LARGE-SCALE RANDOMIZATION TECHNIQUES

Neal R. Wagner[1]
Drexel University

Paul S. Putter
The Pennsylvania State University

Marianne R. Cain[1]
Drexel University

Abstract.

This paper looks at a collection of especially simple conventional cryptosystems that use a very large blocksize. One variation uses a single xor randomization followed by a single bit permutation. Tight upper and lower bounds are obtained on the number of bits of matching plaintext/ciphertext needed to break the systems. These results follow from two interesting combinatorial theorems. The cryptosystems are not practical because the number of bits above is about the same as the keysize. We can make the systems practical by introducing key-dependent pseudo-random numbers, though we then lose any proofs of the difficulty of cryptanalysis.

1. Introduction.

Recent work in cryptography has focused on proving that the difficulty of breaking a cryptosystem is equivalent to the difficulty of solving some other mathematical problem; for example, the problem of factoring composite integers.

We have been investigating two other approaches to conventional cryptography:

- Very simple encryption/decryption algorithms -- perhaps so simple that one can prove average-case lower bounds [Mas85] [Wag86].
- Very complex encryption/decryption algorithms -- perhaps so complex that mathematical analysis is not feasible [Wol85].

[1] Research supported in part by NSF grant DCR-8403350 and by a Research Scholar award from Drexel University.

A.M. Odlyzko (Ed.): Advances in Cryptology - CRYPTO '86, LNCS 263, pp. 393-404, 1987.

This article considers the first approach -- looking for simple algorithms. To compensate for the simplicity of the algorithm we propose using a very large blocksize, or even encrypting an entire file at once. We also use randomization.

As an example, we present a code that uses one randomization step and one bit-permutation with a blocksize in the range of 1K to 1M bits. We present the simple motivating examples and their generalizations in Section 2. Section 3 looks at cryptanalysis of these various systems. We are able to obtain tight bounds on the number of plaintext/ciphertext pairs needed to break the systems. These bounds show exactly how much matching plaintext/ciphertext is needed to break the system, but the systems investigated in Section 3 are not at all practical. In fact, the keysize must be very large -- equal roughly to the amount of information that must be transmitted before the system can be broken. Finally, Section 4 considers several practical variations using pseudo-random numbers. The pseudo-random number algorithm is not assumed to be at all secure.

Another example of this basic approach is considered in [Wag 86]. Here in what might be called *global randomization*, we encrypt an entire relation of a relational database as a unit.

2. The Rip van Winkel cipher and generalization.

2.1. The Rip van Winkel cipher.

An especially simple though impractical cipher called the *Rip van Winkel* cipher was recently introduced [Mas85]. It is interesting for a provable lower bound on the average amount of computation needed to break the system. The system is illustrated in Figure 1.

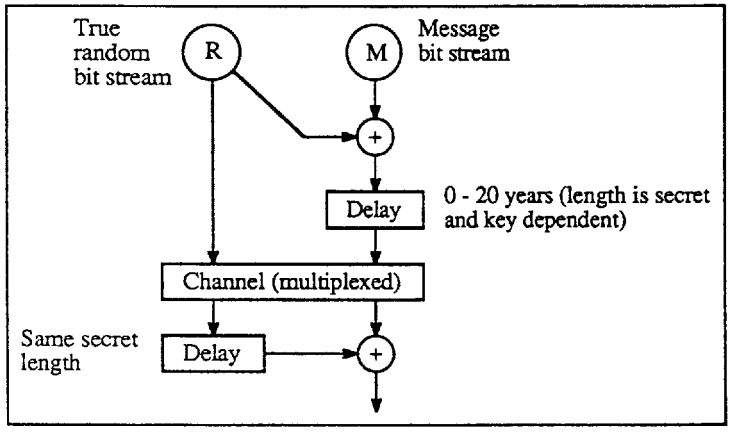

Figure 1. The Rip van Winkel cipher.

