The Anaconda hash functions

Thomsen, Søren Steffen

Publication date:
2008

Document Version
Early version, also known as pre-print

The ANACONDA hash functions

Søren Steffen Thomsen

DEPARTMENT OF MATHEMATICS

TECHNICAL UNIVERSITY OF DENMARK

Mat-Report No. 2008-05

November 2008

ISSN nr. 0904-7611

DTU

The Anaconda hash functions

Søren Steffen Thomsen

Department of Mathematics
Technical University of Denmark
Matematiktorvet 303S
DK-2800 Kgs. Lyngby
Denmark

1 Introduction

In this report, we describe the Anaconda hash functions. Anaconda is an attempt at designing a cryptographic hash function achieving the good diffusion properties of the 4×4 matrix structure used in the block ciphers Square [2] and Rijndael/AES [3, 11]. The challenge in this respect is to arrive at a larger state size, since the 128-bit state of these block ciphers is not large enough for hashing. There are, of course, several methods to increase the state size; one method is to increase the size of the matrix (this method was used in the `Grøstl` hash function [5], submitted to the NIST SHA-3 competition [10]), and another method is to increase the word size from 8 bits to, e.g., 64 bits. The latter method was followed in the design of Anaconda.

Unfortunately, 64-bit (and also 32-bit) S-boxes are infeasible to use in practice. Therefore, increasing the word size necessitates other changes as well. The non-linear transformation was therefore changed from operating on a word, to operating on a column, but in a bitslice fashion (as in Serpent [1]). Apart from introducing non-linearity, this also introduces diffusion within each column, and offers better protection against some side-channel attacks [6] than a table-based S-box.

On the other hand, a bitslice S-box does not introduce diffusion within a word. Hence, this must be taken care of by other means. The MixColumns transformation known from Rijndael ensures maximal diffusion *among* words, but it, too, does not introduce much diffusion within a word. Therefore, it was decided to abolish the principles underlying the MixColumns transformation, and instead focus on an efficient linear transformation, that provides a large amount of diffusion within words, and also provides (sub-optimal) diffusion among the words in each row. The diffusion within words is obtained via a primitive known from the SHA-2 hash functions [12].

The end result is a hash function that bears resemblance with both Rijndael and Serpent at the same time. It is simple and (in the author's view) elegant.

The name Anaconda refers to a class of hash functions returning outputs of any size between 1 and 512 bits. The variants returning outputs of sizes between 1 and 256 bits all use the same method, only the initial value and the amount of truncation taking place in the end are different. This method is described in detail in the following section. For the larger variants, another method is used; this method is described briefly in Section 8.

2 Specification of ANACONDA

The ANACONDA hash function H takes messages of length up to about 2^{64} bits and returns a hash result of n bits, where n is any number between 1 and 512. We now describe how to produce hash results up to 256 bits.

ANACONDA assumes a big-endian byte ordering. It applies a compression function $f : \{0,1\}^{512} \times \{0,1\}^{512} \to \{0,1\}^{512}$ in standard Merkle-Damgård mode [4,9]. The output of the final application of the compression function is truncated to an n-bit value.

The compression function f operates with a 16-word state, that is seen as a 4×4 matrix of words (as in Rijndael). A state $A = a_0 \| \cdots \| a_{15}$ is seen as the matrix

$$A = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{bmatrix}.$$

Each word is 64 bits in size. Let $\mathcal{W} = \{0,1\}^{64}$. The compression function takes a 512-bit message block and a 512-bit chaining value, forms a 1024-bit state viewed as the matrix above, and applies a number of rounds to the state. The final state is then truncated to 512 bits. The round function is now described.

