New Cipher Structures

Provably Secure and Efficient Block Ciphers

Pat Morin
School of Computer Science
Carleton University
morin@scs.carleton.ca

Abstract.

The security and efficiency of two recently proposed block ciphers, BEAR and LION, both based on a hash function and a stream cipher, is discussed. Meet-in-the-middle attacks are presented which can be used to dramatically reduce the complexity of a brute-force key search on both these ciphers. A new block cipher is described which is not susceptible to meet-in-the-middle attacks, is provably secure against any chosen plaintext or ciphertext attack, and is more efficient than BEAR or LION.

1 Introduction

A number of examples exist which show how cryptographic primitives can be composed to yield other cryptographic primitives. Two examples of this are the well known output-feedback mode of DES which converts a block cipher into a stream cipher, and feedforward mode which converts a block cipher into a hash function. Often, these compositions are provably secure in that an efficient attack on the composite function would lead to an efficient attack on the underlying primitive.

These types of construction are of practical interest, as there often exist efficient cryptographic primitives which, although not provably secure, are widely believed to be secure because of empirical evidence. Composing such primitives to yield higher level functions can lead to new cryptographic functions that are practical to implement and have the advantage that their security is based on the security of trusted primitives.

In this paper we examine the security of two recently proposed constructions, BEAR and LION, which allow block ciphers to be created from a stream cipher and a hash function. The resulting block ciphers are provably secure in that a key-recovery attack that can be mounted with a single plaintext ciphertext pair can be used to break both the hash function and the stream cipher. We show that although these ciphers are provably secure, they are susceptible to meet-in-the-middle attacks which greatly reduce the complexity of a brute force key search.

We also propose a new construction. This is a block cipher which is more efficient in terms of computation than either of the above schemes, is not susceptible to meet-in-the-middle attacks, and is provably secure against any combination of chosen plaintext

or ciphertext attacks. This cipher is also of practical interest, as it can be implemented very efficiently in software.

The remainder of the paper is divided as follows: Section 2 gives a brief review of BEAR and LION. Section 3 discusses the security of BEAR and LION and presents attacks on them. Section 4 presents AARDVARK a new block cipher that is similar in construction to BEAR and LION. Section 5 discusses the performance of actual implementations of BEAR, LION, and AARDVARK. Section 6 summarizes this work and presents an open problem in this area.

2 BEAR and LION

In this section we present BEAR and LION, two very recent block ciphers due to Anderson and Biham [AnBi96]. Both these ciphers are constructed from a stream cipher, S, and a hash function, H, using a construction similar to those of Luby and Rackoff [LuRa88]. The requirements on S and H are:

1. H is one-way, given only $H(X)$ it is hard to find X;
2. H is strongly collision free, it is hard to find distinct X and Y such that $H(X) = H(Y)$;
3. S resists key recovery attacks, it is hard to find the seed X given $Y = S(X)$;
4. S resists expansion attacks, it is hard to expand any partial stream of $Y = S(M)$, i.e., given some subset of Y it is hard to determine anything more about Y.

The block size, m, of these ciphers is variable, but is on the order of 1Kbyte-1Mbyte. If we define k as the block size of the hash function used then both these ciphers are unbalanced Feistel networks in which $|L| = k$ and $|R| = m - k$.

BEAR performs encryption and decryption using two applications of a hash function and one application of a stream cipher. A BEAR key consists of two sub-keys, K_1 and K_2, both of size $|K| > k$. BEAR encryption and decryption is done by:

Encryption	Decryption
$L = L \oplus H_{K_1}(R)$	$L = L \oplus H_{K_2}(R)$
$R = R \oplus S(L)$	$R = R \oplus S(L)$
$L = L \oplus H_{K_2}(R)$	$L = L \oplus H_{K_1}(R)$

LION is similar in construction to BEAR except that encryption and decryption involve one application of the hash function and two applications of the stream cipher. Again, the key consists of two sub-keys, K_1 and K_2, both of size k. LION encryption and decryption are done by:

Encryption	Decryption
$R = R \oplus S(L \oplus K_1)$	$R = R \oplus S(L \oplus K_2)$
$L = L \oplus H(R)$	$L = L \oplus H(R)$
$R = R \oplus S(L \oplus K_2)$	$R = R \oplus S(L \oplus K_1)$

BEAR and LION are potentially very efficient ciphers given the speeds at which modern stream ciphers and hash functions operate. Anderson and Biham report encryption rates of 13Mbits/sec for BEAR and 16Mbits/sec for LION[1].

3 Security of BEAR and LION

BEAR and LION are provably secure in the sense that an oracle that can recover the key of BEAR or LION given one plaintext/ciphertext pair can be used to "break" the underlying keyed hash function and stream cipher. In the case of the hash function this means that the oracle can be used to undermine the one-way and collision free properties. In the case of the stream cipher, the oracle can be used to undermine the "resists key recovery" property.

While provable security is a desirable property, it does not tell the whole story. In the case where an adversary is able to break either the hash function or the stream cipher, the adversary can obtain partial information about the plaintext given just the cipher text. These properties are discussed by the authors. The authors also discuss attacks such as differential and linear cryptanalysis which require many plaintext/ciphertext pairs. They argue that a successful attack on BEAR or LION using these techniques would yield a successful attack on either the hash function, the stream cipher or both. The authors also suggest that these types of attacks can be avoided by using a different key to encrypt each block of data. This is a reasonable approach given BEAR's large block size

An interesting property of BEAR is that given only half the key bits, namely the bits of K_2, an attacker can determine *most of* the plaintext. Given a ciphertext L' and R', and the sub-key K_2 an attacker can determine the plaintext R by doing a partial decryption:

$$L^* = L' \oplus H_{K_2}(R')$$
$$R = R' \oplus S(L^*)$$

Since $|R|$ is typically much larger than $|L|$ this means that an attacker can determine most of the plaintext without any further cryptanalysis. This is clearly not a desirable property.

