but since we use it in a way that does not necessarily give a permutation, it is from a theoretical point of view, [15], the weakest possible assumption.

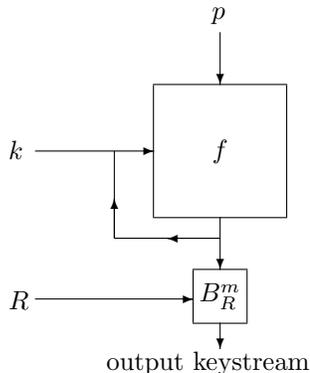
Quantitatively, we have seen (Theorem 3) that if one just picks a random function f , then on average, it will be 1-secure in the above terminology. It therefore seems reasonable to assume that Rijndael (which has been designed to be hard to invert), even if iterated a moderate number of times should be δ -secure, for some δ reasonably close to 1.

It is however important not to confuse the above assumption with “random behaviour” assumptions often made in cryptography. If we for instance assume

that f behaves like a random function (in “all” aspects), then it is trivial to construct a PRG, e.g. by running it in counter mode; $f(r), f(r+1), \dots$. We are assuming that f is about *as hard to invert* on its iterates as a randomly chosen function, a strictly weaker property.

2.4.1 Remark: A New Mode of Operation

Using a block cipher $f = f_p(k)$ as above in the $BMGL_{n,m,L}(f)$ construction can be viewed as a new mode of operation for Rijndael, in some sense a “dual” and more elaborate version of OFB (output feedback mode). The mode, which might be called KFB (key feedback mode) can be viewed as follows



2.4.2 Security of the generator

Our objective is to show that if $BMGL_{n,m,L}(f)$ is not (L, T, δ) -secure for “practical” values of L, T, δ , then there is also a practical attack on the underlying one-way function f . In particular, we show the following theorem:

Theorem 4. *Suppose that $G = BMGL_{n,m,L}(f)$ is based on an n -bit function f , computable by E operations, and that G produces L bits in time S . Suppose this generator can be (L, T, δ) -distinguished. Then, setting $\delta' = \frac{\delta m}{2L}$, there is an integer $i \leq L/m \triangleq \lambda$ such that f can be $(T', \delta'/4, i)$ -inverted, where T' equals*

$$\delta'^{-1}(8n+1)2^{m+3}(n[2m+T+3+\log(8n+1)+\log\delta'^{-1}]+E).$$

In particular, the time over success ratio is about $n^2m^{-1}2^mL^2\delta^{-2}T$. For any $\mu \in (0, 1)$, the value of i can, with probability at least $(1-\mu)^{\log\lambda}$, be found in time $12\lambda^2\delta^{-2}(T+S)\log\lambda\ln\mu^{-1}$.

(The approximate time over success ratio follows from neglecting the log-terms and making the very reasonable assumption that $T \geq E$.)

This result can in principle be obtained directly from the original works by Blum-Micali and Goldreich-Levin, but here we are interested in a tight result and hence we have to be more careful than in [11] were, basically, any polynomial time reduction from the inverting f to distinguishing the generator would be enough. Optimizations of the original proof also appeared in [16].

Lemma 5. *Let $L = \lambda m$. Suppose that $BMGL_{n,m,L}(f)$ runs in time $S(L)$. If this generator is not $(L, T(L), \delta)$ -secure, then there is an algorithm $P^{(i)}$, $1 \leq i \leq L/m$ that, using $T(L) + S(L)$ operations, given $f^{(i)}(x)$, R for random $x \in \mathcal{U}_n$ and $R \in \mathfrak{M}_m$, distinguishes $B_R^m(f^{(i-1)}(x))$ from \mathcal{U}_m with advantage $\delta' \geq \frac{\delta m}{2L}$.*

The algorithm $P^{(i)}$ depends on an integer i , and for any $0 < \mu < 1$, using $12\delta^{-2}\lambda^2 \log \lambda(T(L) + S(L)) \ln \mu^{-1}$ operations, i can be found with probability at least $(1 - \mu)^{\log \lambda}$.

Note: we have recently been able to improve the preprocessing step (finding i) in the above lemma, removing both log-factors. In fact, we conjecture that the new result is optimal up to constants. This improvement will of course have impact on the results derived from it. Details will be given later.

The current proof uses an optimized version of the so called *universality of the next-bit-test*, by Yao [28], see also [7].

Proof. For simplicity let us refer to our generator simply as G . Let D be an algorithm that uses $T(L)$ operations and distinguishes the output of G from a random L -bit string with advantage δ . Define the hybrid distributions H^i , $i = 0, 1, \dots, \lambda$, on $\{0, 1\}^L$ as follows. First, x_0, R are randomly chosen in $\{0, 1\}^n \times \mathfrak{M}_m$. Next, the first im bits of H^i are obtained as outcomes of random coin-flips. Then, to generate bits $im + 1, im + 2, \dots, L$, we apply the generator by setting $x_j = f^{(j)}(x_0)$, $j \geq i$, and let the j th m -bit block of H^i be $B_R^m(x_j)$.

Observe that H^0 equals the distribution of outputs of G , and that H^λ is the uniform distribution on $\{0, 1\}^L$. By assumption, D distinguishes H^0 and H^λ with advantage δ , so from the triangle follows the existence of an i , $0 \leq i < \lambda$, for which D distinguishes H^i from H^{i+1} with advantage at least δ/λ . We postpone the discussion how to find such an i for the moment.

The algorithm $P^{(i)}$ (depending on i) now works as follows. On input $(r, y = f^{(i)}(x), R) \in \{0, 1\}^m \times \{0, 1\}^n \times \mathfrak{M}_m$ it produces an element by first flipping im coins, appending r , and then iteratively applying f, B_R^m $\lambda - i$ times as in the generator construction. Call the generated element z and observe that if r is random, then z is an element from H^{i+1} , whereas if $r = B_R^m(f^{(i-1)}(x))$, z is from H^i . $P^{(i)}$ then feeds z to D and answers the same as D does. Clearly, $P^{(i)}$ distinguishes with the same advantage as D distinguishes H^i and H^{i+1} .

To find the right i , we perform a binary search and evaluate the alternatives by sampling. Due to sampling errors the quality of the found i can not be guaranteed to be exactly as good as the i in the existence proof above. Assume for notational simplicity that λ is a power of 2.

It must either be the case that D distinguishes between $H^0, H^{\lambda/2}$, or, between $H^{\lambda/2}, H^\lambda$ with advantage at least $\delta/2$. In general, for $j = 1, \dots, \log \lambda$, there must be an $l_j \in [0..2^j - 1]$ such that D distinguishes $H^{l_j 2^{-j} \lambda}, H^{(l_j+1) 2^{-j} \lambda}$ with advantage at least $\delta_j \triangleq 2^{-j} \delta$. We determine an l_j that does almost this well. The problem of finding the best l_j is that we have sampling errors. Let t be a parameter.