2.1 Round function

The round function $p : \mathcal{W}^{16} \to \mathcal{W}^{16}$ is a permutation. It applies two different transformations: the linear transformation $\mathsf{lt} : \mathcal{W}^4 \to \mathcal{W}^4$, and the non-linear transformation $\mathsf{nt} : \mathcal{W}^4 \to \mathcal{W}^4$. We shall come back to how these transformations are defined in a moment. The round function operates on a state $A = a_0 \| \cdots \| a_{15}$ as follows.

$$\begin{array}{rcl} a_{15} & \leftarrow & a_{15} \oplus 1 \\ (a_0, a_1, a_2, a_3) & \leftarrow & \mathsf{lt}(a_0, a_1, a_2, a_3) \\ (a_7, a_4, a_5, a_6) & \leftarrow & \mathsf{lt}(a_7, a_4, a_5, a_6) \\ (a_{10}, a_{11}, a_8, a_9) & \leftarrow & \mathsf{lt}(a_{10}, a_{11}, a_8, a_9) \\ (a_{13}, a_{14}, a_{15}, a_{12}) & \leftarrow & \mathsf{lt}(a_{13}, a_{14}, a_{15}, a_{12}) \\ (a_0, a_4, a_8, a_{12}) & \leftarrow & \mathsf{nt}(a_0, a_4, a_8, a_{12}) \\ (a_1, a_5, a_9, a_{13}) & \leftarrow & \mathsf{nt}(a_1, a_5, a_9, a_{13}) \\ (a_2, a_6, a_{10}, a_{14}) & \leftarrow & \mathsf{nt}(a_2, a_6, a_{10}, a_{14}) \\ (a_3, a_7, a_{11}, a_{15}) & \leftarrow & \mathsf{nt}(a_3, a_7, a_{11}, a_{15}) \end{array}$$

See also Figure 1. In words, first a linear layer is applied to each row, and then a non-linear layer is applied to each column. The linear layer provides diffusion both on the bit-level and on the word-level (row-wise). The non-linear layer is a 4-bit S-box in bitslice mode, which provides diffusion on the word-level (column-wise). Notice that the order of the input words to the lt function is shifted for each row, and that the least significant bit of a_{15} is flipped in the beginning. These measures are in order to introduce asymmetry.

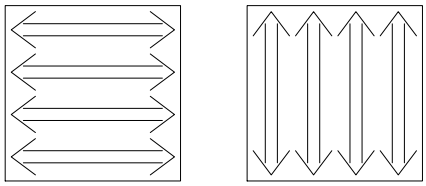

Fig. 1. The effects of the transformations lt and nt.

2.2 Compression function

The compression function takes the message block m and the chaining value h and forms from these the state $A = m\|h$. It then applies the round function p a number ℓ of times. ℓ is a security parameter, and hence its value may be chosen depending on the desired security level. We suggest $\ell = 16$. Since p is a permutation, no collisions are formed (yet).

After the ℓ applications of p, the compression function returns $\mathsf{trunc}_{512}(A)$, meaning the last 512 bits of A. Hence, this final truncation is the only operation that introduces collisions. More formally, the compression function f is defined as follows:

$$f(h, m) = \mathsf{trunc}_{512}(p^{\ell}(m\|h)).$$

2.3 Linear transformation lt

In the definition of lt, four transformations operating on a single word each are applied. Let these be Σ_i, $0 \leq i < 4$. They are defined as follows (where $a^{\lll r}$ means a rotated left by r positions).

$$\begin{array}{lcl}
\Sigma_0(a) & \leftarrow & a^{\lll 1} \oplus a^{\lll 20} \oplus a^{\lll 24} \\
\Sigma_1(a) & \leftarrow & a^{\lll 13} \oplus a^{\lll 22} \oplus a^{\lll 60} \\
\Sigma_2(a) & \leftarrow & a^{\lll 30} \oplus a^{\lll 50} \oplus a^{\lll 63} \\
\Sigma_3(a) & \leftarrow & a^{\lll 24} \oplus a^{\lll 51} \oplus a^{\lll 54}
\end{array}$$

Given the four input words (a, b, c, d), lt is defined as follows.

$$\begin{array}{lcl}
a & \leftarrow & \Sigma_0(a) \\
b & \leftarrow & \Sigma_1(b) \\
c & \leftarrow & a \oplus b \oplus c \\
d & \leftarrow & a \oplus (b^{\gg 1}) \oplus d \\
c & \leftarrow & \Sigma_2(c) \\
d & \leftarrow & \Sigma_3(d) \\
a & \leftarrow & a \oplus c \oplus d \\
b & \leftarrow & b \oplus (c^{\gg 3}) \oplus d
\end{array}$$

$a^{\gg r}$ means a shifted right by r bit positions (hence, the leftmost r bits of $a^{\gg r}$ are zeros). See Figure 2.