A brute force key search on BEAR would be of complexity $2^{2|K|}$, but this can be reduced considerably through the use of a meet-in-the-middle attack [MeHe81]. Given a plaintext/ciphertext pair, $P = (L, R)$, $C = (L', R')$, the attacker computes and stores $H_{K_1}(R) \oplus L$ for all $2^{|K|}$ possible values of K_1. Using these stored values the attacker then start computing $H_{K_2}(R') \oplus L'$ for all $2^{|K|}$ possible values of K_2 until she finds K_1 and K_2 such that $H_{K_1}(R) \oplus L = H_{K_2}(R') \oplus L'$. She can then test if this is the correct key-pair by verifying whether $S(H_{K_1}(R) \oplus L) = R \oplus R'$ or not. The correct key-pair will be found using at most $2^{|K|+1}$ encryption operations. Of course, such an attack is largely theoretical, as the value of $|K|$ is likely to be 128 or more.

The same type of partial decryption attack that was demonstrated against BEAR can be used against LION to recover L if K_2 is known. However, this attack is less effective

[1] A software implementation on a 133MHz DEC Alpha using SHA as the hash function and SEAL as the stream cipher

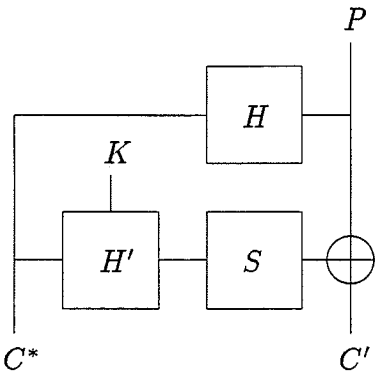

Figure 1: The AARDVARK cipher.

against LION as it only allows the attacker to recover L, which would typically not be more than 256 bits in length.

A similar meet-in-the-middle attack can be mounted against LION as was demonstrated against BEAR. In this case the attack is mounted on the stream cipher S and the attacker attempts to find K_1 and K_2 such that $S(K_1 \oplus L) \oplus R = S(K_2 \oplus L') \oplus R'$ and verifies candidate key pairs by checking whether $H_{K_1}(S(K_1 \oplus L) \oplus R) = L \oplus L'$.

The existence of meet-in-the-middle attacks against these ciphers leads one to believe that they are not as secure as a cipher with a 256+ bit key size could be. In the following section we discuss a new block cipher which is similar in construction to BEAR and LION, uses only a single key of length approximately k, and is not susceptible to meet-in-the-middle attacks.

4 AARDVARK

This section presents a new block cipher, which we call AARDVARK[2]. AARDVARK is based on a stream cipher, S, a hash function, H, and a keyed hash function, H'. Like BEAR and LION, the block size of AARDVARK is variable with values of 1Kbyte-1Mbyte recommended. Unlike BEAR and LION, AARDVARK is not a Feistel network and has the property that the ciphertext is slightly larger than the corresponding plaintext. As we will see, this property is not undesirable as it allows users to verify the integrity of a message during decryption.

An AARDVARK network is shown pictorially in figure 1. An AARDVARK key, K, is a bit string suitably long for keying H'. It is recommended that $|K|$ be greater than or equal to the number of bits output by H'. Encryption in AARDVARK consists of one application each of the hash function and the stream cipher and an application of the keyed hash function on a short (< 512 bit) string. This produces two values, C^* and C', both of which make up the ciphertext $C = (C^*, C')$. Encryption is done by:

$$C^* = H(P)$$
$$C' = P \oplus S(H'_K(C^*))$$

[2]All the exotic animal names were already being used.

Decryption is done by:

$$P = C' \oplus S(H'_K(C^*))$$

To verify that a ciphertext was not modified, either maliciously or by accident, the receiver can also verify that $H(P) = C^*$. This is particularly important since an adversary could easily modify the ciphertext in such a way that the plaintext is changed in a predictable manner. This is because flipping a bit in C' has the effect of flipping the corresponding bit in the decrypted plaintext.

To ensure the security of AARDVARK, S and H must have the following properties:

1. H is strongly collision free, it is hard to find distinct X and Y such that $H(X) = H(Y)$;
2. H' resists existential forgery, given an oracle that computes H'_K for an unknown K it is hard to compute $H'_K(X)$ for any X without using the oracle directly;
3. S resists expansion attacks, it is hard to expand any partial stream of $Y = S(M)$;
4. S and H' are independent, there is nothing about S and H' which allows someone to compute $S(H'_K(X))$ easily without knowing K.

The strength of AARDVARK lies in the difficulty of computing $H'_K(X)$ without knowing K. Since S and H are independent, this makes computing $S(H'_K(X))$ difficult, which in turn makes computing the encryption and decryption functions difficult. We now prove the main theorem about the security of AARDVARK.