This cipher has the following interesting properties:

- The only secret is the *delay length*, a key-dependent number from 0 to the maximum delay.
- There is a provable average-case lower bound of $2\sqrt{N}$ comparisons needed to break the system, where N is the maximum delay length.
- Encryption and decryption require only a single xor per bit.
- The delay hardware must contain buffer storage.
- The system is completely impractical.

The rest of this paper looks at generalizations of this cipher.

2.2. Pseudo-random selection from queues.

During a study of randomization techniques [Wag85], we looked at ways to generalize the concept of concatenation to work with bit streams. One method employed a pseudo-random bit-sequence to control selection from two bit streams; the first, a random stream and the second, a randomization of a message stream (see Figure 2).

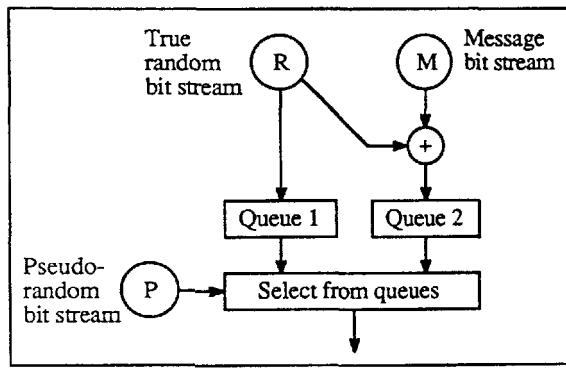

Figure 2. Pseudo-random selection from queues.

It is interesting that this system is a generalization of the Rip van Winkel cipher of Section 2.1. In fact, one can take P = "01010101..." and initialize Queue 2 to hold 0 - 20 years worth of true random bits.

Section 2.3 and 2.4 consider further generalizations of this system, while Sections 3.1 and 3.3 give cryptanalytic attacks on the generalizations.

2.3. Rip van Winkel in RAM.

In implementing some version of the Rip van Winkel cipher, efficiency considerations almost force one into the use of RAM storage. Once RAM storage is chosen, one might as well plan to use arbitrary locations within RAM. This leads to the type of generalization we consider in this section.

Start with m plaintext bits $P_1, P_2, \ldots, P_m$ and m true random bits $R_1, R_2, \ldots, R_m$. Use a RAM buffer C of size $n = 2m$. Initially load C with

$$C = (R_1, R_2, \ldots, R_m, R_1 \oplus P_1, R_2 \oplus P_2, \ldots, R_m \oplus P_m).$$

Then apply an arbitrary permutation of these n bits. This permutation is the encryption/decryption key, requiring $n\log_2 n$ bits of storage in the obvious represention.

Notice that any pair of bits R_i and $R_i \oplus P_i$ is a two-bit randomization of P_i, so that the xor of these two yields P_i. There is nothing that distinguishes the $R_i \oplus P_i$ bit from the R_i bit.

This is a randomized cryptosystem. There are $n!2^{n/2}$ distinct encryption functions and $n!2^{-n/2}$ distinct decryption functions. The average number of bits needed to represent a decryption key is

$$\log_2 (n!2^{-n/2})$$

or approximately (using Stirling's formula)

$$n\log_2 n - n\log_2 e - (1/2)n + (1/2)\log_2 n.$$

Thus we could represent the decryption key with slightly fewer than the $n\log_2 n$ bits needed in just listing n addresses, but any gain would go asymptotically to zero as n increases.

Note that there are

$$\frac{n!}{(n/2)!\, 2^{n/2}}$$

ways to choose pairs of bits, where each pair represents a randomization of a plaintext bit. For each choice of pairs, there are $(n/2)!$ permutations of the pairs, giving $n!2^{-n/2}$ decryption functions altogether.

Cryptanalysis of this cipher will be considered in Section 3.3.