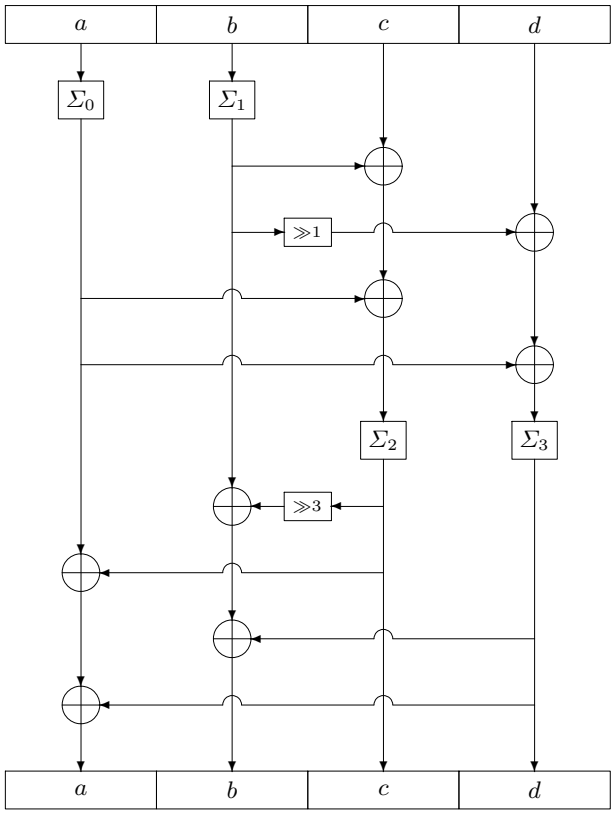

Fig. 2. The linear transformation lt.

2.4 Non-linear transformation nt

As mentioned, the non-linear transformation is a 4-bit S-box in bitslice mode. The S-box S used for nt is defined as follows.

x :	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S(x)$:	2	9	8	3	11	5	7	14	12	15	1	4	6	0	10	13

This S-box is, conceptually, applied as follows. Place the jth bit of the ith input word (counting from 0) in position (i, j) in a 4×64 matrix of bits. Apply the S-box to each of the 64 4-bit words defined by the columns of this matrix, where the bits in the top row are the most significant bits. Then, map the matrix back to four 64-bit words. In practice, the S-box application can be done via logical operations on the four 64-bit words. A method to do this is described in Section 5.

2.5 Padding

A message M (of length N bits) to be hashed is padded to a multiple of 512 bits in length as follows. First, a '1' bit is appended. Then, $-N-65 \bmod 512$ '0' bits are appended, and finally, a 64-bit representation of N is appended. This padding rule induces a maximum length of messages that can be hashed to $2^{64}-1$ bits (approximately 2^{55} message blocks).

2.6 The hash construction

The hash function applies the compression function f in Merkle-Damgård mode, with the addition that the final output is truncated to n bits. The initial value is the 512-bit representation of the output size n. To be more precise, the message M is padded to $M^+ = m_1 \| m_2 \| \cdots \| m_t$, and we set $h_0 = \langle n \rangle_{512}$. Let

$$h_i \leftarrow f(h_{i-1}, m_i) \quad \text{for } i = 1, \ldots, t.$$

Finally, let $H(M) = \mathsf{trunc}_n(h_t)$.

3 Some preliminary observations

An alternative representation of the compression function f, assuming a single 1024-bit input x and $P = p^\ell$, is

$$f(x) = P(x) \bmod 2^{512}.$$

Collisions can easily be found for the compression function: choose y_1 and y_2 to be distinct 1024-bit values such that $y_1 \equiv y_2 \pmod{2^{512}}$, and compute $x_1 = P^{-1}(y_1)$ and $x_2 = P^{-1}(y_2)$. Then the pair (x_1, x_2) forms a collision for the compression function. However, if P is a good permutation, then an attacker will have no direct control over x_1 and x_2.