Theorem 1. *Given two oracles, one which computes E_K and one which computes D_K for an unknown K, it is hard to find (P, C) such that $E_K(P) = C$ without using one of the oracles to compute $E_K(P)$ or $D_K(C)$ directly.*

Proof. Suppose (bwoc) that we have an efficient attack which allows us to find such a (P, C) without using the oracles directly. Using this attack we find a valid plaintext/ciphertext pair, (P, C), without using the oracles to compute $C = E_K(P)$ or $P = D_K(C)$ directly. During the attack one of two situations occurred:

1. We found two distinct plaintexts P_1 and P_2 such that $H(P_1) = H(P_2)$.
2. We did not find two distinct plaintexts P_1 and P_2 such that $H(P_1) = H(P_2)$.

In the first case, we found a collision in H which contradicts the strongly collision free assumption on H. In the second case, we were able to find (P, C^*, C') such that $C' = P \oplus S(H'_K(C^*))$ and $C^* = H(P)$. In particular, since we did not find a collision in H, we are able to compute $P \oplus C' = S(H'_K(C^*))$, where without using the oracle directly. This contradicts either our assumption that H' resists existential forgery (since we were never given the value of $H_K(C^*)$) or that S and H' are independent.

□

Block Size	4096	65536	1024000
AARDVARK	1024000	2952072	3277849
BEAR	853333	1790601	1909735
LION	602353	2184533	2568991

Table 1: Encryption rates of AARDVARK, BEAR and LION for various block sizes. Block sizes are given in bytes. Encryption rates are given in bytes/second.

This theorem says that AARDVARK is secure against any combination of chosen plaintext/ciphertext attack provided that the above mentioned requirements on H and S are met. Security is defined in a very general sense: it is not computationally feasible for an attacker to find a single valid plaintext/ciphertext pair. This is a powerful result, since hash functions and stream ciphers exist which (for practical purposes) satisfy these requirements.

The attentive reader may have noticed that we have not made use of the "resists expansion attacks" requirement placed on S. It is, however, an important requirement. Since the block size of AARDVARK is large, it is possible that an adversary may know some of the plaintext of a message which means that the adversary knows a partial output of S. The "resists key expansion" requirement on S prevents such an adversary from learning any more about the plaintext.

AARDVARK is clearly resistant to meet-in-the-middle attacks since the key is not divided in any way, and has a built in method of ensuring ciphertext integrity. Since AARDVARK uses only one application each of the hash function and stream cipher plus an application of the keyed hash function on a short bit string, one would also hope that it would run faster than BEAR or LION.

5 Performance of AARDVARK, BEAR, and LION

In order to test the relative performance of AARDVARK, BEAR, and LION we implemented them. SHA [NBS93] was used as the hash function and SEAL [RoCo93] as the stream cipher. In BEAR and AARDVARK, SHA was keyed using $H'_K(M) = \text{SHA}(K\|M\|K)$. The stream cipher was given by $S(M) = \text{SEAL}_M(0)\|\text{SEAL}_M(1)\|\cdots$. The test machine was a Sun Ultra-Sparc with a 140MHz Ultra-1 processor. All source code was written in C and compiled using *gcc* with optimization enabled.

All three ciphers were tested with various block sizes.The results are shown in table 1. As we would expect, AARDVARK outperforms BEAR and LION for all block sizes, with encryption rates varying from 1Mbyte/sec with a 4KByte block size to 3.2MBytes/sec with a 1MByte block size.

A perhaps surprising result is that BEAR outperforms LION for small block sizes, but the opposite is true for larger block sizes. This is because SEAL runs faster than SHA, but has a high key setup time. Because of this, LION which uses two applications of SEAL is slower for small block sizes, but as the block size is increased, the key setup times become less important and LION overtakes BEAR. This suggests that a stream cipher with a low

key setup time is called for.

6 Conclusions

Meet-in-the-middle attacks on both BEAR and LION have been presented which, although largely theoretical, suggest that these ciphers are not as strong as they could be. A new block cipher has been proposed which is more efficient than BEAR or LION, is provably secure against any combination of chosen plaintext and/or ciphertext attacks, and has a built-in method of verifying ciphertext integrity. This work is of practical interest as SHA/SEAL based AARDVARK makes for a very fast software block cipher.

As noted in section 5 and in [AnBi96], the key setup times for the SEAL stream cipher have a significant negative impact on the performance of block ciphers based on it. BEAR, LION, and AARDVARK could be made substantially faster by the development of a stream cipher with lower key setup times. This is the subject of ongoing research.

As with all new cipher proposals, we encourage cryptanalysis of AARDVARK. Such analysis could take the form of attacks against general AARDVARK or against specific instances of AARDVARK (e.g. SHA/SEAL based AARDVARK).

Acknowledgments

The author would like to thank Mike Just and Carlisle Adams for their comments and suggestions on earlier drafts of this paper.

References

[AnBi96] R. Anderson, E. Biham, "Two Practical and Provably Secure Block Ciphers: BEAR and LION," in *Proceedings of the Third International Workshop on Fast Software Encryption*, Cambridge, UK, pp.113-120, 1996.

[BeRo93] M. Bellare, P. Rogaway, "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols," in *First ACM Conference on Computer and Communications Security*, ACM, November, 1993.

[LuRa88] M. Luby, C. Rackoff, "How to construct pseudorandom permutations from pseudorandom functions," in *SIAM Journal on Computing*, v. 17, no. 2, pp.373-386, 1988.

[MeHe81] R. Merkle, M. Hellman, "On the Security of Multiple Encryption," in *Communications of the ACM*, v.24, n.7, pp.465-467, 1981.

[NBS93] "Secure Hash Standard," National Bureau of Standards FIPS Publication 180, 1993.

