Security Weaknesses in Maurer-Like Randomized Stream Ciphers

Niels Ferguson*, Bruce Schneier**, and David Wagner***

Abstract. TriStrata appears to have implemented a variation of Maurer's randomised cipher. We define a variation of Maurer's cipher that appears to be similar to the TriStrata version, and show several cryptanalytical attacks against our variant.

1 Introduction

In 1990, Maurer introduced an information-theoretic provably-secure randomised cipher [Mau90]. Using a large pool of public random data, he shows how to use a key made up of multiple pointers into the pool to create a series of random streams to XOR into the plaintext. Someone with the same key can XOR the same random streams into the ciphertext to recover the plaintext; an analyst without the key must brute-force search the set of possible pointers to recover the plaintext.

Recently, TriStrata Corporation [Tri98,Bec98] implemented a complexity-theoretic variation of that technique as a proprietary encryption algorithm. In this paper we first present what we infer to be the exact variation used by TriStrata, and take some initial steps in cryptanalysing the cipher.

We should point out that our analysis does *not* apply to the original cipher proposed by Maurer, where the pool is chosen to be large enough that it is infeasible to read the entire contents of the pool. Unfortunately, Maurer's cipher is not very practical—in his paper, he proposed digitizing the surface of the moon as one means of getting enough public randomness to make the cipher work—and hence not suited to present-technology implementation. We attack a simplified version, one which is more practical.

2 The TriStrata Cipher

The TriStrata Cipher is a key stream cipher with a two-part key. The first part of the key is a 1 Mbyte (2^{20} bytes) block of random data which we call the pool. For practical reasons the pool is not changed very often. In the TriStrata system,

* Counterpane Systems; 101 E Minnehaha Parkway, Minneapolis, MN 55419, USA; niels@counterpane.com.
** Counterpane Systems; schneier@counterpane.com.
*** University of California Berkeley, Soda Hall, Berkeley, CA 94720, USA; daw@cs.berkeley.edu.

it appears to be fixed for a long time. Additionally, the same pool is used by all clients in the system.

The second part of the key consists of a number of pointers that point to a byte in the pool. Although our information is sketchy, we believe there are 5 pointers. Each pointer is 20 bits long, so the pointers together make up a 100-bit key.[1]

To generate a byte of the key stream, the five bytes that the pointers point to are XORed together. Each of the pointers is then advanced by one byte position. If all pointers were to wrap around at the end of the pool, the resulting key stream would have a period of only 2^{20} bytes. We assume that the first pointer wraps at the end of the pool, the second pointer wraps around one byte from the end, etc. We can thus represent the keystream generation algorithm as follows:

$$k_i = \bigoplus_{j=1}^{5} \mathrm{pool}[(t_j + i) \bmod (2^{20} - i + 1)]$$

where t_j is the starting position of the j'th pointer and k_i is the i'th byte of the key stream. The i'th byte of the ciphertext is formed by XORing the i'th byte of the plaintext and k_i. This is one possible generalisation of the Morehouse variation of the Vernam cipher [Kah68].

There are of course many other possible variations. The number of pointers can be changed, as can the update rule for the pointers (increment one is the simplest update rule). In Maurer's cipher, each pointer cycles through its own unique subset of the pool. These variations do not affect the spirit of our analysis, and many of our attacks will work against such variations.

In Sections 3–5 we will discuss various attacks on this cipher. Most of our attacks are known-plaintext attacks, which corresponds to a known-keystream attack. Some of the attacks are of theoretical interest only (as certificational attacks), but several are efficient enough to be of practical concern. See Table 1 for a summary of cryptanalytic results.

3 Finding the Pointers from a Known Pool

In many situations the pool does not change very often, and is not really a secret. For example, in the TriStrata system the same pool is used for all clients in the system, so it is plausible to assume that the pool is known to the attacker.[2] Alternatively, we might be able to recover the pool using the techniques to be described later in Section 4.

3.1 Exhaustive Search

Given 16 bytes of the key stream, we can search for a set of t_j values that would produce this key stream. It is very unlikely that one of the pointers performed a

[1] In comparison: Maurer gave an example that used 50 pointers and a pool of 2^{72} bits.
[2] Note that in Maurer's original system, this pool is assumed to be public information.