A "meet-in-the-middle" preimage attack can be launched in time 2^{256} as follows. Let $y = H(M)$ be a target image. Compute $v_i = f(h_0 \| a_i)$ for 2^{256} arbitrary 512-bit values of a_i, $0 < i \leq 2^{256}$. Compute $b_i \| w_i = P^{-1}(z_i \| y)$ for arbitrary $(1024 - n)$-bit values of z_i, $0 < i \leq 2^{256}$. Find a match (i, j) between v_i and w_j. Then $H(a_i \| b_j) = y$. We note that the complexity is at least as high as the complexity of a brute force preimage attack, since $n \leq 256$. The memory requirements are 2^{256}.

The invertibility of the compression function also leads to almost trivial pseudo-attacks. For instance, a pseudo-preimage attack: given target image $y = H(M)$, choose arbitrary z, and compute $m \| h_0^* = P^{-1}(z \| y)$. Then m is a preimage of y when h_0^* is used as initial value of the hash function. We argue that h_0^* cannot be controlled (except by the method of the above preimage attack), and therefore we do not consider these pseudo-attacks a threat.

4 Design considerations

In this section, we explain some of the considerations that we made in the process of designing Anaconda.

4.1 The linear transformation

The linear transformation was inspired by the block cipher Serpent. The Σ transformations were inspired by the transformations with the same name in the SHA-2 hash functions.

Although not immediately clear, the overall design of the linear transformation is based upon a linear, binary code of length 8, dimension 4 and minimum distance 4.

Hence, the code is not MDS [8], as in Rijndael. However, minimum distance 4 is optimum for a binary code of the dimensions mentioned.

To see why it makes sense to talk about a binary code, look at what happens if a, b, c, d were bits, and we omit the shift operations. Then lt would have the specification

$$\begin{aligned} a' &\leftarrow a \oplus c \oplus d \\ b' &\leftarrow b \oplus c \oplus d \\ c' &\leftarrow a \oplus b \oplus c \\ d' &\leftarrow a \oplus b \oplus d, \end{aligned}$$

where the primed values are the new values of a, b, c, d. Hence, a generator matrix for the code is

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{bmatrix},$$

for which it is easily seen that the minimum distance is 4. Therefore, differences in d, $0 < d < 4$, of the words a, b, c, d spread to at least $4 - d$ output words.

When 64-bit words are used, the Σ functions ensure that all output words depend on all input words. On the other hand, the minimum distance of the code is no longer guaranteed to hold. However, if there is only a difference on a, then this will spread to at least a, c, and d. The inverse of lt has properties similar to lt itself.

The Σ functions. The purpose of the Σ functions is to cause diffusion within words. Σ functions are also used in the SHA-2 hash functions [12]. We define the class of Σ functions as the functions

$$\Sigma(a) \leftarrow a^{\lll r_1} \oplus a^{\lll r_2} \oplus a^{\lll r_3},$$

where a is a 64-bit value, and $0 \leq r_1, r_2, r_3 < 64$. Each choice of rotation values (r_1, r_2, r_3) defines a member of the class. Hence, a Σ function is completely defined by its rotation values. We now state and prove two theorems regarding Σ functions.

Theorem 1. *For any set of rotation values, Σ is a bijection, and $\Sigma^{64}(x) = x$.*

Proof. We first prove that $\Sigma^{64}(x) = x$. We have:

$$\begin{aligned} \Sigma^2(x) = {} & x^{\lll 2r_1} \oplus x^{\lll r_1+r_2} \oplus x^{\lll r_1+r_3} \oplus x^{\lll r_1+r_2} \oplus x^{\lll 2r_2} \oplus x^{\lll r_2+r_3} \oplus \\ & x^{\lll r_1+r_3} \oplus x^{\lll r_2+r_3} \oplus x^{\lll 2r_3} \\ = {} & x^{\lll 2r_1} \oplus x^{\lll 2r_2} \oplus x^{\lll 2r_3}. \end{aligned}$$

Repeating this reasoning, we have $\Sigma^{2^k}(x) = x^{\lll 2^k r_1} \oplus x^{\lll 2^k r_2} \oplus x^{\lll 2^k r_3}$. Since $x^{\lll 64d} = x$ for all integers d, we get $\Sigma^{64}(x) = x \oplus x \oplus x = x$. This also shows that $\Sigma^{64}(x) = \Sigma(\Sigma^{63}(x)) = x \iff \Sigma^{63}(x) = \Sigma^{-1}(x)$, and therefore, Σ is a bijection. We note that the proof extends to the general case of 2^k-bit words.

