1 Description of Two-Track-MAC

1.1 Introduction

We will present a new message authentication code. It is based on a two trail-construction, which underlies the unkeyed hash function RIPEMD-160. It is in comparison with the MDx-MAC (see [1]) based on RIPEMD-160, much more efficient on short messages (that is on messages of 512 or 1024 bits) and percentage-wise a little bit more efficient on long messages. Moreover, it handles key-changes very efficiently. This positive fact remains if we compare our Two-Track-MAC with HMAC (see [2]) based on RIPEMD-160.

We will give a textual description of our proposal, with diagrams as well as pseudo-code. For a description of RIPEMD-160 itself we refer to [3] (although it is reflected in the pseudo-code provided here).

1.2 Presentation

The unkeyed hash function RIPEMD-160 (see [3]) uses two trails (in its compression function). If we separate those two trails then each trail can be seen as a transformation of a 160-bit input $I$, controlled by a message $M$, consisting of sixteen words of 32 bits. Those 160 bits of the input $I$ (and of the output) consist of five words of 32 bits. Call the different trails $L(I, M)$ and $R(I, M)$ (left respectively right trail for an input $I$ and a message $M$), then our proposal for a MAC on a relative short message $M$ (of 512 bits) and a key $K$ of 160 bits is (in short notation)

$$L(K, M) - R(K, M).$$

Or as $L(K, M)$ can be viewed as five words $A_i$ of 32 bits : $(A_0, A_1, A_2, A_3, A_4)$, and similarly the value $R(K, M)$ as $(B_0, B_1, B_2, B_3, B_4)$, we get an output $E = (E_0, E_1, E_2, E_3, E_4)$ of five 32-bit words. Here

$$E_i = A_i - B_i \text{ (subtraction modulo } 2^{32}) \text{ for } i = 0, 1, 2, 3, 4.$$ 

Then the 160-bit string $E$ is the MAC of the 512-bit message $M$.

If the message is longer, i.e. \( M = M_1M_2M_3\cdots M_n \) where each \( M_i \) is of length 512 bits, we define, using a new operation \( L^* \) and a new operation \( R^* \), the 160-bit quantity \( A \), respectively the 160-bit quantity \( B \). 

\[
A = (A_0, A_1, A_2, A_3, A_4) = L^*(K, M_1),
\]

where each \( A_i \) is a 32 bit word. And 

\[
B = (B_0, B_1, B_2, B_3, B_4) = R^*(K, M_1)
\]

as the result of the right trail. The operation \( L^* \) is based on the operation \( L \), which had a straightforward inverse operation on the first (160 bits long) argument. This new operation \( L^* \) has a simple feedback with the first argument, i.e.

\[
L^*(I, M) = L(I, M) - I
\]

(this is five times a substraction modulo \( 2^{32} \)). Similarly the operation \( R^* \) is defined in shorthand as

\[
R^*(I, M) = R(I, M) - I
\]

(this is five times a substraction modulo \( 2^{32} \)). Now we introduce two 160-bit blocks \( C \) and \( D \) of five 32-bit words, \( C = (C_0, C_1, C_2, C_3, C_4) \) and \( D = (D_0, D_1, D_2, D_3, D_4) \), which are now defined as follows

\[
\begin{align*}
C_2 &= A_3 - B_0, \\
C_3 &= A_4 - B_1, \\
C_4 &= A_0 - B_2, \\
C_0 &= (A_1 + A_4) - B_3, \\
C_1 &= A_2 - B_4, \\
D_1 &= (A_4 + A_2) - B_0, \\
D_2 &= A_0 - B_1, \\
D_3 &= A_1 - B_2, \\
D_4 &= A_2 - B_3, \\
D_0 &= A_3 - B_4.
\end{align*}
\]