2.4. The matrix cipher.

Suppose in the previous cipher we wish to protect against chosen plaintext attacks. We could add an extra randomization step, in which one xors the plaintext with new random bits *before* carrying out the encryption of Section 2.3. The extra random bits would be concatenated onto the ciphertext. More generally, we could think of xoring triples together – two random bits and one plaintext bit -- before applying the permutation. From here it is natural to think of xoring general subsets of bits. This leads to the general form

$$KC = P,$$

where

- $P = (P_i)$ is a $m \times 1$ matrix of *plaintext bits*,
- $C = (C_j)$ is an $n \times 1$ matrix of *ciphertext bits*,
- $K = (K_{ij})$ is an $m \times n$ matrix of *key bits*,
- $n \geq m$,
- all arithmetic is over GF(2), and
- the matrix K has rank n.

The matrix K is the *secret key*. For encryption, one solves m equations in the n unknowns $C_1, C_2, \ldots, C_n$. For decryption, just multiply K by C. If $n>m$, then encryption is randomized and non-linear, but decryption is a linear transformation. We are picturing n in the range from 1K to 1M bits. Assuming $n = 2m$, the keysize (size of matrix K) will be in the range from 500K to 500000M bits. Note that the Hill cipher [Den82] is a special case of this when $n = m$.

For cryptanalysis of this system, refer to Section 3.1.

3. Cryptanalysis -- upper and lower bounds.

3.1. The matrix cipher.

It is easy to break this system using enough known plaintext/ciphertext pairs. Suppose we have l ciphertexts $C_1, C_2, \ldots, C_l$ and corresponding plaintexts $P_1, P_2, \ldots, P_l$. Write the ciphertexts as an $n \times l$ matrix. We will be able to break the system once this matrix has rank n. Assuming the entries of $(C_1, C_2, \ldots, C_l)$ are random, Appendix I gives probabilities for the matrix to have rank n.

Theorem. Given n known plaintext/ciphertext pairs and assuming random ciphertexts, the probability of being able to break the system is 0.288.... After $n + 10$ pairs, the probability is 0.999....

See Appendix I for probabilities of being able to break the system after $n + i$ known plaintext/ciphertext pairs, $i = 0, 1, 2, \ldots$.

We can also state a lower bound.

Theorem. We must process at least m plaintext/ciphertext pairs on the average in order to break the matrix cipher.

Proof. This follows from an information theory argument. The keysize is mn. If $m = n$, only 29% of all matrices can serve as keys, but the percentage tends rapidly to 1 as n gets greater than m (see Appendix I). Thus the number of bits needed to represent the key is at least mn - 2. Each chosen of known plaintext/ciphertext pair adds at most n bits of information about the key. Thus m pairs must be processed before cryptanalysis is possible.

3.2. Bit permutations.

Assume $m = n$ and restrict to ciphers that are just permutations. Thus the matrix K is a permutation matrix -- exactly one 1 in each row and column.

Theorem. We can always uniquely determine a permutation of m bits using $\lceil \log_2 m \rceil$ chosen plaintext/ciphertext pairs, and no fewer number of pairs will suffice to uniquely determine a permutation.

Proof. The proof that $\lceil \log_2 m \rceil$ pairs will suffice is very simular to the Hamming code construction.

Next we show that at least $\lceil \log_2 m \rceil$ pairs are required. Suppose we have fewer than $\lceil \log_2 m \rceil$ plaintexts, say k of them. For each i, $1 \le i \le m$, regard the i^{th} positions of each plaintext as a k-bit number. There are m such numbers, but fewer than $\lceil \log_2 m \rceil$ bits. Thus we do not have enough bits for each of these m numbers to be distinct. In other words, there exist positions i and j such that each of the k plaintexts have the same value at i and j. Then given these k plaintexts and the corresponding ciphertext, we cannot tell whether the permutation leaves i and j fixed, or interchanges i and j, or does something more complex with positions i and j. Thus the permutation is not uniquely determined, completing the proof.