Set $l_0 \triangleq 0$. Repeat for $j = 1, \dots, \log \lambda$. We run D t times on $H^{(2l_{j-1}+1)2^{-j} \lambda}, H^{(2l_{j-1})2^{-j} \lambda}, H^{(2l_{j-1}+2)2^{-j} \lambda}$ and record the number of 1-answers as c, c_l and c_r ,

respectively. If $|c_r - c| \geq |c - c_l|$ we set $l_j = 2l_{j-1} + 1$ and otherwise $l_j = 2l_{j-1}$.

We say that the j iteration above is successful if for the l_j found we have that D distinguishes $H^{l_j 2^{-j\lambda}}$ and $H^{(l_j+1)2^{-j\lambda}}$, with advantage at least $(1 - 2^{j-(\log \lambda+1)})\delta_j$. If all iterations are successful, this gives us $\delta_{\log \lambda}$ as claimed. Note, that by assumption we start in a successful situation. We now estimate the probability that we are successful in iteration j .

Given success up to iteration $j - 1$, we know that for the true probabilities p_l and p_r of D of outputting 1 on $H^{l_{j-1} 2^{-(j-1)\lambda}}$ and $H^{(l_{j-1}+1)2^{-(j-1)\lambda}}$ we have $|p_l - p_r| \geq (1 - 2^{(j-1)-(\log \lambda+1)})\delta_{j-1}$. Now assume that one of the two choices for l_j is bad. Let p be the probability that D outputs 1 on $H^{(2l_{j-1}+1)2^{-j\lambda}}$. Clearly it is enough to estimate the probability that $l_j = 2l_{j-1}$ in the case when $p_l - p_r \geq (1 - 2^{(j-1)-(\log \lambda+1)})\delta_{j-1}$, and $p_l - p < (1 - 2^{j-(\log \lambda+1)})\delta_j$. We have to estimate the probability that $Y \triangleq c_l + c_r - 2c$ is positive. Clearly the expected value of this is $t(p_l + p_r - 2p)$ which by the above assumption equals

$$t(p_r - p_l) + 2t(p_l - p) > t2^{j-1-(\log \lambda+1)}\delta_{j-1} = \frac{t\delta}{2\lambda}.$$

To finalize the analysis, we use the following claim.

Claim 6. *Let X be a random variable with support on $[0, d]$, and expected value q . Then*

$$\mathbb{E}[e^{\gamma(X-q)}] \leq e^{d^2\gamma^2/8}.$$

Proof. Since we can divide X by d and multiply γ with d it is enough to prove the lemma for $d = 1$. Then, by convexity, the worst case is when $X = 1$ with probability q and $X = 0$ otherwise. In this case we have

$$\mathbb{E}[e^{\gamma(X-q)}] = qe^{\gamma(1-q)} + (1-q)e^{-\gamma q} \leq e^{\gamma^2/8},$$

where the last inequality is Lemma A.1.6 of [1]. \square

We above have a sum, Y , of $2t$ independent variables with $d = 1$ and t with $d = 2$. Its expected value is $q = \frac{t\delta}{2\lambda}$, so we have

$$\mathbb{E}[e^{\gamma(Y-q)}] \leq e^{3t\gamma^2/4}$$

and hence setting $\gamma = -\frac{2q}{3t}$ we get

$$\Pr[Y \leq 0] \leq e^{-\frac{q^2}{3t}}.$$

So in our case, $t \triangleq 12\delta^{-2}\lambda^2 \ln \mu^{-1}$ is sufficient to obtain the claimed error bound.

To analyze the running time is straightforward since we have t calls to G and D for each step in binary search. \square

We now reprove the theorem of Goldreich and Levin trying to be careful with our estimates and construction.

Theorem 7. *Suppose there is an algorithm, P , that using T operations given random $R \in \mathfrak{M}_m$, distinguishes $B_R^m(x)$ from random strings of length m with advantage at least ϵ where ϵ is given. Then we can in time*

$$(8n + 1)\epsilon^{-2}2^m(2m + \log((8n + 1)\epsilon^{-2}) + 2 + T)n$$

produce a list of $(8n + 1)2^m\epsilon^{-2}$ values such that the probability that x appears in this list is at least $1/2$.

Before we give the proof, some additional preliminaries. Let $\text{bin}(i)$ be the map that sends the integer i , $0 \leq i < 2^m$ to its binary representation as an m -bit string.

In the sequel, we perform some computations in \mathbb{F}_{2^k} , the finite field of 2^k elements, represented as $\mathbb{Z}_2[t]/(q(t))$ where $q(t)$ is a polynomial of degree k , irreducible over \mathbb{Z}_2 . (Computationally, for the practical values of k that will be of interest to us, we may assume that such q is available by table look-up and if really needed, we can easily find a q with an expected number of k^4 operations.) Viewing \mathbb{F}_{2^k} as a vector space over \mathbb{F}_2 , for any $\gamma = \sum_{i=0}^{k-1} \gamma_i t^i \in \mathbb{F}_{2^k}$, we let in the natural way $\text{bin}(\gamma)$ denote the vector $(\gamma_0, \dots, \gamma_{k-1})$ corresponding to γ 's representation over the standard polynomial basis. Note also that $\text{bin}(\gamma)$ can be interpreted as a subset of $[0..k-1]$ in the obvious way.

Lemma 8. *Fix any $x \in \{0, 1\}^n$. For $m < k$, from $m + k$ randomly chosen a_0, \dots, a_{m-1} and $b_0, \dots, b_{k-1} \in \{0, 1\}^n$, it is possible in time $2m2^k + k^2 + m + 4k$ to generate a set of $s = 2^k$ uniformly distributed, pairwise independent matrices $R^1, \dots, R^s \in \mathfrak{M}_m$. Furthermore, there is a collection of $m \times (m + k)$ matrices $\{M_j\}_{j=1}^{2^k}$ and a vector $z \in \{0, 1\}^{m+k}$ such $B_{R_j}^m(x) = M_j z$ for all j .*

The construction is similar to that of Rackoff in his proof of the Goldreich-Levin theorem, see [9, 10].

Proof. Choose randomly and independently m strings, a_0, a_1, \dots, a_{m-1} and k strings b_0, \dots, b_{k-1} , each of length n . The j th matrix, R^j is now defined by $\{a_i\}$, $\{b_l\}$, and an element $\alpha_j \in \mathbb{F}_{2^k}$ as follows. Its i th row, R_i^j , $0 \leq i < m$, is defined by

$$R_i^j \triangleq a_i \oplus (\oplus_{l \in \text{bin}(\alpha_j \cdot t^i)} b_l),$$

where α_j is the lexicographically j th element of \mathbb{F}_{2^k} (this is simply the lexicographically j th binary string), and the multiplication, $\alpha_j \cdot t^i$, is carried out in \mathbb{F}_{2^k} , and \oplus is bitwise addition mod 2.