All substractions and additions are modulo \( 2^{32} \). These 160-bit blocks \( C \) and \( D \) are the starting values for the left, respectively, right trail to incorporate the next 512-bit message block \( M_2 \). If there are more message blocks \( M_i \) the iteration is the same. So we have

\[
\begin{align*}
HL(1) &= A = L^*(K, M_1) \\
HR(1) &= B = R^*(K, M_1)
\end{align*}
\]

and then iteratively the three operations:

\[(A, B) \rightarrow (C, D)\]

\[HL(i) = A = L^*(C, M_i)\]

\[HR(i) = B = R^*(D, M_i)\]

Once we have \(HL(n)\) and \(HR(n)\) we define our MAC as \(TTMAC(K, M)\) by \(HL(n) - HR(n)\) (five times a substraction modulo \(2^{32}\)).

1.3 More details

Before processing a message we apply padding so that the result is a multiple of 512 bits. We use the same rule as in the RIPEMD-160 hash function. After the message a single 1-bit is appended. Next a number of 0-bits are appended so that the result has a bitlength 64 less than a multiple of 512. Finally the 64-bit representation of the length (mod \(2^{64}\)) of the original message is appended as two 32-bit words with least significant word first.

Regarding converting between streams of bytes and 32-bit words, the convention is little-endian (as in RIPEMD-160).

We have added one more feature as an extra defence against prefix-attacks on the MAC algorithm: the role of the left and right trails is interchanged for the last message block. This means that the description given above should be altered as follows.

For a message of a single block (512 bits after padding) the MAC is computed as

\[R(K, M) - L(K, M).\]

Or in detail, we compute \(A = (A_0, A_1, A_2, A_3, A_4) = R(K, M)\) and \(B = (B_0, B_1, B_2, B_3, B_4) = L(K, M)\). We get the output \(E = (E_0, E_1, E_2, E_3, E_4)\) of five 32-bit words. Here

\[E_i = A_i - B_i \text{ (substraction modulo } 2^{32}\text{) for } i = 0, 1, 2, 3, 4.\]

Then the 160-bit string \(E\) is the MAC of the 512-bit message \(M\).

For a message of \(n\) 512-bit blocks after padding, the description of section 1.2 is valid until the last iteration. This becomes

\[(A, B) \rightarrow (C, D)\]

\[HL(n) = A = R^*(C, M_n)\]

\[HR(n) = B = L^*(D, M_n)\]

Then we define our 160-bit long MAC result as \(TTMAC(K, M)\) by \(HL(n) - HR(n)\) (five times a substraction modulo \(2^{32}\)).

1.4 Output transformation for shorter bitlengths

The output length of the TTMAC algorithm can be shorter than the full length of 160 bits (the computation of which has been described above). All output lengths in multiples of 32 bits up to the key length of 160 bits are supported in the following manner.

Let the normal 160-bit result be $E = (E_0, E_1, E_2, E_3, E_4)$, and denote the final (shortened) MAC result with $F$, consisting of $t$ 32-bit words $F_i$ ($t = 1, 2, 3, 4$). For a 32-bit MAC we compute (using addition modulo $2^{32}$)

$$F_0 = E_0 + E_1 + E_2 + E_3 + E_4.$$ 

For a MAC result of 64, 96 or 128 bits we compute respectively the first two, the first three or all four of the following values (all additions are modulo $2^{32}$).

$$F_0 = E_0 + E_1 + E_3,$$
$$F_1 = E_1 + E_2 + E_4,$$
$$F_2 = E_2 + E_3 + E_0,$$
$$F_3 = E_3 + E_4 + E_1.$$

1.5 Outline of Two-Track-MAC

An outline of Two-Track-MAC, operating on a message of one (512-bit) block, is given in Figure 1. In this figure $f_i$ denotes the boolean functions used in the left and right trails, $c_i$ and $c'_i$ are the additive constants used in the left respectively right trail, $\rho$ and $\pi$ are two permutations on the message words. Note that the left and right trail have been switched since they operate on the last message block.

