CRUNCH

January 27, 2009

Contents

Part I
Specifications

1 Introduction

This paper proposes a secure hash algorithm CRUNCH. This algorithm is an iterative one-way hash function that can process a message to produce a condensed representation called a *message digest.* This algorithm enables the determination of the message's integrity: any change in the message will, with a very high probability, result in a different message digest. This property is useful in the generation and verification of digital signature and message authentication codes, and the generation of random numbers (bits).

The algorithm enables to obtain digests of 224, 256, 384 and 512 bits. First an encryption permutation based on an unbalanced Feistel scheme with expanding functions will be designed. This permutation will be a pseudorandom permutation from kn bits to kn bits using random expanding functions from n bits to $(k-1)n$ bits. Then a compression function is constructed by xoring two such permutations and choosing a number of bits depending on the desired length of the message digest.

The hash algorithm can be described in four stages: preprocessing, encryption permutation, compression function and hash computations. Preprocessing involves padding the message, setting an initialization vector and an initial value. The hash computation uses 2 encryption permutations, the compression function together with the Merkle-Damgård construction.

Key words: Hash Functions, Unbalanced Feistel Schemes, Expanding Functions, Cryptography with Random S-Boxes.

2 Definitions

2.1 Glossary terms

Bit	A binary digit having a value 0 or 1
Byte	A group of eight bits
Word	A group of either 32 bits (4 bytes) or 64 bits (8 bytes)

2.2 Algorithm parameters, symbols and terms

2.2.1 Parameters

The following parameters are used in the secure hash algorithm specifications on this standard.

M	Message to be hashed.

l	Length of the message M in bits.
p	Number of zeros appended to a message during the padding step.
$\tilde{M}$	The padded message.
m	Numbers of bits in a message block of $\tilde{M}$. $m = 1024 - \beta$.
$\tilde{M}^{(i)}$	i^{th} block of $\tilde{M}$, of size m.
I_t	The initial value for the compression function after t steps for the compression function. I_0 is the input value for the compression function.
I_t^i	The i^{th} 8-bit block of I_t.
$H^{(i)}$	The i^{th} hash value. $H^{(N)}$ is the *final* hash value and is used to determine the message digest.
β	The number of bits of the message digest.
F_d^k	An unbalanced Feistel scheme with expanding functions of d rounds applied on a k-block input.
G_β, G'_β	Encryption permutations used to get a β-bit message digest. G_β, G'_β are unbalanced Feistel schemes with expanding functions.
d_β	The number of rounds of an unbalanced Feistel scheme needed to get encryption permutations when the message digest has β bits.
g_j	Internal functions from 8 bits to 1024 bits used in the unbalanced Feistel scheme with expanding functions to get G_β.
g'_j	Internal functions from 8 bits to 1024 bits used in the unbalanced Feistel scheme with expanding functions to get G'_β.
C_β	The compression function.
K_t	32 bit word equal to the first 32 decimals of $8 \times \lvert\sin(t + 29)\rvert$.
$X_{i\ldots j}$	Block obtained by choosing the $j - i + 1$ bits from the i-th leftmost bit to the j-th leftmost bit of X where $X \in \{0,1\}^{1024}$. If $j < i$ then this block is void.

2.2.2 Symbols

The following symbols are used in the secure hash algorithms specifications, and each operates on w-bit words.

$\oplus$	Bitwise OR ("exclusive-OR") operation.
$\|\|$	Concatenation of blocks.
$\lfloor\ \rfloor$	Floor function.

3 Notations and conventions

The following terminology related to bit strings and integers will be used.

1. A *hex digit* is an element of the set $\{0, 1, \ldots, 9, a, \ldots, f\}$. A hex digit is the representation of a 4-bit string. For example, the hex digit "7" represents the 4-bit string "0111", and the hex digit "a" represents the 4-bit string "1010".

2. A *word* is a w-bit string that may be represented as a sequence of hex digits. To convert a word to hex digits, each 4-bit string is converted to his hex digit equivalent, as described in 1. above. For example, the 32-bit string

   $$1010\ \ 0001\ \ 0000\ \ 0011\ \ 1111\ \ 1110\ \ 0010\ \ 0011$$

   can be expressed as "a103fe23", and the 64-bit digit

   $$1010\ \ 0001\ \ 0000\ \ 0011\ \ 1111\ \ 1110\ \ 0010\ \ 0011$$

   $$0011\ \ 0010\ \ 1110\ \ 1111\ \ 0011\ \ 0000\ \ 0001\ \ 1010$$

   can be expressed as "a103fe2332ef301a".

   Throughout this specification, the "big-endian" convention is used when expressing both 32- and 64 bit words, so that within each word, the most significant bit is stored in the left-most position

4 Constants

This section details on the generation of constants needed in the CRUNCH algorithm. There are exactly $2 \times 16 \times 256 \times 32 + 28 = 262172$ constants of 32 bits. This represents 1 MB. For $t \in \{-28; -27; \ldots; 262143\}$ let K_t be the 32 bit word equal to the first 32 decimals of $8 \times |\sin(t + 29)|$. In other words, K_t are the decimals from 4 to 35 of $|\sin(t + 29)|$.