Clearly the matrices are uniformly distributed, since the a_i and b_l are chosen at random. To show pairwise independence it suffices to show that an XOR of any subset of elements from any two matrices is unbiased. Since the columns are independent, it is enough to show that the XOR of any non-empty set of rows from two distinct matrices R^{j_1} and R^{j_2} is unbiased. Take such a set of rows, $S_1 \subset R^{j_1}$, and $S_2 \subset R^{j_2}$. We may actually assume that $S_1 = S_2 = S$, say, since otherwise, the a -vectors makes the XOR uniformly distributed. Thus, the XOR can be written

$$\oplus_{i \in S} \oplus_{l \in \text{bin}((\alpha_{j_1} + \alpha_{j_2}) \cdot t^i)} b_l,$$

but this is the same as

$$\bigoplus_{l \in \text{bin}((\alpha_{j_1} + \alpha_{j_2}) \cdot (\sum_{i \in S} t^i))} b_l,$$

which is unbiased if, and only if, $\text{bin}((\alpha_{j_1} + \alpha_{j_2}) \cdot (\sum_{i \in S} t^i)) \neq 0$. However, $\sum_{i \in S} t^i \neq 0$, and as $\alpha_{j_1} \neq \alpha_{j_2}$, $\alpha_{j_1} + \alpha_{j_2} \neq 0$ too, so we have two nonzero elements and hence their product is nonzero.

Notice that if we know $\sum_i a_i x_i$ and $\sum_i b_i x_i \pmod 2$ for all a_i, b_i (a total of $m + k$ bits, the l th choice of which will be z_l), then by the linearity of the above construction, we also know the matrix-vector products $R^j x$ for all j . To calculate all the matrices we first compute the reduction of t^i for all $i = k + 1, \dots, 2k$ in $GF[2^k]$. Using an iterative procedure this can be done with $3k$ operations on k bit words and since we only care about $k \leq n$ these can be done in unit time. Now generate the vectors a and b in time $m + k$ operations. Then we compute $\bigoplus_{l \in \text{bin}(t^i)} b_l$ for each $i = 0, \dots, 2k$ using k^2 operations. By using a gray-code construction each row of a matrix can now be generated with two operations and thus the total number of operations is $2m2^k + k^2 + m + 4k$. \square

The requirement $m < k$ is really not essential for the applications we have in mind for the lemma. We will only use it to guarantee the existence of *at least* 2^k matrices with the above properties. Technically, if $k < m$, we can carry out the same details, replacing k by $m + 1$ in the proof and then simply not use the $2^{m+1} - 2^k$ “extra” matrices. However, as will be seen, the interesting uses of the lemma is when $k > m$.

Before we go on, recall that any function $g : \{0, 1\}^m \rightarrow \{-1, 1\}$ can be expressed as a linear combination over the orthonormal set of functions $\{\chi_u(z) = (-1)^{\langle u, z \rangle_2}\}$, where $\langle u, z \rangle_2 = \sum_{i=0}^{m-1} u_i x_i \pmod 2$. More precisely,

$$g(z) = \sum_{u=0}^{2^m-1} (-1)^{\langle z, u \rangle_2} \hat{g}_u, \quad (1)$$

where $\hat{g}_u \triangleq \mathbb{E}_v[\chi_u(v)g(v)] = 2^{-m} \sum_v (-1)^{\langle u, v \rangle_2} g(v)$. In other words, this is the discrete Fourier series expansion of g .

Recall that the Fourier transform of 2^t elements can be computed in time $t2^t$ by the following observations:

$$\hat{g}_u = \sum_{v=0}^{2^{t-1}-1} (-1)^{\langle u, v \rangle_2} g(v) + \sum_{v=2^{t-1}}^{2^t-1} (-1)^{\langle u, v \rangle_2} g(v), \quad (2)$$

and if $u' = u + 2^{t-1}$ then similarly:

$$\hat{g}_{u'} = \sum_{v=0}^{2^{t-1}-1} (-1)^{\langle u, v \rangle_2} g(v) - \sum_{v=2^{t-1}}^{2^t-1} (-1)^{\langle u, v \rangle_2} g(v), \quad (3)$$

neglecting the constant factor 2^{-t} . We thus compute two sub-transforms of half the size by conditioning on the most significant bit of v , and from the

sign, determined by the msb of u , we can combine the solution for the two subproblems. For convenience, we shall therefore sometimes change the range of function from $\{0, 1\}$ to $\{-1, 1\}$ in the natural way.

Lemma 9. *Let P be an algorithm, mapping pairs $\mathfrak{M}_m \times \{0, 1\}^m \rightarrow \{-1, 1\}$, whose running time is T , let R^j, M_j be the matrices generated as described in Lemma 8 and let S be an arbitrary matrix in \mathfrak{M}_m .*

In time $2^{m+k}(2m+k+T)$ it is possible to compute 2^{m+k} values, $c_1, \dots, c_{2^{m+k}}$ such that for at least one l ,

$$c_l = E_j[P(R^j + S, B_{R^j}^m(x))].$$

The value of l is independent of S .

(The role of the matrix S may seem unclear at this point, but will be explained shortly.)

Proof. First run P on all the 2^{m+k} possible inputs of form $(R^j + S, r)$ and record the answers: $\{P(R^j + S, r)\}$. A fixed value of l above corresponds to a value of the $m+k$ bits z_l in Lemma 8. Let us assume that z_l is the correct choice, i.e. $B_{R^j}^m(x) = M_j z_l$. Then, by construction,

$$c_l \triangleq 2^{-k} \sum_{j=0}^{2^k-1} P(R^j + S, M_j z_l) = 2^{-k} \sum_{j=0}^{2^k-1} \sum_{r=0}^{2^m-1} P(R^j + S, r) \Delta(r, M_j z_l), \quad (4)$$

where $\Delta(r, r') = 1$ if $r = r'$ and 0 otherwise. The naive way to calculate this number would require time 2^{2k+m} and we want to do better using the Fast Fourier transform. First note that

$$\Delta(r, r') = 2^{-m} \sum_{\alpha \subseteq [0..m-1]} (-1)^{\langle r \oplus r', \alpha \rangle_2}.$$

This implies that the sum (4) equals

$$\begin{aligned} c_l &= 2^{-(m+k)} \sum_{j, r, \alpha} P(R^j + S, r) (-1)^{\langle r \oplus M_j z_l, \alpha \rangle_2} \\ &= 2^{-(m+k)} \sum_{j, \alpha} (-1)^{\langle M_j z_l, \alpha \rangle_2} \sum_r P(R^j + S, r) (-1)^{\langle r, \alpha \rangle_2}. \end{aligned}$$

Let $Q(j, \alpha)$ be the inner sum. Fix a value of j . The different α -values then correspond to a Fourier transform (on m -bit quantities, $m < n$) and hence the 2^m different $Q(j, \alpha)$ can be calculated in time $m2^m$. Thus, all the numbers $Q(j, \alpha)$ can be computed in time $m2^{k+m}$. Finally we have

$$c_l = 2^{-(m+k)} \sum_{j, \alpha} (-1)^{\langle M_j z_l, \alpha \rangle_2} Q(j, \alpha) = 2^{-(m+k)} \sum_{j, \alpha} (-1)^{\langle z_l, M_j^T \alpha \rangle_2} Q(j, \alpha),$$

where M_j^T is the transpose. Comparing this to (1), (2), and (3) above, this is just a rearrangement of a standard Fourier-transform of size 2^{k+m} and can be

computed with $(k+m)2^{k+m}$ operations. (We here allow ourselves to perform operations on $m+k$ bit quantities in unit time, motivated by the fact that we will have $m, k < n$.) The lemma follows. \square

Now we prove that we can compute useful information about x .