It is interesting to note that we lose guarenteed security just as we finish transmitting $m\log_2 m$ bits of plaintext, and $m\log_2 m$ is the keysize. So as we might expect, we would be better off just using a one-time pad. Later we will attempt to refine examples like the one in this section into a practical cryptosystem.

Suppose we have *known* plaintext/ciphertext pairs available, rather than *chosen* pairs.

Theorem. Let $P_{m,k}$ denote the probability that a permutation of m bits is uniquely determined by k random known plaintext/ciphertext pairs, i.e., the probability that k pairs will suffice to break the system. Then $P_{m,k}$ is the same as the number $p(m, 2^k)$ of Appendix I, the probability that m k-bit numbers will be distinct. In particular $P_{m,k} > 0.606$ for $k \ge 2\lceil \log_2 m \rceil$, and $P_{m,k} > 0.9995$ for $k \ge 2\lceil \log_2 m \rceil + 10$.

Proof. From Appendix I, once you believe that the m k-bit numbers must be distinct to uniquely determine a permutation.

3.3. The Rip van Winkel cipher in RAM.

Refer to Section 2.3 for the encryption/decryption algorithms.

Let us mount a chosen plaintext attack on this system using $P = (0\ 0\ \ldots\ 0)$, i.e., all m bits equal to 0. Repeated attacks will yield corresponding ciphertexts $C_1, C_2, \ldots$, each $n = 2m$ bits long. For a given C_l, each pair R_i and $R_i \oplus P_i$ of bits will be the same, either both 0 or both 1, whereas all other pairs have probability 1/2 of differing. For a given bit position i ($1 \le i \le n$) and for a given ciphertext C_l, we expect half of the other positions j will be eliminated as candidates for the position paired with i. Thus intuitively, for fixed bit position i, we expect with probability 1/2 to have identified the unique position paired with i after processing $\log_2 m$ of the C_l. For *all* bit positions i, we intuitively expect with probability 1/2 to have identified all pairs after processing $2\log_2 m$ of the C_l. This intuition turns out to be correct, as we show below.

We need an additional $\log_2 m$ chosen plaintext/ciphertext pairs to uncover the permutation of the pairs, using a variation of the Hamming code.

Theorem. The system described here can be broken with probability at least 0.606... using $3\lceil \log_2 m \rceil$ chosen plaintext/ciphertext pairs. With $3\lceil \log_2 m \rceil + 10$ pairs, the probability of breaking the system is greater than 0.9995.

Proof. Let Q_l be the n-bit vector obtained by applying the inverse of the encryption permutation to C_l. Then Q_l consists of a random m-bit string followed by the identical m-bit string. Suppose we have k such n-bit vectors. Look at the columns of the Q_l rather than the rows, so that we have m random k-bit quantities followed by a repetition of these. By an argument simular to that used for the Hamming code, these k ciphertexts will uniquely determine the pairings of bit positions if and only if the m k-bit numbers are all distinct. Thus from Appendix I, the probibility of uniquely determining the pairings becomes 0.606... for $k = 2\lceil \log_2 m \rceil$. The remainder of the proof is the same as the method for breaking a permutation cipher.

If we look at the proof, we see that for this type of attack we cannot do any better.

Theorem. Assuming we first determine the bit pairings using a chosen plaintext of all zero bits or all one bits and then the permutation of pairs, $3\lceil \log_2 m \rceil$ chosen plaintext/ciphertext pairs are necessary and sufficient to have probability 0.606... of breaking this system. Exact probabilities for breaking the system using this attack with fewer or with more chosen plaintext/ciphertext pairs are given in Appendix I for $m = 2^{10}$ and $m = 2^{20}$.

We are not able to prove the lower bound for an arbitrary chosen plaintext attack on this system, though we believe that no attack uses fewer pairs than the one we presented.

Notice that it seems much more difficult to break this system given only *known* plaintext/ciphertext pairs -- particularly if the plaintext contains roughly equal number of zero and one bits. In fact, the only attack we know of in this case is the one of Section 3.1, requiring at least n known plaintext/ciphertext pairs.