Attack type	Time	Space	Data	Attack model
Exhaustive search	2^{93}	—	16 bytes	Known-pool
Meet-in-the-middle	2^{57}	2^{39}	16 bytes	Known-pool
Improved MITM	2^{57}	2^{23}	16 bytes	Known-pool
Weak keys	2^{39}	2^{39}	2^{17} keys	Known-pool
Linear algebra	$\leq 2^{60}$	$\leq 2^{40}$	2^{20} bytes	Known-pointers
Exhaustive search	$\leq 2^{160}$	$\leq 2^{40}$	2^{20} bytes	Nothing known
Vigenere analysis	$\leq 2^{60}$	$\leq 2^{40}$	$2^{22.3}$ bytes	Nothing known
Diff. related-key	$\leq 2^{60}$	$\leq 2^{40}$	2^{21} bytes	Related-key; nothing known
Diff. related-key	$5 \cdot 2^{20}$	—	80 bytes	5 related keys; known-pool
Diff. fault	2^{39}	2^{19}	16 bytes	Fault attack; known-pool

Table 1. A summary of our attacks on the Maurer-like stream cipher.

wrap-around within these 16 bytes, so we assume that this did not happen. We are thus looking for 5 sub-ranges in the pool of 16 bytes each that when XORed together results in the key stream.

If all t_j values are distinct, then we have to try $\binom{2^{20}}{5} \approx 2^{93}$ different sets of values. If two t_j values are the same their contributions to the key stream cancel out, and we look for a set of 3 pointers. The extreme case is when there are two pairs of t_j values that are identical, in which case the key stream is just a sub-range of the pool. These special cases do not contribute significantly to the complexity of the attack.

If one of the pointers did wrap around, the attack fails. We can repeat the attack on the next 16 bytes of the key stream, which will most likely succeed.

We conclude that an exhaustive search over the pointer values for a known pool has a complexity of 2^{93} steps. Simple optimisations can make each of these steps extremely efficient.

3.2 Meet-in-the-Middle on the Pointers

We can improve this if we use a meet-in-the-middle attack on the pointer space. We generate all pairs (t_1, t_2) and consider the key stream contributions made by these pointers. As (t_1, t_2) produce the same result as (t_2, t_1) we can restrict ourselves to the 2^{39} cases where $t_1 < t_2$. We store these pairs in a list, and sort them lexicographically by the key stream that they generate. We have 2^{39} pairs of 5 bytes each, which requires 2.5 Terabytes of memory. This list is computed once for a given pool and then stored. The complexity of this phase is 2^{39} steps.

We then try all possible values for t_3, t_4, and t_5 in a second phase. There are $\binom{2^{20}}{3} \approx 2^{57}$ different values (again taking advantage of the fact that swapping two t values gives the same key stream). From these three t values and the known key stream, we compute the key stream contribution of the first two pointers. We now look in the sorted list to see if there is a pair of values for (t_1, t_2) that generates the required key stream. Each such search in the sorted list may be performed quickly using binary search or hashing.

The complexity of the second phase of the attack is about 2^{57} steps, where each step requires a lookup in a sorted list of 2^{39} elements.

We can make this attack practical by a simple divide-and-conquer technique. We split the work into 2^{16} tasks, each identified by a two-byte string σ. For each task, we first generate all pairs (t_1, t_2) as above whose contribution to the key stream start with σ. There are about 2^{23} elements in this list. If we pre-compute a table that given a two-byte string points to all places in the pool where this two-byte string occurs, then we can construct this list in about 2^{23} steps. The entire list requires about 40 Mbytes of memory, well within the range of even a standard desktop PC.

The second phase is to try all possible values for t_3, t_4, and t_5 with the restriction that their contribution XORed with the key stream starts with σ. We try all possible values for t_3 and t_4, compute the desired first two bytes of the key stream contribution of t_5 and use the same pre-computed table to find these values. We then search for a suitable pair (t_1, t_2) in our list for which the rest of the key stream also matches.

For any single task σ, the first phase has a complexity of 2^{23} steps, and the second phase has a complexity of about 2^{41} steps if we use some simple ordering requirements to avoid equivalent pointer values. As we have to perform a total of 2^{16} tasks, the overall complexity of our attack is still $2^{16} \times (2^{23} + 2^{41}) \approx 2^{57}$ steps. However, this improved version has a very reasonable memory requirement, and can be spread out over many machines. This attack should certainly be considered feasible for any reasonably sized organisation.