Theorem 2. *Exactly one or three of the rotation values r_1, r_2, r_3 are odd $\iff$ $\Sigma^i(x) \neq x$ for all x and all i, $1 \leq i < 64$.*

Proof. Since $\Sigma^{64}(x) = x$, we have that if $\Sigma^{32}(x) \neq x$, then $\Sigma^i(x) \neq x$ for all x and all $i < 64$. From the proof of Theorem 1, we know that $\Sigma^{32}(x) = x^{\lll 32r_1} \oplus x^{\lll 32r_2} \oplus x^{\lll 32r_3}$. If r_i is even, then $x^{\lll 32r_i} = x$, and otherwise, $x^{\lll 32r_i} = x^{\lll 32}$. Hence, if one or all three of the r_i are odd, then $\Sigma^{32}(x) = x^{\lll 32}$, and otherwise, $\Sigma^{32}(x) = x$. Again, the proof extends to the general case of 2^k-bit words.

Obviously, r_1, r_2, r_3 should be distinct – otherwise, Σ is only a rotation and does not provide any diffusion. We consider a Σ function with distinct rotation values "good" if

- $\Sigma^i(x) \neq x$ for all x and all i, $1 \leq i < 64$.
- The inverse function Σ^{-1} causes good diffusion.

Let us investigate some requirements for the inverse function Σ^{-1} to cause good diffusion. From the proof of Theorem 1, we have that $\Sigma^{-1}(x) = \Sigma^{63}(x)$, and hence we see that the inverse is the sum (XOR) of all terms of the form $x^{\lll a}$, where $a = r_{i_0} + 2r_{i_1} + 4r_{i_2} + 8r_{i_3} + 16r_{i_4} + 32r_{i_5} \bmod 64$, and $i_j \in \{1, 2, 3\}$. Intuitively, the inverse causing good diffusion corresponds to this sum having many terms that don't cancel out: for inputs of Hamming weight 1, the output Hamming weight is equal to the number of terms in the sum that haven't been cancelled out by other terms. For inputs of Hamming weight 2 to have high output Hamming weight, we need that $x^{\lll a} \oplus x^{\lll a+i}$ contains many terms for all i, $1 \leq i < 64$. There always exist inputs to Σ^{-1} of Hamming weight 3 with output Hamming weight 1, because if x has Hamming weight 1, then $\Sigma(x)$ has Hamming weight 3.

If r_1, r_2, r_3 are all odd, then a is always odd, and hence the inverse is the sum of at most 32 terms. We may obtain better results by requiring that r_1 is odd and r_2 and r_3 are even.

In order to find optimal rotation values, we performed a search. We fixed $r_1 = 1$ and searched for distinct, even values of r_2 and r_3 that would yield high output Hamming weights for inputs to Σ^{-1} with Hamming weights 1 and 2. Of course, this search could have been done without the theory described above, since the search space has size at most 2^{18}. It turned out that 60 pairs (r_2, r_3) produced equally good results: a weight 1 input to Σ^{-1} causes at least a weight 37 output, and a weight 2 input causes at least a weight 26 output. Note that translating the rotation values do not change the mentioned properties. By translating a set of rotation values, we mean adding the same even constant to all three rotation values.

In terms of the properties that we tried to optimise for Anaconda, the Σ functions used in SHA-2 are not optimal. For instance, $\Sigma_0^{\{512\}}$ (see [12]) has $(35, 18)$ in place of $(37, 26)$ (recall that these are the minimum output Hamming weights of Σ^{-1} given input Hamming weights $(1, 2)$). We assume there are other good reasons why the Σ functions used in SHA-2 were chosen as they were.

After having found the 60 equally good pairs, we chose four of them more or less at random, but such that the two even rotation values were quite different. We then searched for translations of these four sets of rotation values such that when used inside lt, a weight 1 input to $\mathsf{lt} \circ \mathsf{lt}$ yielded an output with a large Hamming weight. The first solution, for which the output Hamming weight was maximal, was chosen.