[RoCo93] P. Rogaway, D. Coppersmith, "A Software-Optimized Encryption Algorithm," in *Proceedings of the 1993 Cambridge Security Workshop*, Cambridge, UK, pp.56-63, 1993. es

Message Encryption and Authentication Using One-Way Hash Functions

Chae Hoon Lim

Center for Advanced Crypto-Technology, Baekdoo InfoCrypt, Inc.
96-14, Chungdam-Dong, Kangnam-Gu, Seoul, 135-100, KOREA
E-mail : chlim@baekdoo.co.kr

Abstract

A one-way hash function is an important cryptographic primitive for digital signatures and authentication. Recently much work has been done toward construction of other cryptographic algorithms (e.g., MACs) using hash functions. In particular, such algorithms would be easy to implement with existing codes of hash functions if they are used as a black box without modification. In this paper we present new such constructions for block ciphers and MACs in some general form (i.e., with variable key sizes, block lengths and MAC lengths).

1 Introduction

Hash functions play an important role in various cryptographic protocol designs. They are used as a cryptographic primitive for digital signatures and message/user authentication. Consequently a lot of optimized implementations of hash functions, such as MD5 [23] and SHA [24], exist. In this paper we describe several algorithms constructed from keyed hash functions: DES-like block ciphers, stream cipher-like algorithms and MAC algorithms. All our constructions are parameterized, so that suitable parameters can be chosen depending on applications. For security of these algorithms we require the underlying hash function to be pseudorandom. This requirement seems not too restrictive since widely accepted hash functions are believed to behave pseudorandomly.

Furthermore, all the algorithms presented in this paper were designed only using the underlying hash function as a black box. This has two important advantages. First, the hash function can be easily replaced if some serious weaknesses are found in the hash function or if newly designed hash functions are preferred (from the viewpoint of security or efficiency). Easy implementation is another practical advantage. Such algorithms can be easily implemented using widely available hash codes. These advantages will be the main motivation for the design of hash-based cryptographic algorithms. If a slight modification (e.g., in initializing variables or padding rules) is allowed, we may obtain a little better efficiency in some algorithms.

2 DES-like Ciphers from Hash Functions

2.1 Background

The famous paper by Luby and Rackoff [19] showed that a pseudorandom permutation (a super pseudorandom permutation, resp.) can be constructed from three (four, resp.) pseudorandom functions by using them as the round functions in three(four, resp.)-round DES. The resulting three-round block cipher is then provably secure against chosen plaintext attacks, and the four-round cipher is secure against chosen plaintext/ciphertext attacks. Though we can construct pseudorandom functions from any one-way function, they are not practical for use in block cipher design. Instead, hash functions may be used to approximate pseudorandom functions (e.g., see [7]).

Our main objective in designing hash-based block ciphers is to use well established and widely deployed one-way hash functions (e.g., MD5 [23], SHA [24] and RIPEMD [9, 15]) as a cryptographic primitive for block ciphers. We here note that recently MD4 and MD5 have been shown not collision-free [12, 13, 14]. However, the fact that MD5 is not collision-free does not mean that it is easy to find collisions for secretly keyed MD5. Anyway, for security of block ciphers based on hash functions we need a more stronger assumption on the underlying hash function, i.e., its pseudorandomness.

2.2 Our Construction

There have been published several block ciphers based on one-way hash functions. Kaliski and Robshaw [16] designed a fast block cipher with a large block size by applying MD5 design technique. Anderson and Biham [1] proposed block ciphers based on the unbalanced Feistel structure using hash functions and stream ciphers. A similar implementation only using hash functions was proposed by Lucks [18]. As mentioned before, since we want direct call of hash functions as a subroutine, we do not try to improve performance either by modifying hash functions (as in [16]) or even by using compression functions (as in [18]). Thus our construction is straightforward from Luby-Rackoff's results.

Since most well-known hash functions process messages by 512-bit blocks, this block size will be assumed throughout this paper. This means that if we only need compression of one message block, the input length should be restricted to less than 448 bits (Note the padding length of 1 to 512 bits and 64-bit encoding of message length in MD4 family hash functions, except for HAVAL [26]). The user-supplied secret key can be of any length within a certain maximum due to the restriction of at most 447-bit one-block message. The use of variable-size secret keys, which is particularly easy to implement in hash-based block ciphers, may be useful for export control and software key escrowing (e.g., see [4, 5]). The secret key for normal use is often recommanded to be 128 bits long.

- Notations:
  - r: number of rounds in DES-like cipher ($r \geq 3$).
  - b: bit-length of plaintexts/ciphertexts (block size).
  - l: bit-length of hash output (typically 128 or 160).

- N: bit-length of hash input (typically 512).
- H : secure hash function.

- Key Scheduling:
  The subkeys $\{K_i\}_{i=1}^{r}$ required by r rounds are derived from the user-supplied secret key K as
  $$K_i = H(u_i, K) \text{ for } i = 1, 2, \cdots, r,$$
  where u_i's are defined by $u_i = u_{i-1} +$ 0x9e3779b9 (mod 2^{32}) with $u_1 =$ 0xb7e15163 ($i > 1$).

- F Function: $F(K, X) = Y$
  The round function F accepts the round key K of l bits and data X of $\frac{b}{2}$ bits, and outputs Y of $\frac{b}{2}$ bits as
  $$Y = H(K, X) \bmod 2^{b/2}.$$

- Encryption/Decryption
  The encryption and decryption processes are the same as in most Feistel-type ciphers. Let the plaintext P as $P = L_0 || R_0$, where $|L_0| = |R_0| = \frac{b}{2}$. Then the ciphertext C is computed as:
  $$\begin{aligned} L_i &= R_{i-1}, \\ R_i &= L_{i-1} \oplus F(K_i, R_{i-1}) \text{ for } i = 1, 2, \cdots, r. \end{aligned}$$
  The ciphertext is given by $C = R_r || L_r$. The decryption process is the same as the encryption process, except that the subkeys are applied in reverse order.