4. Practical variations using pseudo-random numbers.

4.1. Rip van Winkel in RAM.

For a RAM buffer of size n = 1M bits, the Rip van Winkel in RAM cipher introduced and studied in Sections 2.3 and 3.3 has two fatal flaws:

- The keysize is relatively large, namely $n\log_2 n$, or 20M bits = 2.5M bytes for n = 1M.
- Relatively few chosen plaintext/ciphertext pairs are needed to break the system with high probabilty, namely $3\log_2 n + 10$, or 70 for n = 1M. (However, it appears that *known* plaintext attacks would require many more pairs.)

We can make these difficulties go away by

- generating the bit permutation pseudo-randomly, using a key-dependent pseudo-random number generator, and
- changing the key to the pseudo-random number generator with each encryption step. (We can use some of the random bits used to encrypt the plaintext for the key for the next encryption.)

Several difficulties then emerge:

- We must choose a sutable key-dependent pseudo-random number generator.
- We no longer have a required $3\log_2 n$ chosen plaintext/ciphertext pairs in order to break the system. In fact, a single known plaintext attack might succeed using a brute-force approach and arbitrarily large computing resources.

Though we no longer have any provable security, we feel that almost any pseudo-random number generator with a long key and very long period should provide good security. For example, the Tausworth generator in [Bri79] with n = 521 has key (=seed) 521 bits long and period of 2^{521} 64-bit numbers. Since there is a key change at each stage, the opponent effectively has just one plaintext/ciphertext pair to work with. Best for an opponent would be the ciphertext corresponding to a chosen plaintext of all 0 bits (or all 1 bits), and this might be sufficient information to uniquely determine the key. However, the information is in a very diffuse and vague form that should make anything but brute-force cryptanalysis difficult. Basically, this information breaks the buffer into two subsets, and bit pairings are known to be confined to one or the other subset.

The actual algorithm is very simple to state:

```
(* Encryption algorithm. *)
(* initial load of buffer *)
    for i := 1 to m do
        begin
            C[i] := truerandom;
            C[i+m] := C[i] xor P[i]
        end;
(* pseudo-random permutation *)
for i := n downto 2 do
    interchange ( C[i], C[pseudorandom ( key, i )] )
```

Here C and P are bit arrays of ranges respectively 1 .. n and 1 .. m, where $n = 2m$. The procedure "truerandom" returns a true random bit and the procedure "pseudorandom" returns an integer in the range 1 .. i. The permutation algorithm is described in [Knu81], and it is easy to see that if this algorithm is supplied uniformly distributed random integers, then the permutations generated are also uniformly distributed. Notice that between the two halves of the algorithm, the key for the *next* encryption would be extracted from the lower half of the buffer.

This algorithm seems interesting for its simplicity. It requires very few computational resources per plaintext bit: 1 true random bit, 1 call to the pseudo-random number generator, 1 xor, and several accesses and stores. Thus a larger buffer size requires no extra computation per message bit, though a larger buffer means that a longer pseudo-random integer must be returned. (Of course a larger buffer increases the overhead for short messages, since the entire buffer must still be sent.)

Decryption can be carried out in general without much difficulty (see [Knu73] for an algorithm for inverting a permutation). Decryption becomes especially easy if the pseudo-random number generator will run backwards.

4.2. Other practical variations.

Many variations and extensions of the basic system presented in Section 4.1 are possible. If we have fewer than $m = n/2$ message bits, we can fill up the buffer with true random bits. We can also enlarge the buffer size to fill up the channel. Finally, instead of xoring just two bits together, we can xor as large a subset as we can afford computationally. Thus we can plan to use up the resources of personal workstation in order to enhance security.