A high level view of Two-Track-MAC, operating on messages of one or two blocks, is shown in Figure 2 and Figure 3 respectively. For more than two blocks the iteration is the same. Note again the reversal of the role of the $L$ (left) and $R$ (right) trail for the last message block. The output transformation for shorter bitlengths (see section 1.4) is not shown in these figures.
Figure 1: Outline of the TTMAC algorithm for a message of a single block. Inputs are a 16-word message block $M_j$ ($0 \leq j \leq 15$) and a 5-word key $K_0K_1K_2K_3K_4$, output is a 5-word MAC value $E_0E_1E_2E_3E_4$. Appear in Proceedings of the 1st NESSIE Workshop, 15 pages, 2000.
Figure 2: High level view of TTMAC for a message of a single block.

Figure 3: High level view of TTMAC for a message of two blocks.

1.6 Pseudo-code for Two-Track-MAC

The TTMAC algorithm computes a 160-bit MAC value for an arbitrary message, under a 160-bit key. It works in an iterative manner and operates on 32-bit words. In each iteration a 16-word message block is applied to transform two 5-word chaining variables, according to the two trails of the RIPEMD-160 hash function. A feedback is used to make the trails uninvertible. Next a mixing between both trails defines the values of the chaining variables for the next iteration. The key used defines the initial value of both chaining variables. The role of the left and right trail is interchanged for the last message block. The MAC value is obtained by combining both trails after the last iteration. This value can be transformed into a shorter one (see section 1.4, not reflected in the pseudo-code). First we define all the constants and functions.

TTMAC: definitions

nonlinear functions at bit level: exor, mux, -, mux, -

\[
\begin{align*}
f(j, x, y, z) &= x \oplus y \oplus z \\
f(j, x, y, z) &= (x \land y) \lor (\neg x \land z) \\
f(j, x, y, z) &= (x \lor \neg y) \oplus z \\
f(j, x, y, z) &= (x \land z) \lor (y \land \neg z) \\
f(j, x, y, z) &= x \oplus (y \lor \neg z)
\end{align*}
\]

added constants (hexadecimal)

\[
\begin{align*}
c(j) &= 00000000_x \\
c(j) &= 5A827999_x \\
c(j) &= 6ED9EBA1_x \\
c(j) &= 8F1BBDC_x \\
c(j) &= A953FD4E_x \\
c'(j) &= 50A28BE6_x \\
c'(j) &= 5C4DD124_x \\
c'(j) &= 6D703EF3_x \\
c'(j) &= 7A6D76E9_x \\
c'(j) &= 00000000_x
\end{align*}
\]

selected of message word

\[
\begin{align*}
r(j) &= j \\
r(16..31) &= 7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8 \\
r(32..47) &= 3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12 \\
r(48..63) &= 1, 9, 11, 10, 0, 8, 12, 4, 13, 3, 7, 15, 14, 5, 6, 2
\end{align*}
\]
\[ r(64..79) = 4, 0, 5, 9, 7, 12, 2, 10, 14, 1, 3, 8, 11, 6, 15, 13 \]
\[ r'(0..15) = 5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3, 12 \]
\[ r'(16..31) = 6, 11, 3, 7, 0, 13, 5, 10, 14, 15, 8, 12, 4, 9, 1, 2 \]
\[ r'(32..47) = 15, 5, 1, 3, 7, 14, 6, 9, 11, 8, 12, 2, 10, 0, 4, 13 \]
\[ r'(48..63) = 8, 6, 4, 1, 3, 11, 15, 0, 5, 12, 2, 13, 9, 7, 10, 14 \]
\[ r'(64..79) = 12, 15, 10, 4, 1, 5, 8, 7, 6, 2, 13, 14, 0, 3, 9, 11 \]