3.3 Weak Keys

If the pointer values are generated randomly, then about one in every 2^{17} keys has at least two pointers with the same starting value. The contributions of these pointers cancels, and we are left with at most three relevant pointers. Using the techniques described here, these pointers can be found with very little work. Thus about one in every 2^{17} keys is a weak key.

With sufficient cryptanalytic targets, we could expect to break our first key after about 2^{39} steps. Build a table of 2^{39} elements by enumerating all possibilities for the first two pointers. Next, for each cryptanalytic target, check whether it forms a weak key by guessing the third pointer and doing a table lookup. This takes 2^{20} work to see if a key is weak; after 2^{17} such attempts, we expect to find a single weak key which we can break. Thus, the total complexity is 2^{39} work and 2^{39} space, assuming at least 2^{17} different keystream segments are available for analysis.

4 Finding the Pool

The attacks in Section 3, above, assumed that the pool was known to the adversary. However, as we now show, even when the pool is unknown, it may be possible to recover the pool using more sophisticated techniques, and then the techniques of Section 3 will apply.

4.1 The Known-Pointers Case: Linear Algebra

Let us assume that we know the t_j values for a given encryption. Then each known byte of the key stream gives us a linear equation in 5 values of the pool. After one million key stream bytes we expect to have a non-singular set of equations and can solve for the bytes in the pool.

If we do not use the correct values for t_j, then we can expect the set of equations to be contradictory very soon after the first million key stream bytes. This allows us to detect whether a set of values for t_j is correct.

If we are just interested in detecting whether a set of t_j values is correct, we can perform this analysis for some subset of the bits in a byte. For example, it could be done for the least significant bit of each pool byte. This optimisation requires less memory and might be faster. We have not investigated the exact performance tradeoffs of this attack in detail. It might be possible to take advantage of the highly structured form of the equations instead of using an algorithm for general linear equations.

4.2 When Pointers are Unknown: Exhaustive Search

We can use the result of Section 4.1 to mount an exhaustive search attack. For each possible set of t_j values, perform the attack of Section 4.1. If the set of equations is contradictory, the values for t_j are incorrect. If a consistent set of pool bytes is found, the full key has been recovered. This attack requires about 2^{100} steps, where each step consists of setting up just over a million equations and checking for a contradiction.

4.3 Unknown Pointers: A Vigenere Analysis

When the t_j values are unknown, we can still recover the entire pool from about $5 \cdot 2^{20}$ bytes of known keystream by treating the cipher as a multiple-loop Vigenere cipher. We let

$$k_{i,j} = \mathrm{pool}[(t_j + i) \bmod (2^{20} - i + 1)]$$

so that $k_i = k_{i,1} \oplus \ldots \oplus k_{i,5}$. Each $k_{\cdot,j}$ takes the form of a Vigenere cipher, and their XOR is a five-loop Vigenere cipher. Here we shall ignore the fact that the five streams are related, and simply treat them as independent values.

Such ciphers can be readily cryptanalyzed by standard methods [Sin68,Tuc70], such as the application of linear algebra. In this way, with about $5 \cdot 2^{20}$ bytes of known keystream, we can completely recover the five streams $k_{i,1}, \ldots, k_{i,5}$. Once the $k_{i,j}$ are known, the initial pointer values t_j and the contents of the pool can be readily obtained by inspection of the $k_{i,j}$.

The discussion in Section 2 suggested using a different modulus for each pointer, because otherwise the keystream k_i would have a relatively short period of only 2^{20} bytes. This attack shows that using a different modulus for each pointer does not extend the security of the cipher very much: not more than by a factor of five, in any case. Also, we note that this attack does not work against the variant where each stream has the same period, because we will not be able to obtain the required quantity of known keystream material. These observations suggest that it might not make much difference whether we use the same modulus for the pointers or not.

5 Differential Fault and Related-Key Attacks

We next show that, when a few bits from the pool can be corrupted, improved attacks are available. We first discuss how to mount such attacks when the adversary can make related-key queries, and then we illustrate how random bit errors can enable a differential fault attack on the cryptosystem.

5.1 Differential Related-Key Attack on the Pool

Let us assume that the attacker can force a low-weight change in the pool. This can either be done by manipulating the distribution protocol, tampering with the stored pool, or waiting for a random bit error to occur.