Consider the following extreme case with $n > m$, a generalization of the system in Section 4.1:

```
(* Encryption -- general form *)
(* initial load of buffer *)
for i := 1 to n-m do
    C[i] := truerandom;
```

```
(* xor of subsets *)
for i := 1 to m do
    begin
        PR[1 .. (n-m+i -1)] := pseudo-random bit stream;
        C[n-m+i ] := P[i] xor (PR[ 1 .. (n-m+i -1) ]
                        innerproduct C[ 1 .. (n-m+i -1) ]
    end;
(* pseudo-random permutation, as before *)
```

Acknowledgment.

Doren Zeilberger and Jet Wimp gave suggestions about the material in the appendices.

References.

[Bri79] H. S. Bright, and R. L. Enison, "Quasi-random number sequences from a long-period TLP generator with remarks on application to cryptography," *ACM Computing Surveys*, Vol. 11, No. 4, (Dec. 1979), pp. 357-370.

[Den82] D. Denning, *Cryptography and Data Security*, Addison-Wesley, 1982.

[Knu81] D. E. Knuth, *The Art of Computer Programming, Vol. II, Seminumerical Algorithms*, 2nd Edition, Addison-Wesley, 1981.

[Knu73] D. E. Knuth, *The Art of Computer Programming, Vol. I, Fundamental Algorithms*, 2nd Edition, Addison-Wesley, 1973.

[Mas85] J. L. Massey, and I. Ingemarsson, "A simple and provable computationally secure cipher with a finite key," abstract submitted to ISIT-85 and presented at Eurocrypt 85.

[Wag86] N. R. Wagner, P. S. Putter, and M. R. Cain, "Encrypted database design: Specialized approaches," *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, IEEE Computer Society, 1986, pp. 148-153.

[Wag85] N. R. Wagner, P. S. Putter, and M. R. Cain, "Using algorithms as keys in stream ciphers," *Proceedings of Eurocrypt 85, Lecture Notes in Computer Science*, Springer Verlag (to appear).

[Wol85] S. Wolfram, "Random sequence generation by cellular automata," presentation at Crypto 85.

Appendix I. Probability that m k-bit random numbers are distinct.

Choose m k-bit numbers at random, $k \geq \lceil \log_2 m \rceil$. Let $p(m, 2^k)$ denote the probability that all m of the numbers are distinct. (If $k < \lceil \log_2 m \rceil$, then it is impossible for them all to be distinct.) More generally, consider $p(m, N)$, the probability of picking m distinct items out of N, where the selection is done at random with replacement. It is easy to see that

$$p(m, N) = \prod_{i=0}^{m-1} (1 - i/N) = \frac{N!}{(N - m)!N^m}.$$

Values of $p(2^{10}, 2^k)$ and $p(2^{20}, 2^k)$ are tabulated below for various k. Note that for increasing k, $p(m, 2^k)$ first gets greater than 0.5 for $k = 2\lceil \log_2 m \rceil$. As k becomes large, the value $p(2^{k/2}, 2^k)$ tends to 0.6065306597....

The entries in the table were all calculated using the exact formula. However, we can set $m = \alpha N$ and use Stirling's formula to obtain the approximation

$$p(m, N) = p(\alpha N, N) \approx \left[\frac{e^{-\alpha}}{(1 - \alpha)^{(1-\alpha)}} \right]^N (1 - \alpha)^{-1/2}.$$

Though it is only an approximation, this formula is accurate enough to give every digit of every entry in the table, except for the entry $p(2^{10}, 2^{10})$.