The security of the above block cipher depends heavily on the pseudorandomness of the underlying hash function. The three round version of the cipher will be provably secure against chosen ciphertext attacks under the assumption that the hash function is pseudorandom. And the four round version will be provably secure against combined chosen plaintext and ciphertext attacks under the same assumption. These are the main results of Luby and Rackoff [19]. Though no dedicated hash functions can be shown to be pseudorandom, it is widely accepted that well-designed hash functions should behave like a pseudorandom function. Otherwise, non-pseudorandomness could be exploited to break some of the design goal of the hash function. Thus the above three or four round Luby-Rackoff cipher seems practically secure for any block size.

Under the assumption that the secret key used is large enough to defeat an exhaustive key search attack, the most practical threat to iterative block ciphers comes from the differential and linear cryptanalysis methods [8, 20]. It seems quite unlikely that keyed hash functions used as round functions have any differential and/or linear characteristics useful enough to attack the cipher. In particular, truncation of keyed hash outputs is expected to provide stronger resistance against such attacks even if the keyed hash function has some vulnerabilities. In this respect, the most prudent choice of block size would be $b = l$, i.e., use only half of the hash output. For example, with this choice one can obtain a 128-bit cipher from MD5. We may even increase the number of rounds if four rounds turned out to be insufficient, of course at the expense of performance.

The speed of the cipher depends on the speed of the hash function used and the choice of the parameters r, b. More specifically, we can expect that the described cipher can encrypt data at the speed of $\frac{b}{rN}$ times the speed of the hash function. For example, the cipher using MD5 and $b = l = 128$ will be about 12 times slower for $r = 3$, and 16 times slower for $r = 4$ than MD5. For the same number of rounds, the cipher with $b = 2l$ will be two times faster than the cipher with $b = l$.

3 Stream Cipher-like Algorithms

3.1 Basic Construction

We can also construct block ciphers by using a keyed hash function as a key stream generator. The resulting ciphers run much the same way as stream ciphers. Suppose that the message X to be encrypted consists of n subblocks of b bits, i.e., $X = x_1 \| x_2 \| \cdots \| x_n$, where $|x_i| = b$ for $i = 1, 2, \cdots, n-1$ and x_n may be less than b bits. Then we may generate a sequence of b-bit random numbers from a keyed hash function and then encrypt X by xor-ing with this random numbers. However, such a cipher is vulnerable to the attack of reusing key streams under known plaintext attacks. This is true for any stream cipher if suitable precautions are not taken. We overcome this problem by adding kind of MAC for the ciphertext.

- Key stream generation:
  Choose a one-time random seed s of b bits and generate a sequence of b-bit random numbers $G(s) = z_1 \| z_2 \| \ldots \| z_n$ as

$$\begin{aligned} R &= H(K, s),\ R_0 = 0, \\ R_i &= H(R, R_{i-1}), \\ z_i &= R_i \bmod 2^b,\ i = 1, \ldots, n-1, \\ z_n &= R_i \bmod 2^{|x_n|}. \end{aligned}$$

  The key stream generator runs in output feedback mode and only the first b bits of the outputs are used as a key stream. Thus, if $b < l$, this will make it more difficult to derive the secret K from known key streams even with an exhaustive search.

- Encryption:
  Given a message X, compute the ciphertext $C = (C_1, C_2)$ as

$$\begin{aligned} C_1 &= X \oplus G(s) = x_1 \oplus z_1 \| \cdots \| x_n \oplus z_n, \\ C_2 &= s \oplus H(K, C_1, K) \bmod 2^b. \end{aligned}$$

  The second part of the ciphertext, C_2, plays two important roles. First, it securely conveys the one-time random secret s to the receiver. Second, it provides resistance against known plaintext attacks. That is, even if the key stream $\{z_i\}$ is revealed, an adversary cannot use this key stream to encrypt his chosen message.

- Decryption:
  The decryption process is straightforward: first recover s from C_2, generate $G(s)$ as before and then recover X from C_1.

It is easy to see that the above cipher is secure if the hash function used behaves like a pseudorandom function. One disadvantage of this encryption method, compared to the previous block cipher, is the message expansion due to the additional transmission of C_2. Therefore, this cipher seems not suitable for encryption of very short messages. However, we note that such expansion is necessary in any xor stream cipher to prevent known plaintext attacks.

The above cipher runs about $\frac{b+N}{b}$ times slower than the underlying hash function. For example, the cipher with $b = \frac{l}{2} = 64$ will be about 9 times slower than the underlying hash function with $l = 128$ and $N = 512$. This will be the most prudent choice for the block size b.

3.2 Improved Schemes

We can devise block ciphers with better performance by breaking down and incorporating the computation of C_2 into each step of key stream generation through ciphertext feedback mode. First consider the following cipher:

- Encryption: For a given plaintext $X = \{x_1, x_2, \cdots, x_n\}$, where $|x_i| = b$ for $i = 1, 2, \cdots, n-1$ and $|x_n| = t \leq b$, compute the ciphertext $C = \{c_1, c_2, \cdots, c_{n+1}\}$ as:

  1. choose a b-bit random seed s and compute $R = H(K, s)$.
  2. set $y_0 = c_0 = 0$ and do the following steps for $i = 1, 2, \cdots, n$, successively.

$$\begin{aligned} R_i &= y_i \| z_i = H(R, y_{i-1} \| c_{i-1}), \\ c_i &= x_i \oplus z_i, \end{aligned}$$

     where $|y_i| = l - b$, $|z_i| = b$ for $i = 1, 2, \cdots, n-1$ and $|z_n| = t$.