The reason why we chose to measure the diffusion of lt by using two applications of the linear transformation is that for only a single application, all translations have the same effect: a weight 1 input yields at least a weight 8 output. For $\mathsf{lt} \circ \mathsf{lt}$, with the Σ functions chosen for ANACONDA, a weight 1 input yields at least a weight 120 output.

4.2 The non-linear transformation

The S-box used in ANACONDA is a variant of the Serpent S-box numbered 0. The change as compared to the original Serpent S-box is based on the Serpent implementation by Osvik [13]: here, each output bit is not in its right place after the S-box application, and subsequent word moves are done implicitly. We would like to avoid this, and hence we have just taken the output bits in the order they come out of the S-box application. Furthermore, a logical negation was omitted in order to speed up the implementation. This change only corresponds to an affine transformation, and hence does not change the cryptographic properties of the S-box.

The S-box has the following properties [1]:

- A single input difference will always lead to an output difference in at least two bits
- The non-linear order is 2, and the non-linear order of each (single) output bit as a function of the input bits is 3
- Each differential characteristic has a probability of at most $1/4$
- Each linear characteristic has a probability in the range $1/2 \pm 1/4$.

An S-box in bitslice mode offers better protection against side-channel attacks such as timing analysis [6], than an S-box implemented via table look-ups.

4.3 Global diffusion

Every bit of the state depends on every bit of the message block three rounds after the message block is injected. In fact, most message bits affect all state bits after just two rounds, and most of the state bits are affected by all message bits after just two rounds.

The linear transformation ensures that each word in a row affects all words in the row. Combined with the non-linear transformation, this means that every word in the state affects every other word after just one round.

4.4 State size

Since the chaining value is at least twice as large as the output, ANACONDA is a wide pipe construction [7]. An important advantage of this construction is that it allows "slight failures" in the compression function. Consider, for instance, a collision attack on the compression function, where the chaining input is not under the control of the attacker. Assume this attack has complexity $2^{c/4}$, where c is the size of the chaining value. Although being an indication that the compression function does not provide ideal security, this attack would not constitute a collision attack on the hash function, since $c \geq 2n$ and therefore $2^{c/4} \geq 2^{n/2}$.

5 Implementation issues

As mentioned, the S-box was chosen from the set of Serpent S-boxes. These were implemented by Osvik [13], who focused on their performance on Pentium processors. A C implementation of the S-box used in Anaconda (based on Osvik's implementations) follows (`t` is a temporary variable).

```
#define S(x0,x1,x2,x3,t) do { \
    t   = x3; \
    x3 |= x0; \
    x0 ^= t;  \
    t  ^= x2; \
    t  =~ t;  \
    x3 ^= x1; \
    x1 &= x0; \
    x1 ^= t;  \
    x2 ^= x0; \
    x0 ^= x3; \
    t  |= x0; \
    x0 ^= x2; \
    x2 &= x1; \
    x3 ^= x2; \
    x2 ^= t;  \
    x1 ^= x2; \
  } while (0)
```

More efficient implementations are likely to exist, particularly on other processors than the Pentium. It is generally difficult to find optimal implementations of bitslice S-boxes.

Since no words are copied from one position in the state matrix to another, all computations can be done "in place", and therefore an implementation using 1024 bits of memory is possible.

An implementation of Anaconda in C has been developed [14] and run on an Intel Core2 Duo (E4600) 64-bit processor. With 16 rounds, the implementation reaches a speed of around 23 cycles/byte. The compiler used was Intel's C compiler version 10.1 (build 20080112) for Linux. The `gcc` compiler does not achieve the same speed for reasons that are unclear at this point. We note that improvements to the implementation are almost certainly possible.

6 Security claims

We claim that the best collision attack on Anaconda variants returning up to 256 bits has complexity around $2^{n/2}$, and the best preimage and second preimage attacks have complexity around 2^n.