Lemma 10. *Let P, T, x be as in Theorem 7 and let t be a parameter such that $2^k = t\epsilon^{-2}$ for some $k \geq m$. Then for any set of N vectors $\{v_i\}_{i=1}^N \subset \{0, 1\}^n$ we can in time $2^{m+k}(2m+k+T+2)N$ produce a set of lists $\{b_i^{(j)}\}_{i=1}^N$, $j = 1, 2, \dots, 2^{k+m}$ such that with probability $1/2$ we have for at least one j , $\langle x, v_i \rangle_2 = b_i^{(j)}$, except for at most $\frac{8N}{t}$ of the N values of i .*

Proof. Start by randomly generating the 2^k matrices $\{R^j\}$ as described in Lemma 8. Now repeat the process below for each $i = 1, \dots, N$.

Select a random string $s_i \in \{0, 1\}^m$, and let S_i be the $m \times n$ matrix defined by $S_i \triangleq s_i \otimes v_i$ (the outer product²). Notice that by linearity

$$(R^j + S_i)x = R^j x + s_i \langle v_i, x \rangle_2, \quad (5)$$

which is $B_{R^j}^m(x)$ if $\langle v_i, x \rangle_2 = 0$, and a random string otherwise.

As described in Lemma 9, we now compute the values $\{c_l^i\}$.

$$c_l^i = 2^{-k} \sum_{j=0}^{2^k-1} P(R^j + S_i, M_j z_l).$$

Focus on the correct choice for l . If $\langle v_i, x \rangle_2 = 0$, then c_l^i is the outcome of a uniform random, pairwise independent sample of the distinguisher P on inputs of the form $\{P(R, B_R^m(x))\}$. On the other hand, if $\langle v_i, x \rangle_2 = 1$, it is a sample of $\{P(R, u)\}$ over random u .

Let p_1 be the probability that P outputs 1 on $(R, B_R^m(x))$ and p_2 the probability that it outputs 1 on (R, u) , where $|p_2 - p_1| \geq \epsilon$. We in fact assume that $p_2 = \frac{1}{2}$ and that $p_1 \geq \frac{1}{2} + \epsilon/2$. If not, modify P to $P'(R, r)$ which outputs $P(R, r)$ and $1 - P(R, r')$, for $r' \in \mathcal{U}_m$, also with probability $1/2$. This reduces the distinguishing advantage as above, but simplifies the rest of the analysis.

Let $p \triangleq \frac{1}{2} + \epsilon/4$. Then we would simply guess that $\langle v_i, x \rangle_2 = 0$ if $c_l^i \geq p$ and $\langle v_i, x \rangle_2 = 1$ otherwise. The choice is correct unless the average of 2^k pairwise independent Boolean variables is at least $\epsilon/4$ away from its mean. By Chebychev's inequality the probability this happens is bounded by $4 \cdot 2^{-k} \epsilon^{-2} = 4t^{-1}$.

This implies that for the correct value of l the expected number of errors is $4N/t$, and by Markov's inequality, with probability at least $1/2$ it is below $8N/t$. Trying all these alternatives for l gives the lists described in the lemma.

The total running time is now $2^k[2^m(2m+k+T)N + 2m] + k^2 + m + 4k$, which for m, k of interest can be bounded by $2^{k+m}(2m+k+T+2)N$. \square

Let us next establish Theorem 7.

² $(S_i)_{k,l} = (s_i)_k \cdot (v_i)_l$

Proof of Theorem 7. We apply Lemma 10 with $t = 8n + 1$, $N = n$, and set the vectors $\{v_i\}_{i=1}^n$ to be the unit vectors so that $\langle v_i, x \rangle_2$ gives x_i , the i th bit of x . With probability $1/2$ one of the lists give all the inner-products correctly and hence determine x . \square

We use this to establish Theorem 4 but we first point out that just as for the proof of Lemma 5, the proof techniques below have also recently been improved, resulting in a better bound in Theorem 4. Details will be given later.

Proof of Theorem 4. First we apply Lemma 5 to get an i for which we have an algorithm that when given $f^{(i)}(x)$ runs in time $S(L) + T(L)$ and distinguishes $B_R^m(f^{(i-1)}(x))$ from random bits with advantage at least $\delta' \triangleq \frac{\delta m}{2L}$. Since δ' is an average over all x we need to do some work before we can apply Theorem 7.

For each x we have an advantage δ_x and we know that the expected value of δ_x is at least δ' . Let $\mathcal{G} \triangleq \{x \mid \delta_x \geq \delta'/4\}$, the set of *good* x . One way to proceed would now be to observe that by Markov's inequality, at least a $3\delta'/4$ fraction of all x are good, and we are likely to succeed on each good x , but would simply give up on "bad" x . However, in the extreme case that precisely this minimal fraction of x is good, then the average over the bad x must also be $\delta'/4$, and we would be giving up to easy on the bad ones. Instead, of doing this very coarse division into good and bad x , we will take a more fine-grained approach, for each x guessing how good it is, leading to a better overall success rate. Formally, we proceed as follows.

Choose a random value of $j \geq 1$ where a specific value j_0 is chosen with probability 2^{-j_0} . If $2^{-j_0} \geq \delta'/4$ then we apply Theorem 7 with $\epsilon = 2^{-j_0}$ while if $2^{-j_0} < \delta'/4$ we give up. Let $p_x = 1$ if we by this retrieve an inverse image to $f^{(i)}(x)$, 0 otherwise.

To analyze this procedure, first observe that for $x \in \mathcal{G}$, the preconditions for Theorem 7 are satisfied if $\delta_x \geq 2^{-j_0} \geq \delta'/4$, and in this case the algorithm finds a preimage with probability at least $1/2$. The above condition is true with probability (over j_0) at least $\delta_x - \delta'/2$. For $x \notin \mathcal{G}$ we accept the possibility of failure, but still $\mathbb{E}_{x \notin \mathcal{G}}[p_x] \geq 0 > \mathbb{E}_{x \notin \mathcal{G}}[\delta_x - \delta'/2]$. This implies that the overall probability of success is

$$\begin{aligned} \mathbb{E}_x[p_x] &= \mathbb{E}_{x \in \mathcal{G}}[p_x] \Pr_x[x \in \mathcal{G}] + \mathbb{E}_{x \notin \mathcal{G}}[p_x] \Pr_x[x \notin \mathcal{G}] \\ &\geq \frac{1}{2} \mathbb{E}_{x \in \mathcal{G}}[\delta_x - \delta'/2] \Pr_x[x \in \mathcal{G}] + \mathbb{E}_{x \notin \mathcal{G}}[\delta_x - \delta'/2] \Pr_x[x \notin \mathcal{G}] \\ &\geq \frac{1}{2} \mathbb{E}_x[\delta_x - \delta'/2] = \frac{1}{2} (\mathbb{E}_x[\delta_x] - \delta'/2) \geq \delta'/4. \end{aligned}$$

To evaluate the expected running time, we have

Claim 11. *The expected running time of the algorithm in Theorem 7, when run with parameter $\epsilon = 2^{-j}$ with probability 2^{-j} provided $2^{-j} \geq \delta'/4$, is bounded by*

$$\delta'^{-1} (8n + 1) n 2^{m+3} (2m + T + 3 + \log(8n + 1) + \log \delta'^{-1})$$

in our previous notation.