  3. compute the last ciphertext block as

$$c_{n+1} = s \oplus H(K, c_n \oplus c_{n-1}) \bmod 2^b.$$

  4. output the ciphertext $C = \{c_1, c_2, \cdots, c_{n+1}\}$.

- Decryption: The following process recovers the plaintext X from the ciphertext C.

  1. recover the secret seed s from the last three ciphertext blocks as

$$s = c_{n+1} \oplus H(K, c_n \oplus c_{n-1}) \bmod 2^b.$$

  2. compute $R = H(K, s)$.
  3. set $y_0 = c_0 = 0$ and do the following steps for $i = 1, 2, \cdots, n$, successively.

$$\begin{aligned} R_i &= y_i \| z_i = H(R, y_{i-1} \| c_{i-1}), \\ x_i &= c_i \oplus z_i, \end{aligned}$$

     where $|y_i| = l - b$, $|z_i| = b$ for $i = 1, 2, \cdots, n-1$ and $|z_n| = t$.

4. output $X = \{x_1, x_2, \cdots, x_n\}$.

In the above cipher the unused part of the i-th hash output y_i and the ciphertext c_i are used to compute the $(i+1)$-th random number R_{i+1}. Thus we can view that the block cipher runs in ciphertext feedback mode together with output feedback of hash function (if $b < l$). Note that the seed s, randomly chosen at each encryption time, is encrypted in the last block, so that it can be recovered only with knowledge of K. This enables the receiver to first recover s and then decrypt the ciphertext. We used $c_n \oplus c_{n-1}$, instead of c_n, in the last block encryption, since the last plaintext block x_n, thus c_n, may be very short.

On the other hand we may use a non-secret seed as in the following scheme.

- Encryption: The following process encrypts the plaintext $X = \{x_1, x_2, \cdots, x_n\}$, where $|x_i| = b$ for $i = 1, 2, \cdots, n-1$ and $|x_n| = t \leq b$, into the ciphertext $C = \{c_0, c_1, \cdots, c_{n+1}\}$.

  1. set c_0 to a randomly chosen b-bit number and compute $R = H(K, c_0)$.
  2. do the following steps for $i = 1, 2, \cdots, n$, successively.

$$\begin{aligned} R_i &= y_i \| z_i = H(R, y_{i-1} \| c_{i-1}), \\ c_i &= x_i \oplus z_i, \end{aligned}$$

  where $|y_i| = l - b$, $|z_i| = b$ for $i = 1, 2, \cdots, n-1$ and $|z_n| = t$.

  3. compute the last ciphertext block as

$$c_{n+1} = H(R, y_n \| c_n) \bmod 2^b.$$

  4. output the ciphertext $C = \{c_0, c_1, \cdots, c_{n+1}\}$.

- Decryption: The following process recovers the plaintext X from the ciphertext C.

  1. compute $R = H(K, c_0)$.
  2. do the following steps for $i = 1, 2, \cdots, n$, successively.

$$\begin{aligned} R_i &= y_i \| z_i = H(R, y_{i-1} \| c_{i-1}), \\ x_i &= c_i \oplus z_i, \end{aligned}$$

  where $|y_i| = l - b$, $|z_i| = b$ for $i = 1, 2, \cdots, n-1$ and $|z_n| = t$.

  3. check that $c_{n+1} = H(R, y_n \| c_n) \bmod 2^b$. If the check succeeds, output $X = \{x_1, x_2, \cdots, x_n\}$. Otherwise, output NULL.

The above cipher requires one block more transmission. But this enables the receiver to check the legitimacy of the ciphertext, and thus to remove the possibility of being exploited as a decryption oracle under a chosen ciphertext attack scenario. The public seed s may be replaced with other data, such as date/time at the encryption time.

The described two variants of the cipher runs a little faster than the original one. They are about $\frac{N}{b}$ times slower than the underlying hash function. For example, the cipher with

$b = \frac{l}{2} = 64$ runs about 8 (compared to 9 in the previous scheme) times slower than the underlying hash function with $l = 128$ and $N = 512$.

Finally, note that as far as we use one extra block c_{n+1} to encrypt one-time random secret (as in the first variant) or to protect the last plaintext block (as in the second variant), it is not necessarily required to use a different session key R for each encryption. However, the use of different session keys is almost free in out setting since it only requires one computation of compression function.

4 Randomized MAC from Hash Functions

4.1 Related Work

There have been published a number of papers dealing with construction and analysis of MACs from hash functions and block ciphers. In particular, much attention has been paid to MAC constructions from keyed hash functions in recent years (e.g., see [25, 17, 11, 21, 6, 3, 22]), probably because one can directly use widespread implementations of hash functions such as MD5 and SHA, and because they are faster than MACs from block ciphers such as DES.