7 Cryptanalysis

Further work on Anaconda was postponed due to the SHA-3 competition. The author decided to take part in another design, `Grøstl`, and hence, an investigation of the feasi-

bility of known attacks on ANACONDA has yet to be made. However, we note here that in each round, the 4-bit S-box is applied 256 times, totalling 4096 S-box applications over the 16 rounds. This number is huge compared to, e.g., the number of S-box applications in the 10 rounds of Rijndael, which is 160, or the number of S-box applications in the 32 rounds of Serpent, which is 1024. Of course, 4-bit and 8-bit S-boxes cannot be compared directly, and the number of degrees of freedom that an attacker has in the input to ANACONDA is higher than in the block ciphers. However, considering the large number of S-box applications combined with the good diffusion taking place in ANACONDA, we believe that differential, linear, and algebraic attacks will not work on ANACONDA.

8 Larger variants

One possible method of building a 512-bit hash function on the basis of the method described above is to use 128-bit words instead of 64-bit words. Although many processors provide SSE instructions on 128-bit vectors, these are not capable of performing rotations, and hence we estimate that there would be a rather large penalty in terms of efficiency. Instead, we chose to change the definition of the compression function to the following.

$$f(h, m) = \mathsf{trunc}_{512}(p^{2\ell}(m \| h)) \oplus h,$$

or, in the alternative representation (see Section 3),

$$f(x) = P^2(x) \oplus x \bmod 2^{512}.$$

The chaining input is fed forward to protect against the meet-in-the-middle preimage attack mentioned above, which would have complexity below 2^n for $n > 256$.

The security claims for the larger variants of ANACONDA are the same as for the shorter variants, except for second preimage resistance, which we claim to be at a level of 2^{n-k} compression function evaluations for a first preimage of 2^k blocks.

With 32 rounds, the larger variants perform at around 45 cycles/byte in the environment described above.

9 Summary

ANACONDA is a hash function built on some of the design principles underlying Rijndael, and also on some of those underlying Serpent. The compression function (for variants returning up to 256 bits) is a permutation followed by a truncation. The internal state size is at least twice the output size. Collisions can easily be found in the compression function (even assuming the permutation is ideal), and therefore the security proof of the Merkle-Damgård construction [4, 9] does not apply to ANACONDA. However, it does not seem possible to extend the collision attack on the compression function to the full hash function, assuming that the permutation contains no weaknesses.

For the larger variants returning more than 256 bits, it seems harder to find collisions for the compression function. The compression function is not invertible, due to a feed-forward of the chaining input. This feed-forward is omitted in the shorter variants in order to avoid having to store a copy of the 512-bit chaining input.

References

1. R. J. Anderson, E. Biham, and L. R. Knudsen. Serpent: A Proposal for the Advanced Encryption Standard. AES Algorithm Submission, 1998. Available: `http://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf` (2008/11/06).
2. J. Daemen, L. R. Knudsen, and V. Rijmen. The Block Cipher Square. In E. Biham, editor, *Fast Software Encryption 1997, Proceedings*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 1997.
3. J. Daemen and V. Rijmen. AES Proposal: Rijndael. AES Algorithm Submission, September 3, 1999. Available: `http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf` (2008/10/29).
4. I. Damgård. A Design Principle for Hash Functions. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1990.
5. `Grøstl` – a SHA-3 candidate. `http://www.groestl.info` (2008/11/03).
6. P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO '96, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.
7. S. Lucks. A Failure-Friendly Design Principle for Hash Functions. In B. K. Roy, editor, *Advances in Cryptology – ASIACRYPT 2005, Proceedings*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer, 2005.
8. F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland Mathematical Library, 1977.
9. R. C. Merkle. One Way Hash Functions and DES. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1990.
10. National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. In Federal Register Vol. 27, No. 212, November 2007.
11. National Institute of Standards and Technology/U.S. Department of Commerce. Federal Information Processing Standards Publication (FIPS PUB) 197. Advanced Encryption Standard (AES), November 2001.
12. National Institute of Standards and Technology/U.S. Department of Commerce. Federal Information Processing Standards Publication (FIPS PUB) 180-2. Secure Hash Standard, August 2002.
13. D. A. Osvik. Speeding up Serpent. In *Third AES Candidate Conference*, pages 317–329, 2000.
14. S. S. Thomsen. Website of the Anaconda hash functions. `http://www.mat.dtu.dk/people/S.Thomsen/anaconda` (2008/11/05).