Here are the first 28 constants:
$K_{-28} = bb5523c2$, $K_{-27} = 463dbab4$, $K_{-26} = 210386db$, $K_{-25} = 0dee7777$,
$K_{-24} = abe07d78$, $K_{-23} = 3c3e3156$, $K_{-22} = 4182309b$, $K_{-21} = ea34a80a$,
$K_{-20} = 4c04c6c1$, $K_{-19} = 5a27bd7c$, $K_{-18} = fffadd8b$, $K_{-17} = 4ae6bdf5$,
$K_{-16} = 5c808910$, $K_{-15} = ecc38c9f$, $K_{-14} = 33ca1c73$, $K_{-13} = 4da0410b$,

Message digest (bits)	IV (in hex)
224	bb5523c2463dbab4210386db0dee7777abe07d783c3e31564182309b
256	bb5523c2463dbab4210386db0dee7777abe07d783c3e31564182309bea34a80a
384	bb5523c2463dbab4210386db0dee7777abe07d783c3e31564182309bea34a80a4c04c6c15a27bd7cfffadd8b4ae6bdf5
512	bb5523c2463dbab4210386db0dee7777abe07d783c3e31564182309bea34a80a4c04c6c15a27bd7cfffadd8b4ae6bdf55c808910ecc38c9f33ca1c734da0410b

Figure 1: IV Values

$K_{-12} = b0f12b10$, $K_{-11} = 02059a04$, $K_{-10} = 32f2d28f$, $K_{-09} = 4db63d57$,
$K_{-08} = b17882ec$, $K_{-07} = 1220a29f$, $K_{-06} = c50f3409$, $K_{-05} = 3e9fde46$,
$K_{-04} = 0f0e6f31$, $K_{-03} = 19b83eb0$, $K_{-02} = a6a86c39$, $K_{-01} = 2ad0a768$.

These constants have been chosen because it is quite easy to calculate them, and because it is very difficult to establish relationships between them. So we can consider these constants as random numbers. We will see later that it is very easy to change the algorithm so that these constants are not stored explicitly in memory (because it's clear putting a required of 1 MB of memory on some device can not easily be achieved). We will see that these constants can be computed on the fly. This might slow down the computation.

5 Preprocessing

Preprocessing shall take place before the hash computation begins. This preprocessing consists of three steps: padding the message, M, (Section 5), parsing the padded message into message blocks (Section 5.2) and setting the initialization vector IV to compute the initial input of the compression function C_β (Section 5.3). We define $IV = K_{-28}||...||K_{-28+\frac{\beta}{32}-1}$.

5.1 Padding the message

The message M shall be padded before hash computation begins. The purpose of this padding is to ensure that, in the padded message, the number of bits is a multiple of 1024 -β = m. Append the bit "1" to the end of the message followed by p zeroes and the binary representation of the length l of the message. k is chosen so that the total number of bits is a multiple of m. The padded message is denoted by $\tilde{M}$. N is the number of blocks of size m of $\tilde{M}$.

5.2 Parsing the message

$\tilde{M}$ is parsed into N m-bit blocks.

5.3 Setting the initialization vector and computing initial value

An initialization vector IV of size β is defined (see Figure 1) and the initial value for the compression function is obtained by concatenating IV and the leftmost m bits of the padded message (the first block of

$\tilde{M}$). This is given by

$$I_0 = IV\|\tilde{M}^{(0)}.$$

I_0 is of size 1024.

6 Encryption Permutations

6.1 Unbalanced Feistel Schemes with Expanding Functions

An unbalanced Feistel scheme with expanding functions enables to construct a pseudorandom permutation from kn bits to kn bits by using random functions from n bits to $(k-1)n$ bits. The first round of an unbalanced Feistel scheme with expanding functions is given in Figure 2. If d rounds are applied, the scheme is denoted F_d^k.

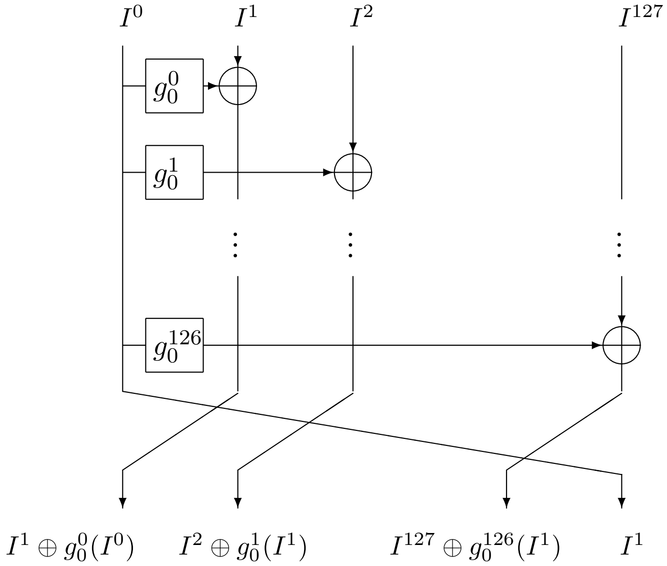


Figure 2: First round of an unbalanced Feistel scheme with expanding functions, for $k = 128$