Two preferred methods for constructing MACs from keyed hash functions are the envelope method and the HMAC construction. In the envelope method, the MAC for message X is generated as $MAC(X) = H(K, X, K)$ with one key or as $MAC(X) = H(K_1, X, K_2)$ with two keys [25, 17, 21]. These MACs overcome some weaknesses existing in the secret prefix method (e.g., an appending attack) and in the secret suffix method (e.g., an off-line collision search attack on the hash function) (see [21, 17] for further information). The HMAC method produces MACs as $MAC(X) = H(K, H(K, X))$ or $MAC(X) = H(K_1, H(K_2, X))$ [17, 3][1] These constructions are in popular use, since they are simple, easy to implement with existing hash codes, and have some evidence for security under reasonable assumptions on the hash function [2, 3]. All these MAC constructions, however, are susceptible to birthday-type attacks against MAC forgery and key recovery described in [21, 22], though the required number of known text-MAC pairs is impractically large for most choices of parameters.

On the other hand, Bellare et al. [6] proposed a new approach for MAC construction, called XOR MAC, based on any finite pseudorandom function. It is a first MAC algorithm allowing parallel processing and incrementality. For example, when using a hash function, to authenticate a message $X = x_1 \| x_2 \| \cdots \| x_n$, the sender picks at random R and computes

$$Z = H(0, R, K) \oplus H(1, < 1 >, x_1, K) \oplus H(1, < 2 >, x_2, K) \oplus \cdots \oplus H(1, < n >, x_n, K),$$

where $< i >$ denotes the binary representation of block index i. Then the MAC of X consists of $MAC(X) = \{R, Z\}$. This is the randomized XOR MAC, called XMACR, based on the hash function. They also presented the counter-based XOR MAC, called XMACC, where a counter C is used instead of a random R. We note that it would be better to use the secret

[1] More precisely, the prepended secret key should be padded with some padding strings to one complete block, so as to give the effect of using a random and secret initial value. This is also true for the envelope method in actual implementations.

prefix method for the computation of Z in XMACR to prevent the off-line collision search attack on the hash function.

The above XOR MAC schemes seem to resist the general attack by Preneel and van Oorschot [21, 22] due to the involvement of a random or counter number. However, there exists another MAC forgery attack on the XMACR scheme, as shown in [6]. That is, we can forge a MAC with high probability if about $2^{t/2}$ MAC queries are available (assuming $|R| = t$). For example, from three MACs for messages $X_1 = x_1 \| x_2, X_2 = x_1' \| x_2$ and $X_3 = x_1 \| x_2'$, i.e.,

$$\begin{aligned} Z_1 &= H(0, R_1, K) \oplus H(1, <1>, x_1, K) \oplus H(1, <2>, x_2, K), \\ Z_2 &= H(0, R_2, K) \oplus H(1, <1>, x_1', K) \oplus H(1, <2>, x_2, K), \\ Z_3 &= H(0, R_3, K) \oplus H(1, <1>, x_1, K) \oplus H(1, <2>, x_2', K), \end{aligned}$$

we can easily see that if we can find two MACs for X_1 and X_2 with $R_1 = R_2$, then the MAC for $X_4 = x_1' \| x_2'$ can be obtained by

$$Z_4 = Z_1 \oplus Z_2 \oplus Z_3 = H(0, R_3, K) \oplus H(1, <1>, x_1', K) \oplus H(1, <2>, x_2', K).$$

Finding Z_1 and Z_2 with $R_1 = R_2$ requires about $2^{t/2}$ MAC queries for X_1 and X_2.

4.2 Our Construction

We now present a new MAC construction from keyed hash functions. The basic idea is to generate a completely randomized MAC, so that known MACs do not provide any useful information for MAC forgery. The processes for MAC generation and verification are as follows:

- MAC generation: $MAC(X) = \{D_1, D_2\}$

$$\begin{aligned} S &= H(K, X), \\ D_1 &= H(K, R, S) \bmod 2^{b-t}, \\ D_2 &= R \oplus H(K, D_1, S) \bmod 2^t, \end{aligned}$$

  where R is a random t-bit number chosen by the sender ($0 \le t < b \le t + l$).

- MAC verification:

$$\begin{aligned} S' &= H(K, X), \\ R' &= D_2 \oplus H(K, D_1, S') \bmod 2^t, \\ D_1 &= H(K, R', S') \bmod 2^{b-t} \ ? \end{aligned}$$

The secret key is often recommanded to be at least l bits long. The length of MAC is b bits, and the parameter t determines the level of randomness involved and the probability of a randomly guessed MAC being correct (which is $2^{-(b-t)}$). An appropriate choice of b and t for MD4 family hash functions would be $b = 2t = 128$. Note that taking $t = 0$ in our scheme results in the HMAC construction [3] if the secret key K is padded to an N-bit block in each

computation of the hash function. Thus we can view that our construction corresponds to a randomized version of HMAC, which we call HR-MAC.

Compare our MAC with Bellare et al.'s XMACR with one message block (the whole message X), which consists of $\{R, Z\}$, where $Z = H(0, R, K) \oplus H(1, X, K) \bmod 2^t$ with $|R| = t$. HR-MAC requires compression of one more block. However, in HR-MAC, the number R randomly chosen each time is hidden from, and more tightly combined to, the MAC. This increases the complexity of MAC forgery under known/chosen text attacks, since it is hard to even verify that internal collisions occur in S, and effectively defeats the cancellation attack applied to XMACR. We thus can see that HR-MAC is more robust than XMACR, in particular when used to construct XOR MACs.