As shown Table 1, two lists of $p(m, 2^k)$ values for two different values of m are nearly the same when shifted so that the entries $k = 2\lceil \log_2 m \rceil$ are lined up. This property follows from the approximation

$$p(2m, 4N) \approx p(m, N),$$

which is quite accurate as long as N is not too small. Other interesting approximate recurrences include

$$p(m, 2N) \approx p(m, N)^{1/2}$$
$$p(2m, N) \approx p(m, N)^4$$

and more generally,

$$p(2^a m, 2^b N) \approx p(m, N)^{4^{\wedge}a/2^{\wedge}b}.$$

To prove one variation of these recurrences, use the formula for $p(\alpha N, N)$ with the $(1 - \alpha)^{-1/2}$ part dropped out (accurate for α small). Then $p(2\alpha N, 2N) = p(\alpha(2N), 2N) \approx p(\alpha N, N)^2$, so that

$$p(2m, 2N) \approx p(m, N)^2.$$

$m = 2^{10} = 1024$		$m = 2^{20} = 1048576$	
k	$p(2^{10}, 2^k)$	$p(2^{20}, 2^k)$	k
10	1.5371837 E -443	3.7069749 E -223	30
15	0.00000009652	0.00000011251	35
16	0.0003242093	0.0003354515	36
17	0.018196318	0.018315522	37
18	0.13524703	0.13533519	38
19	0.36799933	0.36787955	39
20	0.60672822	0.60653085	40
21	0.77895928	0.77880093	41
22	0.88259566	0.88249699	42
23	0.93946801	0.93941311	43
24	0.96926219	0.96923326	44
25	0.98451130	0.98449645	45
30	0.99951231	0.99951183	50
35	0.99998475	0.99998474	55

Table 1.

n-m	Probability that random m ×n matrix over GF(2) has rank m (accurate to 12 digits for m ≥ 40)
0	0.288788095086
1	0.577576190173
2	0.770101586897
3	0.880116099311
4	0.938790505932
5	0.969074070639
6	0.984456198745
7	0.992207822357
8	0.996098833425
9	0.998048146211
10	0.999023755347 = 1 - 0.976244 E -3
20	1 - 0.953674 E -6
30	1 - 0.931322 E -9
40	1 - 0.909494 E -12
50	1 - 0.888124 E -15

Table 2.

Appendix II. Probability that an $m \times n$ matrix over GF(2) with random entries has rank m.

Theorem: Consider an $m \times n$ ($n \geq m$) matrix over GF(2) with all entries randomly chosen. For $m = n$, the approximate probability that the matrix is invertible is given by the constant $q_0 = 0.28878809508660242\ldots$ (The approximation is accurate to about 12 digits for $m \geq 40$.) More generally, for $n \geq m$, the approximate probability that the matrix has rank m is given by Table 2. (The probability of rank m is very roughly $1 - 10^{-0.3(n-m)}$.)

Proof: Consider first the $m \times m$ case. There are $2^{m\wedge 2}$ matrices altogether over GF(2). Let $F(m)$ denote the number of invertible matrices. To calculate $F(m)$, we just count the number of ways to build up m linearly independent rows. This gives $F(m) = (2^m - 2^0)(2^m - 2^1)(2^m - 2^2)\ldots (2^m - 2^{m-1})$. $F(m)$ satisfies the recurrence $F(m) = (2^{2m-1} - 2^{m-1})F(m - 1)$. Since $G(m) = 2^{m\wedge 2}$ satisfies $G(m) = 2^{2m-1}G(m - 1)$, we suspect that $F(m)$ looks like $2^{m\wedge 2}$.

$$\frac{F(m)}{2^{m\wedge 2}} = 1/2 * 3/4 * 7/8 * 15/16 * \ldots * (2^m - 1)/2^m.$$

The corresponding infinite product is given by a special Theta function known as a Q-function.

$$Q_0(q) = \prod_{m=1}^{\infty} (1 - q^{2m}), \text{ where we want } Q_0(1/\sqrt{2}).$$

This product converges rapidly to $q_0 = 0.28878809508660242\ldots$. Thus F(m) is approximately $q_0 * 2^{m\wedge 2}$. (The approximation is accurate to 3 digits when $m = 10$, 6 digits when $m = 20$ and generally about $0.3 * m$ digits.) The other figures in the table of the theorem are obtained by dropping the initial $n - m$ terms of the infinite product.