Proof. We need to bound the expected value, when $\epsilon = 2^{-j}$ with probability 2^{-j} , of $t\epsilon^{-2}2^m(2m + \log(t\epsilon^{-2}) + T + 2)N$ (where $N = n$, $t = 8n + 1$) truncating at $j_0 - 1$, where j_0 is the smallest integer satisfying $2^{-j_0} < \delta'/4$. Rewriting this, collecting powers of ϵ^{-1} and $\log \epsilon^{-1}$ we get

$$tN2^m([2m + T + 2 + \log t]\epsilon^{-2} + \epsilon^{-2} \log \epsilon^{-2}).$$

First note that

$$\mathbb{E}_\epsilon[\epsilon^{-2}] = \sum_{j=1}^{j_0-1} 2^j < 2^{j_0} \leq 8\delta'^{-1}.$$

Then

$$\mathbb{E}_\epsilon[\epsilon^{-2} \log \epsilon^{-2}] \leq 8\delta'^{-1} \log(\delta'^{-1}) + 4.$$

Thus the expected running time is as claimed. \square

We then finally need to try all candidate x in the list, applying f to each. The expected length of the list is

$$(8n + 1)2^m \mathbb{E}_\epsilon[\epsilon^{-2}] = (8n + 1)2^{m+3}\delta'^{-1}.$$

Adding this gives the running total time as stated. \square

Instead of applying Lemma 10 with the unit vectors we can, as suggested in [10], use it with $\{v_i\}$ describing the words of an error correcting code. (Similar ideas appears in [15].) If we have code words of length N , containing n information bits, and we are able to efficiently correct e errors we get:

Theorem 12. *Fix x . Suppose there is an algorithm, P , that using T operations given R distinguishes $B_R^m(x)$ from random strings of length m with advantage ϵ where ϵ is given. Suppose further we have a linear error correcting code, C , with n information bits, N message bits that is able to correct e errors in time T_C . Then we can in time*

$$\frac{N}{e}\epsilon^{-2}2^{m+3}([2m + \log(N\epsilon^{-2}/e) + T + 5]N + T_C)$$

produce a list of $\frac{2^{m+3}N}{e}\epsilon^{-2}$ numbers such that the probability that x appears in this list is at least $1/2$.

Proof. We apply Lemma 10 with $t = 8N/e$ and $\{v_i\}_{i=1}^N$ as the row vectors of the generator matrix for C . This produces $\frac{2^{m+3}N}{e}\epsilon^{-2}$ vectors $\{c_j\} \subset \{0, 1\}^N$, such that with probability at least $1/2$, one c_j is at Hamming distance at most e from the correct codeword corresponding to x . Running the decoding algorithm on each c_j then produces a list as claimed. \square

In much the same way as the proof of Theorem 4, this translates to the quality of the inverter.

Theorem 13. *Suppose we have a linear error correcting code with n information bits, N message bits that is able to correct e errors in time T_C and that $G = BMGL_{n,m,L}(f)$ is based on an n -bit function f , computable by E operations, and that G produces L bits in time S . If G can be (L, T, δ) -distinguished then, with $\delta' = \frac{\delta m}{2L}$, there is an $i \leq L/m \triangleq \lambda$ such that f can be $(T', \delta'/4, i)$ -inverted where T' equals*

$$\delta'^{-1} \frac{N}{e} 2^{m+6} ([2m + T + \log(N/e) + \log \delta'^{-1} + 6]N + E + T_C).$$

In particular, the time over success ratio is about $\frac{N^2}{e} m^{-1} 2^m L^2 \delta^{-2} (T + T_C/N)$. For any $\mu \in (0, 1)$, the value of i can, with probability at least $(1 - \mu)^{\log \lambda}$, be found in time $12\lambda^2 \delta^{-2} \log \lambda (T + S) \ln \mu^{-1}$.

2.4.3 Concrete Examples

What does all this say? Suppose that we want to generate $L = 2^{30}$ bits (1Gbit), applying our construction with $m = 40$ (output 40 bits per iteration).

Corollary 14. *Consider $G = BMGL_{256,40,2^{30}}$ (Rijndael) (using key/block length 256) and where Rijndael is computable by E operations, and assume that G runs in time $S \sim EL/m$. If G can be $(2^{30}, T, 2^{-32})$ -distinguished, then there is an $i < 2^{30}/40$ so that Rijndael can be (T', δ', i) -inverted where $\delta' \approx 2^{-59}$, and the running time T' is composed of approximately 2^{120} applications of the distinguisher, about 2^{112} applications of Rijndael and about 2^{128} additional operations. The value of i can be found (with probability at least 0.65) using about 2^{124} applications of G and the distinguisher.*

Making the additional assumption that $T \leq S$, the pre-processing to find i is about 2^{156} , and once i has been found, the time over success ratio is about 2^{211} . In light of Definition 4, Rijndael can not be 2^{-21} -secure.

Recall that we would normally expect f to be 1-secure.

Proof. Apply Theorem 4, setting $\mu = \frac{\ln(4/3)}{\ln \lambda}$. Since G iterates f $i = L/m$ times, Theorem 3 suggests that the best possible time over success ratio would be about 2^{231} , which contradicts any belief that Rijndael is 2^{-21} -secure. \square

Comment: The assumption $S = EL/m$ is natural as the output generation should not be significantly more expensive than computing Rijndael. The assumption $T \leq S$ is, again, motivated by considering “practical” tests a la Diehard or those by Knuth.

If more than a single inversion is to be performed, it is possible to improve the above Corollary by using a good error correcting code and applying Theorem 13 instead. In particular, using a certain *Goppa-code*, see [17], for the same n, L, m and assumed distinguisher performance as above, once the value of i is found, we can reduce the workload per inversion by approximately a factor of 8. Still, the pre-processing time needed to find i is not affected by the use of the code,

and the achieved time-over-success ratio for a single inversion is more or less the same and we here omit further details.

A smaller m gives higher security. Let us see how security varies with m, δ :

Corollary 15. *Consider $G = BMGL_{256,4,2^{30}}$ (Rijndael) (generating 1Gbit, extracting 4 bits per iteration of Rijndael) where Rijndael is computable by E operations. If G can be $(2^{30}, T, 2^{-40})$ -distinguished, then there is an $i < 2^{28}$ so that Rijndael can be (T', δ', i) -inverted where $\delta' > 2^{-71}$, and the running time T' is composed of approximately 2^{95} applications of the distinguisher, about 2^{87} applications of Rijndael and about 2^{103} additional operations. The value of i can be found (with probability at least 0.65) using about 2^{146} applications of G and the distinguisher.*

Making the additional assumption that $T \leq S$, the time to find i is about 2^{182} , and once i is found, the time over success ratio is about 2^{201} . In light of Definition 4, Rijndael can not be 2^{-30} -secure.

The proof is as above. We give another example in the next section where we discuss methods of reducing the seed size and how they affect security.