In this case, the attacker can determine the distances between the different pointers by observing where the erroneous key stream deviates from the proper key stream. If two Megabytes of key stream is available, this reveals the distance between the pointers. If the attacker knows where the error in the pool occurred, then the pointer values are revealed and the pool can be reconstructed using the attacks in Section 4. If the attacker does not know this, then he has to guess the position and has to perform some attack from Section 4 at most 2^{20} times.

5.2 Differential Related-Key Attack on the Pointers

If the pool is known, a differential attack on a pointer will reveal the position of the pointer. Let us assume the attacker flips the least significant bit of pointer 1. The difference between the modified key stream and the original key stream uniquely determines the value of pointer 1. This attack can be repeated for each of the pointers, resulting in an attack that requires 5 related key queries and a complexity of $5 \cdot 2^{20}$.

The attack can be generalized to use fewer related key queries; see the next subsection for an example.

5.3 Random Error Attack

We can extend this to the case where the attacker suspects there is a random bit error in the pointers. As before, the attacker can determine one of the pointers from the difference in the key stream. The remaining 4 pointers can be found with a meet-in-the-middle attack like the one in Section 3.2 with a complexity of 2^{39}. This is a kind of differential fault analysis, where a low-weight difference in the key input yields information that helps the attacker significantly.

6 Security Improvements

The obvious way to improve the security is to increase the number of pointers. This will make many of the attacks harder to implement in practice, but also results in a slower cipher. A rough estimate shows that a version with the same size of pool but 14 pointers can be attacked in 2^{128} steps using the meet-in-the-middle attack of Section 3.2. The resulting cipher would be slower by a factor of 3 or so, and still be vulnerable to the attack in Section 4.3.

7 Uses of Maurer-Type Ciphers

Simplified variants of Maurer's cipher might be useful for very specific applications. For example, let us suppose we need to encrypt a very long stream of data very rapidly. We generate a pool and a set of starting pointers using a cryptographically strong pseudo-random generator, and use the type of cipher we described here to encrypt the bulk data. Using 4, 8, or even 16-byte words instead of bytes would increase the efficiency of the algorithm even further. Such a construction would have to be carefully crafted and analysed. This deserves a lot of further study, and until that has been done this type of cipher should not be used.

The cipher we analyzed is similar (in some ways) to other ciphers published elsewhere. For instance, Chameleon [AM97] uses four pointers into a pool of 2^{16} 64-bit words; however, the primary difference is that Chameleon drives the pointers with another (slower) stream cipher, rather than generating them by incrementing. In fact, in Chameleon the random bit error property of Section 5.1 is a feature, not a bug: it is exploited to detect traitors. Another paper [AVV95] describes a "provably secure" stream cipher which uses a Maurer-like cipher as an internal component; there, they use eight pointers into a pool of 2^8 512-bit words, and again, the pointers are updated as in Chameleon rather than by incrementing. We leave it as an open question to extend our analysis to these cases.

8 Conclusion

Simplified Maurer-like ciphers are not always secure. We have shown that many variations can be broken far more effectively than an exhaustive search of the key space. Several of our attacks are quite practical.

9 Acknowledgements

We are grateful to David Bernier and to the anonymous reviewers for their helpful comments.

References

[AM97] Ross Anderson and Charalampos Manifavas, "Chameleon—A New Kind of Stream Cipher," *FSE'97*, Springer, 1997.

[AVV95] W. Aiello, S. Venkatesan, and R. Venkatesan, "Design of practical and provably good random number generators," *SODA'95*, ACM, 1995, pp. 1–9.

[Bec98] Dan Beckman, "TriStrata: A Giant Step In Enterprise Security," *Network Computing*, 15 September 1998.

[Kah68] David Kahn, *The Codebreakers*, 1968.

[Mau90] Ueli M. Maurer, "A Provably-Secure Strongly-Randomized Cipher," *Advances in Cryptology - Eurocrypt '90*, Springer-Verlag, 1990, pp 361–373.

[Sin68] A. Sinkov, *Elementary Cryptanalysis, A Mathematical Approach.* New York: Random House, 1968.

[Tri98] TriStrata webpage, `http://www.tristrata.com`.

[Tuc70] B. Tuckerman, "A study of the Vigenere-Vernam single and multiple loop enciphering systems," IBM Research Report RC2879, 14 May 1970, Yorktown Heights NY.