amount for rotate left (rol)
\[ s(0..15) = 11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8 \]
\[ s(16..31) = 7, 6, 8, 13, 11, 9, 7, 15, 7, 12, 15, 9, 11, 7, 13, 12 \]
\[ s(32..47) = 11, 13, 6, 7, 14, 9, 13, 15, 14, 8, 13, 6, 5, 12, 7, 5 \]
\[ s(48..63) = 11, 12, 14, 15, 14, 15, 9, 8, 9, 14, 5, 6, 8, 6, 5, 12 \]
\[ s(64..79) = 9, 15, 5, 11, 6, 8, 13, 12, 5, 12, 13, 14, 11, 8, 5, 6 \]
\[ s'(0..15) = 8, 9, 9, 11, 13, 15, 15, 5, 7, 7, 8, 11, 14, 14, 12, 6 \]
\[ s'(16..31) = 9, 13, 15, 7, 12, 8, 9, 11, 7, 12, 7, 6, 15, 13, 11 \]
\[ s'(32..47) = 9, 7, 15, 11, 8, 6, 6, 14, 12, 13, 5, 14, 13, 13, 7, 5 \]
\[ s'(48..63) = 15, 5, 8, 11, 14, 14, 6, 14, 6, 9, 12, 9, 12, 5, 15, 8 \]
\[ s'(64..79) = 8, 5, 12, 9, 12, 5, 14, 6, 8, 13, 6, 5, 15, 13, 11, 11 \]

It is assumed that the message after padding consists of \( n \) 16-word blocks that will be denoted with \( M_i[j] \), with \( 0 \leq i \leq n - 1 \) and \( 0 \leq j \leq 15 \). The key used and the MAC value obtained consist of five words each, respectively \((K_0, K_1, K_2, K_3, K_4)\) and \((E_0, E_1, E_2, E_3, E_4)\). The symbols \( \oplus \) and \( \Box \) denote respectively addition and subtraction modulo \( 2^{32} \); \( \text{rol}_s \) denotes cyclic left shift (rotate) over \( s \) positions. The pseudo-code for Two-Track-MAC is then given below.
TTMAC: pseudo-code

\[ C_0 := K_0; C_1 := K_1; C_2 := K_2; C_3 := K_3; C_4 := K_4; \]
\[ D_0 := K_0; D_1 := K_1; D_2 := K_2; D_3 := K_3; D_4 := K_4; \]
\[ \text{for } i := 0 \text{ to } n - 1 \{ \]
\[ A_0 := C_0; A_1 := C_1; A_2 := C_2; A_3 := C_3; A_4 := C_4; \]
\[ B_0 := D_0; B_1 := D_1; B_2 := D_2; B_3 := D_3; B_4 := D_4; \]
\[ \text{if } (i != n - 1) \text{ for } j := 0 \text{ to } 79 \{ \]
\[ T := \text{rol}_x(j) (A_0 \oplus f(j, A_1, A_2, A_3) \oplus M_i[r(j)] \oplus c(j)) \oplus A_4; \]
\[ A_0 := A_4; A_1 := A_3; A_3 := \text{rol}_0(A_2); A_2 := A_1; A_1 := T; \]
\[ T := \text{rol}_x(j) (B_0 \oplus f(79 - j, B_1, B_2, B_3) \oplus M_i[r'(j)] \oplus c'(j)) \oplus B_4; \]
\[ B_0 := B_4; B_4 := B_3; B_3 := \text{rol}_0(B_2); B_2 := B_1; B_1 := T; \]
\[ \} \]
\[ \text{else for } j := 0 \text{ to } 79 \{ \]
\[ T := \text{rol}_x(j) (A_0 \oplus f(79 - j, A_1, A_2, A_3) \oplus M_i[r'(j)] \oplus c'(j)) \oplus A_4; \]
\[ A_0 := A_4; A_1 := A_3; A_3 := \text{rol}_0(A_2); A_2 := A_1; A_1 := T; \]
\[ T := \text{rol}_x(j) (B_0 \oplus f(j, B_1, B_2, B_3) \oplus M_i[r(j)] \oplus c(j)) \oplus B_4; \]
\[ B_0 := B_4; B_4 := B_3; B_3 := \text{rol}_0(B_2); B_2 := B_1; B_1 := T; \]
\[ \} \]
\[ A_0 := A_0 \oplus C_0; A_1 := A_1 \oplus C_1; A_2 := A_2 \oplus C_2; A_3 := A_3 \oplus C_3; \]
\[ A_4 := A_4 \oplus C_4; \]
\[ B_0 := B_0 \oplus D_0; B_1 := B_1 \oplus D_1; B_2 := B_2 \oplus D_2; B_3 := B_3 \oplus D_3; \]
\[ B_4 := B_4 \oplus D_4; \]
\[ \text{if } (i != n - 1) \{ \]
\[ C_2 := A_3 \oplus B_0; C_3 := A_4 \oplus B_1; C_4 := A_0 \oplus B_2; C_0 := (A_1 \oplus A_4) \oplus B_3; \]
\[ C_1 := A_2 \oplus B_4; \]
\[ D_1 := (A_4 \oplus A_2) \oplus B_0; D_2 := A_0 \oplus B_1; D_3 := A_1 \oplus B_2; \]
\[ D_4 := A_2 \oplus B_3; D_0 := A_3 \oplus B_4; \]
\[ \} \]
\[ E_0 := A_0 \oplus B_0; E_1 := A_1 \oplus B_1; E_2 := A_2 \oplus B_2; E_3 := A_3 \oplus B_3; \]
\[ E_4 := A_4 \oplus B_4; \]