3 Discussion

3.1 Issues

3.1.1 Choice of $f = \text{Rijndael}$

The reasons for choosing Rijndael as the cryptographic core are several:

- it is widely believed to be secure,
- it is efficient,
- as our construction requires that the block size of the cipher is equal to the key size, the fact that Rijndael supports both 128 and 256-bit block size is advantageous, as it makes it possible to vary the security parameter (key size).

Again, note that the one-way function we are suggesting to use is to have a fixed message, p , and let the input be the encryption key, k , and the output the cipher-text. It might appear that it would be better to have the mapping from clear-text to crypto-text as our f since this *is* a permutation. We would then iterate $f_k(p)$ rather than $f_p(k)$. This would therefore also eliminate the need to perform the key-scheduling on each iteration. The problem is that this is by definition not a one-way function since anybody that can compute it can also invert it. We are unable to get any provable properties when f is used in this way.

3.1.2 Number of bits output per application of the core

We would for efficiency reasons like to output as many bits as possible per application of Rijndael. On the other hand, we cannot output too many, if we are to relate the security of Rijndael to the proof of security for the generator. The effect of varying m is clearly visible in the above theorems. With a 256-bit key-size m in the range 30–50 seems to be acceptable and we have decided to recommend $n = 256$, $m = 40$. Test implementation has shown that going beyond $m = 80$ gives almost no speed up as the output generation itself then becomes too expensive.

One problem with a large value of m is that size of the seed grows with m , we discuss this issue below.

3.1.3 The Role of R

As seen, the proof of the Goldreich-Levin theorem does not assume that R is kept secret, and in fact, R can be used as “free” random bits that could be output. It can clearly not hurt to hide R , nor can we *prove* that it increases security to do so.

In fact, in some applications we could think of R as being public, chosen in a certified random way. In this case it is not part of the keying material and since we start getting nontrivial bounds even for $n = 128$, we get the certified generator with short seed as mentioned in the introduction.

3.1.4 Keying/Seeding

With $n = 256$, you need $256(m + 1)$ bits of seeding material: 256 for the initial x_0 (the secret key), and $256m$ to specify the matrix R . This is 1312 bytes when $m = 40$. This is quite large, and below we discuss how to decrease this number.

3.1.5 Efficiency

The output generation, consisting of computing the m inner products mod 2 should not be a critical issue for the performance. It is true that most microprocessors do not have an instruction to compute inner products between registers, but it is quite easy to implement this operation fairly efficiently. One possible way could be to do the following.

We assume a 32-bit architecture. Precompute a 256-byte table, `tab`, where `tab[i]` holds the remainder modulo 2, of the number of 1s in the binary representation of i . Let \wedge , \oplus , and \gg denote bitwise AND, XOR, and right shift, respectively. The inner product of two 8-word (i.e. 256 bit) vectors `r[]`, `x[]` is done by

```
c = 0;
for i = 1 to 8 do
  c = c  $\oplus$  (r[i]  $\wedge$  x[i]);
c = c  $\oplus$  (c  $\gg$  16);
```

```

c = c  $\oplus$  (c  $\gg$  8);
return tab[c  $\wedge$  255];

```

That is, about 40 instructions suffices. Alternatively, the table look-up in the last line can be replaced by a “table” of size 16, stored in an integer:

```

c = c  $\oplus$  (c  $\gg$  4);
return (6996hex  $\gg$  (c  $\wedge$  15))  $\wedge$  1;

```

Without putting too much effort on optimizations, a C-implementation which runs at about 5 Mb/s (PIII, 650) has been done (with $m = 40$ as suggested).

3.1.6 Decreasing Seed Size

To determine an $m \times n$ matrix to be used as part of the seed, we need nm bits. We note that it is possible to reduce this to only n bits by choosing matrices R as follows. Consider the finite field of 2^n elements. Binary strings of length n are interpreted as elements of this field in the natural way. Define the function $h_A(x)$, mapping this field to itself, by $h_A(x) \triangleq Ax$, which can be represented as a matrix multiplication by a boolean $n \times n$ matrix, A' , where A' is uniquely determined by n bits (and the irreducible polynomial representing the field). Now define a function $r_A(x)$ by selecting any fixed set of m bits of Ax . This was in [21] shown to give hard core functions. That is, independently of m , n bits are always enough to specify the m -bit output function (as long as $m \leq n$).

There is however a drawback with this construction. In our current reduction we can show that a distinguisher for $B_R^m(x)$ with advantage δ , essentially enables us to predict $\langle v_i, x \rangle_2$ with the same advantage. This is then amplified using Lemma 8. Since the set of possible A suggested above is more restrictive, we are not able to prove the equivalent of Lemma 8 with this space of matrices. Still, it is possible to get the same kind of reduction, but the only way (we know) of doing this is via the so called *Computational XOR-Lemma*, [27]. Unfortunately, involving this lemma reduces the δ -advantage of the distinguisher to a $2^{-m}\delta$ -advantage for the predictor for $\langle v_i, x \rangle_2$. Nevertheless, for small m , when small seeds are of interest, this could be a possible trade-off one is willing to make.

Moreover, for a fixed desired security level and seed-size, we can to some extent now choose m to achieve this. Specifically, doing the same kind of argument as in §2.4.3 we can show that

Corollary 16. *Consider $BMGL_{256,8,2^{30}}$ (Rijndael) where we choose the matrix R as above (i.e. as 8 rows out of an 256×256 matrix determined by an element of $\mathbb{F}_{2^{256}}$), rather than as random 8×256 matrices. If G can be $(2^{30}, T, 2^{-32})$ -distinguished, then there is an $i < 2^{27}$ so that Rijndael can be (T', δ', i) -inverted where $\delta' \approx 2^{-70}$, and the running time T' is composed of approximately 2^{98} applications of the distinguisher, about 2^{90} applications of Rijndael and about 2^{105} additional operations. The value of i can be found (with probability at least 0.65) using about 2^{130} applications of G and the distinguisher.*

Making the additional assumption that $T \leq S$, the pre-processing to find i is about 2^{164} , and once i has been found, the time over success ratio is about 2^{194} . In light of Definition 4, Rijndael can not be 2^{-35} -secure.

Comparing this to $m = 40$ and the general way of seeding, we obtain the a slightly better level of security and reduce the seed/key size from 1312 to 64 bytes (of which 32 bytes are the secret key). The price we pay is the need to reduce m to 8, leading to a slow-down by an estimated factor of 2 to 3 times (we need 5 times more applications of Rijndael to generate the same length of the stream, but on the other hand, the output generation itself is now 5 times as fast).

An alternative, suggested in [11], is to pick R as a random Toeplitz matrix in which case $n + m - 1$ bits are sufficient. Also in this case we only know how to prove security through the computational XOR-lemma and hence also in this case we lose a factor 2^m in security for a fixed m .

3.2 Attacks

Though we have proven security, there are some “non-mathematical”, attacks that can always be launched that we next discuss.

3.2.1 Exhaustive Seeding- and State Search

Any keystream generator is of course vulnerable to this attack. Our proof above states that to some extent, such attacks are the only ones. A key length of 256 (or even 128) will thwart such attempts for the foreseeable future.