HRX-MAC (Randomized XOR MAC from hash functions): For fast authentication in high speed applications or in applications where the message needs to be frequently updated (e.g., as in databases), we need an extremely fast MAC algorithm. The XOR MAC approach proposed by Bellare et al. [6] provides a good alternative for such applications. Our HR-MAC can be easily modified to XOR MAC; just divide the message X into subblocks $\{x_i\}_{i=1}^n$ of predetermined length, say m, and compute the value of S as

$$S = \bigoplus_{i=1}^{n} H(K, < i >, x_i),$$

where the block index i is represented as a 32-bit number. The efficiency of this XOR MAC is slightly worse due to the repeated use of the secret K.

We may allow the sender to determine the length m of subblocks to be divided and involve m in the computation of a MAC, i.e.,

$$\begin{aligned} D_1 &= H(K, m, R, S) \bmod 2^{b-t}, \\ D_2 &= R \oplus H(K, m, D_1, S) \bmod 2^t. \end{aligned}$$

Of course, m should be transmitted in this case. This may provide better flexibility for MAC processing according to applications.

5 Conclusion

We have presented fast encryption and authentication algorithms constructed from keyed hash functions. These algorithms may be useful in many applications, since we can use widely deployed hash implementations as a subroutine for coding these algorithms. This will result in easy implementations and efficient storage usage.

References

[1] R.Anderson and E.Biham, Two practical and provably secure block ciphers: BEAR and LION, *Fast Software Encryption*, LNCS 1039, Springer-Verlag, 1996.

[2] M.Bellare, R.Canetti and H.Krawczyk, How to key Merkle-cascaded pseudorandomness and its concrete security, http://www.research.ibm.com/security/.

[3] M.Bellare, R.Canetti and H.Krawczyk, Keying hash functions for message authentication, presented at the 1996 RSA Data Security Conference, San Francisco, Jan. 1996, http://www.research.ibm.com/security/.

[4] M.Bellare and S.Goldwasser, Verifiable partial key escrow, Technical Report number CS95-447, Dept. of CS and Engineering, UCSD, 1995.

[5] M.Bellare and S.Goldwasser, Verifiable cryptographic time capsules: A new approach to key escrow, preprint.

[6] M.Bellare, R.Guerin and P.Rogaway, XOR MACs: new methods for message authentication using finite pseudorandom functions, *Advances in Cryptology-Crypto'95*, LNCS 963, Springer-Verlag, 1995, pp.15-28.

[7] M.Bellare and P.Rogaway, Random oracles are practical: A paradigm for designing efficient protocols, In *Proc. 1st ACM Conference on Computer and Communications Security*, ACM Press, 1993, pp.62-73.

[8] E.Biham and A.Shamir, *Differential cryptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.

[9] A.Bosselaers and B.Preneel, *Integrity primitives for secure information systems-Final report for RACE integrity primitives evaluation RIPE-RACE 1040*, LNCS 1007, Springer-Verlag, 1995.

[10] S.Bakhtiari, R.Safavi-Naini and J.Pieprzyk, Keyed hash functions, *Cryptography: policy and algorithms*, LNCS 1029, Springer-Verlag, 1996, pp.201-214.

[11] S.Bakhtiari, R.Safavi-Naini and J.Pieprzyk, Practical and secure message authentication, In *Proc. SAC'95*, may 1995, Ottawa, Canada, pp.58-71.

[12] H.Dobbertin, RIPEMD with two-round compress function is not collision-free, *J. Cryptology*, to appear.

[13] H.Dobbertin, Cryptanalysis of MD4, *Fast Software Encryption*, LNCS 1039, Springer-Verlag, 1996, pp.53-69.

[14] H.Dobbertin, Cryptanalysis of MD5, preprint.

[15] H.Dobbertin, A.Bosselaers and B.Preneel, RIPEMD160: A strengthened version of RIPEMD, *Fast Software Encryption*, LNCS1039, Springer-Verlag, 1996, pp.71-82.

[16] B.Kaliski and M.Robshaw, Fast block cipher proposal, *Fast Software Encryption*, LNCS 809, Springer-Verlag, 1994, pp.33-40.

[17] B.Kaliski and M.Robshaw, Message authentication with MD5, *CryptoBytes*, 1(1), 1995, pp.5-8.

[18] S.Lucks, Faster Luby-Rackoff ciphers, *Fast Software Encryption*, LNCS1039, Springer-Verlag, 1996, pp.189-203.

[19] M.Luby and C.Rackoff, How to construct pseudorandom permutations from pseudorandom functions, *SIAM J. Computing*, 17(2), 1988, pp.373-386.

[20] M.Matsui, Linear cryptanalysis method for DES cipher, *Advances in Cryptology-Eurocrypt'93*, LNCS 765, Springer-Verlag, 1994, pp.386-397.

[21] B.Preneel and P.C. van Oorschot, MDx-MAC and building fast MACs from hash functions, *Advances in Cryptology-Crypto'95*, LNCS 963, Springer-Verlag, 1995, pp.1-14.

[22] B.Preneel and P.C. van Oorschot, On the security of two MAC algorithms, *Advances in Cryptology-Crypto'96*, LNCS 1070, Springer-Verlag, 1996, pp.19-32.

[23] R.Rivest, The MD5 message digest algorithm, RFC 1321, 1992.

[24] NIST, Secure hash standard, *FIPS PUB 180-1*, 1995.

[25] G.Tsudik, Message authentication with one-way hash functions, *ACM Computer Communications Review*, 22(5), 1992, pp.29-38.

[26] Y.Zheng, J.Pieprzyk and J.Sebberry, HAVAL - A one-way hashing algorithm with variable length of output, *Advances in Cryptology-Auscrypt'92*, LNCS 718, Springer-Verlag, 1993, pp.83-104.