For a short message of up to 512 bits (one 16-word block after padding), no feedback or mixing is required and the following simplified pseudo-code can be used.

**TTMAC (one message block): pseudo-code**

\[
A_0 := K_0; \quad A_1 := K_1; \quad A_2 := K_2; \quad A_3 := K_3; \quad A_4 := K_4;
B_0 := K_0; \quad B_1 := K_1; \quad B_2 := K_2; \quad B_3 := K_3; \quad B_4 := K_4;
\]

for \( j := 0 \) to \( 79 \) {
\[
T := \text{rol}_{s(j)} (A_0 \text{ } \Box \text{ } f(79 - j, A_1, A_2, A_3) \text{ } \Box \text{ } M[r'(j)] \text{ } \Box \text{ } c'(j)) \text{ } \Box \text{ } A_4;
A_0 := A_4; \quad A_4 := A_3; \quad A_3 := \text{rol}_{10}(A_2); \quad A_2 := A_1; \quad A_1 := T;
T := \text{rol}_{t(j)} (B_0 \text{ } \Box \text{ } f(j, B_1, B_2, B_3) \text{ } \Box \text{ } M[r(j)] \text{ } \Box \text{ } c(j)) \text{ } \Box \text{ } B_4;
B_0 := B_4; \quad B_4 := B_3; \quad B_3 := \text{rol}_{10}(B_2); \quad B_2 := B_1; \quad B_1 := T;
\]
\}

\[
E_0 := A_0 \text{ } \Box \text{ } B_0; \quad E_1 := A_1 \text{ } \Box \text{ } B_1; \quad E_2 := A_2 \text{ } \Box \text{ } B_2; \quad E_3 := A_3 \text{ } \Box \text{ } B_3;
E_4 := A_4 \text{ } \Box \text{ } B_4;
\]
2 No hidden weaknesses

We have inserted no hidden weaknesses in Two-Track-MAC. The definition of the left and right trails comes directly from the RIPEMD-160 hash function. The elements that we added to the design will be explained in the next sections.

3 Security

3.1 Short philosophy on the security

The idea for the security is simple: Now we have a so-called internal variable \((HL(i), HR(i))\) of 320 bits. Only in the case of very weak transformations a cryptanalist is allowed to hope on so-called inner collisions. In almost all attacks, which do not attack the very heart of the MAC (in our case the two trails of RIPEMD-160), forgery is based on inner collisions. (Another attack is possible if the MAC on a message contains all the information (or lacks ”only” 32 bits of information) of the chaining variable on a longer message, containing the first message as a prefix.) So here the worry for the cryptographer are the two trails of RIPEMD-160 itself. A single trail has one important weakness: it is a bijective operation, where the attacker can choose the bijection, which is parameterized by the 512-bit quantity \(M\). But as long as two trails are used, parametrized by the same 512-bit quantity M, and only a sum will come out in the open, there is no danger that an attacker can invert the operation. Moreover, we have used feedback to counter a straightforward inverse operation. (We did not use feedback on messages of 512 bits, because there the feedback from the left trail would cancel out the feedback from the right trail, in other words we do not need feedback there). All this makes the transformation of a new 512-bit message block on the 320-bit internal variable a one-way operation.