3.2.2 Chosen keys/seeds

We again refer to our formal analysis in the proofs above where we allow “experiments” with the generator in precisely this fashion.

3.2.3 Attacks Related to Implementation Weaknesses

The generator we suggest has the same sensitivity to implementation-dependent attacks as Rijndael. The only additional step is the output generation (inner products). Considering for instance timing attacks, if the output is generated by table look-up as suggested, this step should have a constant execution time.

3.2.4 Hidden Weaknesses

Again, the proofs completely exclude the possibility of hidden weaknesses.

3.3 Comparison to Counter Mode

One very simple alternative method to make a keystream generator from Rijndael would be to use it in counter mode where the keystream is computed as $f(c), f(c + 1), \dots$ where c is the secret key. Bellare et al., [3], show that if f is a pseudo random permutation (PRP), then running f in counter mode gives a provably secure cipher. Thus, based on this a assumption they get a very simple and efficient generator.

As already mentioned, our assumption that Rijndael remains one-way when iterated, is much weaker and hence it is not surprising that our construction is more complex and gives a less efficient generator.

One can argue that in the above mentioned result about counter mode you essentially assume that your primitive has the required property (being random) to begin with, while we have to produce randomness from the weaker property of being hard to invert.

Since our assumptions are strictly weaker we would expect that a cautious user who does not have extreme demands on the speed of the generator would find it a very useful alternative.

3.4 Keystream Synchronization

A common problem with synchronous streamciphers over unreliable transport media is the possible loss of synchronization. This is typically the case for packet switched communication where packets may be re-ordered or even lost. Fortunately, BMGL provides some means to solve this problem. Typically, the packets will contain synchronization data in the form of a sequence number. In our case, this data could be used by entering it into the plaintext block of Rijndael as an “IV”. The keystream would then be generated on a per-packet basis. Moreover, if several data flows belong to one and the same session, it could be possible to use one and the same key for all these flows, by entering some unique flow-identifier into the IV. Careful consideration must of course be taken, not to reuse keystream.

4 Summary and Conclusions

Due to failure of security solutions, there is a growing interest to bring solutions based on provable security into existing standards. Recent examples in the case of asymmetric encryption are OAEP and PSS, [4, 5], in RSA PKCS #1 and IEEE P1363. We believe that also in the symmetric case, the time has come to make provable constructions available. We have here given a provably secure construction of a synchronous keystream generator, using Rijndael as its core. The proof relates the security of the generator to that of Rijndael in a direct, quantitative way. Therefore, the effect of varying key/block-size for Rijndael translates directly into a “numeric value” for the security level of the generator.

The main strength lies in the fact that we are able to obtain a provable security with a simple construction, meaning that the cipher is completely practical. Moreover, unlike other constructions of similar complexity, we do not need to make strong assumptions concerning Rijndael, such as being a pseudo random permutation. All that is needed is that it remains hard to invert when iterated a moderate number of times.

The fact that the construction is generic makes it possible (at least in software implementations) to replace the core with something else, in the unlikely event that some weakness in Rijndael is found.

The generator should be attractive in applications where some “real” confidence in security is desired and one does not need real-time Gbit/s encryption. Another application area is to use it directly as a pseudo-random generator, perhaps for simulations or key generation for number-theoretic cryptography.

Acknowledgment. We would like to thank Bernd Meyer for very careful proofreading, and for supplying comments that greatly improved the readability of this document.

References

- [1] N. Alon and J. Spencer: *The Probabilistic Method*. (2nd ed.), John Wiley & Sons, 2000.
- [2] R. W. Baldwin: *Preliminary Analysis of the BSAFE 3.x Pseudo Random Number Generators*. RSA Laboratories Bulletin, no 8, September 1998.
- [3] M. Bellare, A. Desai, E. Jorjani, and P. Rogaway: *A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation*. Proceedings of the 38th IEEE FOCS, 1997.
- [4] M. Bellare and P. Rogaway: *Optimal Asymmetric Encryption—How to Sign with RSA*. Proceedings, Eurocrypt '94, pp. 92–112, Springer Verlag.
- [5] M. Bellare and P. Rogaway: *PSS: Provably Secure Encoding Method for Digital Signatures*. Submission to IEEE P1363.
- [6] L. Blum, M. Blum, and M. Shub: *A Simple Unpredictable Pseudo-random Number Generator*. SIAM Journal on Computing, **15**, no 2, 1986, 364–383.
- [7] M. Blum and S. Micali: *How to Generate Cryptographically Strong Sequences of Pseudo-random Bits*. SIAM Journal on Computing, **13**, no 4, 1986, 850–864.
- [8] P. Flajolet and A. Odlyzko: *Random Mapping Statistics*. Proceedings, Eurocrypt '89, LNCS 434, pp. 329–354, Springer-Verlag.
- [9] O. Goldreich: *Foundations of Cryptography (Fragments of a Book)*. Available on-line at <http://philby.ucsd.edu/criptolib.html> (Theory of Cryptography Library)
- [10] O. Goldreich: *Modern Cryptography, Probabilistic Proofs and Pseudo-randomness*. Springer-Verlag 1999.
- [11] O. Goldreich and L. A. Levin: *A Hard Core Predicate for any One Way Function*. Proceedings, 21st ACM STOC, 1989, pp. 25–32.
- [12] S. Goldwasser and S. Micali. *Probabilistic encryption*. J. Comput. Syst. Sci., **28**(2), 270–299, 1984.

- [13] J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby: *Pseudo Random Number Generators from any One-way Function*. SIAM Journal on Computing, **28** (1999), 1364–1396.
- [14] D. Knuth, *Seminumerical algorithms*, (2 ed.), Volume 2 of *The art of computer programming*, Addison-Wesley, 1982.
- [15] L. Levin: *One-way Functions and Pseudorandom Generators*. *Combinatorica* 7 (1987), 357-363.
- [16] L. Levin: *Randomness and Non-determinism*. *J. Symb. Logic*, **58**(3), 1102–1103, 1993.
- [17] F. J. MacWilliams and N. J. A. Sloane: *The Theory of Error Correcting Codes*. North-Holland, 1977.
- [18] G. Marsaglia: *The Diehard statistical Tests*.
<http://stat.fsu.edu/~geo/diehard.html>
- [19] U. Maurer: *A Universal Test for Random Bit Generators*. *Journal of Cryptology* **5**(2), 1992, 89–105.
- [20] NESSIE: *New European Schemes for Signatures, Integrity, and Encryption*.
<http://www.cryptonessie.org/>
- [21] M. Näslund: *Universal Hash Functions & Hard-Core Bits*. *Proceedings, Eurocrypt '95, LNCS 921*, pp. 356–366, Springer Verlag.
- [22] National Institute of Standards and Technology:
The Advanced Encryption Standard (AES) Homepage.
<http://csrc.nist.gov/aes/>
- [23] J. Daemen and V. Rijmen: *AES Proposal: Rijndael*. Available at [22].
- [24] R. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin: *The RC6 Block Cipher*. Available at [22].
- [25] P. Rogaway and D. Coppersmith: *A Software-Optimized Encryption Algorithm*. *Journal of Cryptology*, **11**(4), 273–287, 1998.
- [26] B. Schneier. *Applied Cryptography*. Jon Wiley & Sons.
- [27] U. V. Vazirani and V. V. Vazirani: *Efficient and Secure Pseudo-Random Number Generation*. *Proceedings, 25th IEEE FOCS, 1984*, pp. 458–463.
- [28] A. C. Yao: *Theory and Applications of Trapdoor Functions*. *Proceedings, 23rd IEEE FOCS, 1982*, pp. 80–91.

A Results from Statistical Tests

All the results below were obtained, using Rijndael with $n = 128$ bit block- and key size, and outputting $m = 64$ bits per iteration. Thus, we “stressed” the generator a bit by using the a combination of n and m that is beyond our previous recommendations for preserved security. This choice was partly due to time-constraints/NESSIE deadline. (Making these tests is quite time-consuming.) We will perform more extensive tests for different choices of the parameters later.

It should be noted that no fixed set of statistical can be taken as a proof of security/pseudo-randomness (but a truly bad generator would probably expose itself).

A.1 Diehard

The most widely used package when performing statistical tests is probably Marsaglia’s “Diehard”, [18], so we decided to run the generator against it.

For further comparison we also include some results when applying the tests to the Blum-Blum-Shub generator, $x_{i+1} = x_i^2 \bmod N$, for a 1024-bit N , extracting the 10 least significant bits of each x_i as generator output.

Each of the Diehard tests compute a p -value, in our case based on 80Mbits produced by the generator. This p should be uniformly distributed in $[0, 1]$. As in [2], each figure presented is based on the average over 10 runs. (Thus, each value is based on a total of 800Mbit of data.) Under the assumption of uniform distribution, the expected average should therefore be 0.5, and the expected standard deviation is $1/\sqrt{12} \approx 0.289$. For more details on the Diehard tests, we refer to [18] or [2]. The results are shown in Table 1, page 27.

A.2 The Classical Tests

We also performed the “classical” tests. For further description of the tests, see e.g. [14]. The tests are based on $N = 80$ Mbit of data with $m = 64$ as above.

Monobit bias. Computes the correlation between the sequence and the constant “1”, i.e. $2\omega/N - 1$, where ω is the number of 1s. Result: $-5.33 \cdot 10^{-5}$ (expected value 0, standard deviation for a random sequence $5.46 \cdot 10^5$).

Poker test. Divides the sequence into $\lfloor N/16 \rfloor$ non-overlapping 4-bit windows, and counts occurrences of the patterns $i = 0, \dots, 15$. We observed the maximum deviation from the expected value (1310720, std. dev 1109) for $i = 0$, with a count of 1312460.

Runs Test. Counts number of “runs” of consecutive 1s (or 0s) of length $L = 1, 2, \dots$. Results appear in Table 2.

Serial Test. Divides the sequence into (overlapping) 2-bit windows and count occurrences of each pattern. Expected value for each is 20971519. See Table 3.

Auto Correlation. Computes $2A(d)/(N - d) - 1$, where $A(d) = \sum_{i=1}^{N-d} s_i \oplus s_{i+d}$. That is, the correlation of the sequence with itself shifted d steps. The expected value for a random sequence is of course 0. Results are found in Table 4.

A.3 Maurer's Universal Test

Finally, we made some tests using Maurer's universal test. This test computes an "entropy measure", FTU, parameterized by values L, K , and Q . Basically, the test determines distances between occurrences of the same L -bit string. See [19] for details. We made tests for $2 \leq L \leq 13$, and $Q = 10 \cdot 2^L$ (K is then determined by the datasize). Results appear in Table 5 and are as above based on 80Mbit of data.

Test	Algorithm and bits per iteration		
		BMGL/Rinjdael, 64 bits	BBS, 10 bits
Birthday spacing	avg:	0.503	0.507
	sd:	0.293	0.318
5-perm	avg:	0.550	0.555
	sd:	0.324	0.339
Rank 31x31	avg:	0.541	0.541
	sd:	0.161	0.220
Rank 32x32	avg:	0.544	0.556
	sd:	0.229	0.194
Rank 6x8	avg:	0.475	0.527
	sd:	0.304	0.291
Missing 20b word	avg:	0.494	0.495
	sd:	0.278	0.289
Missing 10b pair	avg:	0.496	0.495
	sd:	0.281	0.296
Missing 5b quad	avg:	0.471	0.515
	sd:	0.281	0.280
Missing 2b tens	avg:	0.494	0.520
	sd:	0.290	0.286
Count-1 all bytes	avg:	0.443	0.605
	sd:	0.244	0.279
Count-1 specific	avg:	0.547	0.542
	sd:	0.277	0.288
Parking lot	avg:	0.530	0.516
	sd:	0.265	0.281
Min distance	avg:	0.544	0.567
	sd:	0.245	0.295
Min 3D sphere	avg:	0.496	0.467
	sd:	0.293	0.290
Squeeze iter.	avg:	0.357	0.459
	sd:	0.326	0.323
Overlap sums	avg:	0.473	0.523
	sd:	0.260	0.295
Up/down runs	avg:	0.581	0.498
	sd:	0.280	0.269
Craps game	avg:	0.449	0.538
	sd:	0.317	0.298

Table 1: Results from Diehard tests. Expected average and standard deviation for a true random source is 0.5, resp. ≈ 0.289 .

L	count	expected
1	20969263	$2.097 \cdot 10^7$
2	10484606	$1.049 \cdot 10^7$
3	5239897	$5.243 \cdot 10^6$
4	2621447	$2.621 \cdot 10^6$
5	1310423	$1.311 \cdot 10^6$
6	656400	655360
7	328313	327680
8	163704	163840
9	82643	81920
10	40810	40960
11	20526	20480
12	10239	10240
13	5124	5120
14	2568	2560
15	1291	1280
16	641	640
> 16	611	639

Table 2: Runs Test.

pattern	count
00	20976023
01	20969253
10	20969253
11	20971550

Table 3: Serial Test.

d	correlation
3	$-4.591 \cdot 10^{-5}$
4	-0.000166
5	$6.996 \cdot 10^{-5}$
6	$-3.862 \cdot 10^{-5}$
7	0.0001232
8	$-1.476 \cdot 10^{-5}$
9	-0.0001509
10	0.0001522
11	$4.816 \cdot 10^{-6}$
12	$-6.974 \cdot 10^{-5}$
13	$-6.416 \cdot 10^{-5}$
14	-0.0001280
15	0.0001565
16	$6.266 \cdot 10^{-5}$

Table 4: Auto Correlation.

L	fTU	expected	std. dev
2	1.53728	1.53744	$5.358 \cdot 10^{-5}$
3	2.40153	2.40161	0.000112987
4	3.31087	3.31122	0.00016766
5	4.25362	4.25343	0.000216834
6	5.21790	5.21771	0.000260509
7	6.19635	6.19625	0.000299273
8	7.18352	7.18367	0.000334009
9	8.17655	8.17642	0.00036577
10	9.17183	9.17232	0.000395495
11	10.1698	10.1700	0.000424214
12	11.1681	11.1688	0.00045283
13	12.1684	12.1681	0.000482347

Table 5: Results from Maurer's test.