Suppose a cryptanalyst discovers a message \(N\), such that the function \(L^*(\cdot, N)\), from a 160-bit leftside argument to a 160-bit output, has only a few short cycles (and many relative short tails ending in those cycles). Such a discovery is useless because we have chosen to mix the outputs of the two trails, as soon as the functions \(L^*\) and \(R^*\) have outputted their results. (Otherwise it might be possible for the cryptanalyst to generate collisions for the left trail by appending blocks to the message, and separately trying to find collisions for the right trail.) This mixing is thorough in the following sense: Denote the outcome of the left trail by \(A\), the outcome of the right trail by \(B\) (as we did before), denote the MAC, in case we are done (in case, this was
3.2 Expected strength

In order to make a statement on the expected strength of our proposal Two-Track-MAC we first have to consider a number of general attacks on MAC algorithms. The strength against these attacks depends on: the length $k$ of the key which is 160 bits, the output length $m$ which can be between 32 and 160 bits (in 32 bit steps), and the length $l$ of the internal state which is 320 bits.

A first possible approach for an adversary is trying all possible keys (once he recovers the key he is able to forge the MAC for any message he chooses). For a key length $k$ and output length $m$, such an attack requires $2^k$ trials and $k/m$ known text-MAC pairs (for verification of the attack).

Alternatively an adversary can just guess the MAC corresponding to a chosen message. His success probability will be $1/2^m$ although this attack is not verifiable. The parameter $m$ should be chosen long enough according to the needs of the application.

The discovery of a so called internal collision leads to a verifiable forgery requiring only a single requested MAC (see [1]). When the internal state is $l$ bits long, about $2^{l/2}$ known text-MAC pairs are needed to find such an internal collision with a birthday attack. Next about $2^{l-m}$ chosen text-MAC pairs are needed to distinguish the internal collision from the external ones.

We summarize the difficulty of these attacks on the Two-Track-MAC algorithm in Table 1.

There are other attacks on MAC algorithms, which do not apply to our proposal. The key-less collision attack on the secret suffix method does not work here because the key is used as initial value. The extension attack on the secret prefix method is prevented here because the MAC contains only

<table>
<thead>
<tr>
<th>Attack</th>
<th>Trials</th>
<th>Success Prob.</th>
<th>Known Pairs</th>
<th>Chosen Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key search</td>
<td>$2^{160}$</td>
<td>$1/2^m$</td>
<td>$160/m$</td>
<td></td>
</tr>
<tr>
<td>Guessing the MAC</td>
<td>$2^{160}$</td>
<td>$1/2^m$</td>
<td>$2^{160}$</td>
<td>$2^{420-m}$</td>
</tr>
<tr>
<td>Birthday attack</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Strength of Two-Track-MAC against general attacks. The output length $m$ can take values from 32 to 160 bits.

160 out of 320 bits of information on the internal state. Furthermore as an extra precaution against such an attack we have swapped the role of the left and right trails for the last message block. The divide and conquer attack on the envelope method and on a variant of HMAC with two different keys also shows that the use of two keys in these schemes doesn’t improve the security against exhaustive key search as expected.

We have already discussed the role of the feedback and mixing operations that we added to our design in section 3.1.

It seems that the best hope for a cryptanalyst would lie in the two trails of RIPEMD-160 itself. RIPEMD-160 is widely believed to be a secure hash function, partly because its design has grown out of experience with the analysis of MD4, MD5 and RIPEMD. As protection against known attack strategies, the trails of RIPEMD-160 have been made longer (they consist of 5 rounds) and more different from each other. The algorithm has gained enough trust to be included in the new standard ISO/IEC 10118-3 (together with a scaled down version RIPEMD-128 and SHA-1).

An attack which finds a collision on both of the trails would apply to our proposal too. Although the authors of RIPEMD-160 don’t guarantee a higher security level (i.e. complexity more than $2^{80}$) for a 320-bit extension of RIPEMD (see [4]) because of the possibility of such an attack, in the case of Two-Track-MAC this attack is made very unlikely because of the secret key which serves as the (unknown) initial value of the two trails. An attack for which the difference of the left and right trail at the end of the computation stays the same would work as well, but seems unlikely for the same reason.

4 Strengths and advantages

The first advantage of Two-Track-MAC lies in its internal state which is 320 bits long. This is twice as long as for other MAC constructions based on RIPEMD-160 (or on SHA-1), e.g., HMAC and MDx-MAC. The consequence is that the complexity of the general birthday attack is raised to $2^{160}$ or more.

Furthermore Two-Track-MAC is more efficient than the other constructions, especially for short messages. Both HMAC and MDx-MAC require $n+1$ computations of the underlying compression function, when the message is $n$ blocks (of 512 bits) long. So that is relatively costly on messages of just one block. For Two-Track-MAC $n$ computations are sufficient, which is a substantial improvement for short messages.

Also a keychange will not slow down the speed of computation of TTMAC. In the case of HMAC or MDx-MAC a keychange costs respectively two or six extra computations with the underlying compression function.

5 Design rationale

Starting from the unkeyed hash function RIPEMD-160, it was a natural idea to use the two trails of the compression function to double the size of the internal state of our MAC construction. Feedback was added in order to make the trails uninvertible and a mixing operation to prevent attacks targeting a single trail (see section 3.1). The key is used as (unknown) initial value for both trails.

Since we now have an internal state of 320 bits, we can use 160 bits of information (the difference between the left and the right state variable, this depends on both trails) as the MAC output without compromising the internal state. The idea of interchanging the left and right trails for the last message block is an extra defence against expansion attacks, while it isn’t a burden on an implementation.

Because of this no extra computation of the compression function, using some secret key, is needed at the end of the MAC computation, and this makes our proposal more efficient than other MAC constructions, especially for short messages. Furthermore key changes are handled very efficiently since the key only serves as initial value of the two trails.

The output transformation for shorter bitlengths calculates the necessary number of output words, in such a manner that all of the normal output words are used.

6 Computational efficiency

Our MAC uses only a few percent more operations on a message as RIPEMD-160 would do to get an unkeyed hash of the message. This is already the case for the (shortest possible) message of 512 bits. Also a key-change will not slowdown the speed of the computation of the Two-Track-MAC.

On a Pentium our non-optimised reference code for Two-Track-MAC needs about 96 cycles per byte or 6135 cycles per block (of 64 bytes). By comparison (with a similar reference code, using the Nessie interface, for RIPEMD-160) we found that Two-Track-MAC achieves about 97% of the speed of RIPEMD-160. Since an optimised assembly code for RIPEMD-160 needs 1013 cycles per block on a Pentium (assuming that both code and data reside in the on-chip caches, see [4]), we estimate that a similar implementation for Two-Track-MAC would need about 1044 cycles per block (16 cycles per byte). The time needed for key-setup is negligible (10 cycles).

7 Avoiding implementation weaknesses

As far as we know Two-Track-MAC has no particular implementation weaknesses, since it has no special instructions which are data-dependent (e.g., containing ‘if’ statements). Timing attacks seem to be not applicable. Power analysis, especially on smart cards, might be possible. Standard techniques, using a combination of software and hardware measures, can be used as a protection against this.

References


   http://www.esat.kuleuven.ac.be/~bossela/ripemd160.html