When the internal functions are secret, generic attacks on these schemes have been studied in [9]. When $k+2 \leq d \leq 2k$, the best generic chosen plaintext attacks need $2^{(d-k-1)n}$ messages. For example, when $k = 128$ and $n = 8$, $d = \frac{\text{security}}{8} + 129$. Here the functions g_j are public and are not completely random (as explained in Section 6.2) since there are constraints related to memory space. Obviously, the security bounds are not the same when the internal functions are public.

For symmetric Feistel schemes, the following study is given in [4]. It refers to the model of indifferentiability for hash functions (see [6] and [3]). Two systems are described. In the first system, the random oracles g_i (inner functions) are randomly chosen and a permutation based on a symmetric Feistel scheme is constructed. In the second system, a permutation P is randomly chosen and the inner functions g_i are simulated by a simulator with oracle access to P. It is shown in [4] that, with 5 rounds, 4 messages are enough to distinguish both systems. This contrasts with the classical Luby-Rackoff result where 4 rounds

Message digest (bits)	Security (bits)	Unbalanced Feistel scheme F_d^{128} (Number of rounds)	Encryption Permutation (Number of rounds)
224	112	143	224
256	128	145	256
384	192	153	384
512	256	161	512

Figure 3: Secure Hash Algorithm Properties.

are enough to obtain a strong pseudo-random permutation from pseudo-random functions. However, with 6 rounds the distinguisher is not able to tell which system is used.

The encryption permutations of CRUNCH are based on unbalanced Feistel schemes with internal public expanding functions. The number of rounds will depend on the length of the message digest. Two facts are taken into account. Firstly, the security bound for chosen plaintext attacks of [9] are given when the inner functions are secret. This number of rounds is increased. Secondly, it is more secure to choose a number of rounds to make sure that all the Bytes are used the same number of times. Thus, we choose for the number of rounds a multiple of 128. If β is the length of the message digest, the number of rounds is $d_\beta = \beta$. Figure 6.1 shows the number of rounds needed to reach the security bound in the case of an unbalanced Feistel scheme and the number of rounds chosen for the encryption permutations depending on the length of the message digest.

Using Unbalanced Feistel schemes with expanding functions enables to construct two encryption permutations, which depend on $\beta \in \{224, 256, 384, 512\}$ and are denoted by G_β and G'_β. They are permutations from $(\{0,1\}^8)^{128} = \{0,1\}^{1024}$ to $\{0,1\}^{1024}$.

6.2 Internal Functions, Random S-Boxes

To generate the encryption permutations G_β and G'_β based on Unbalanced Feistel Schemes let us construct the internal functions g_i and g'_i, which will stand for random S-Boxes.

There are $2d$ internal functions to define, where d is the number of rounds. Each function maps 8 bits onto 1016 bits. For $0 \leq j \leq d-1$, g_j represents the internal function of the first permutation, and g'_j the internal function of the second permutation. The first 16 functions of each permutation will be completely independent, since they do not use the same constants K_t (see Section 4).

Let j be an integer between 0 and $d-1$, and i an integer between 0 and 255. We want to define the 1016 bits word of $g_j(i)$, and also the 1016 bit word equal to $g'_j(i)$.
Let $\bar{j} = j$ mod 128, and $q = \lfloor \frac{j}{16} \rfloor$. We define:

$$\alpha(j,i) = (2q+1)i \text{ mod } 256$$

And then

$$\gamma(j,i) = (j \text{ mod } 16) \times 256 \times 32 + 32 \times \alpha(j,i) - 4 \times \lfloor \frac{\bar{j}}{16} \rfloor$$

And:

$$\gamma'(j,i) = \gamma(j,i) + 16 \times 256 \times 32$$

Now, for γ integer between -28 and 131040 included, let Z_γ be the 1024 bit word equal to:

$$Z_\gamma = K_\gamma || K_{\gamma+1} || \dots || K_{\gamma+31}$$

Finally:

$$g_j(i) = (Z_{\gamma(j,i)})_{(8\bar{j}+8)\dots 1023} || (Z_{\gamma(j,i)})_{0\dots(8\bar{j}-1)}$$

and

$$g'_j(i) = (Z_{\gamma'(j,i)})_{(8\bar{j}+8)\dots 1023} || (Z_{\gamma'(j,i)})_{0\dots(8\bar{j}-1)}$$

The definition of $g_j(i)$ uses consecutive predefined constants, and for $j > 16$, $g_j(i)$ is equal to a concatenation of bits from $g_{j \text{ mod } 16}(\alpha(j,i))$ and from $g_{j \text{ mod } 16}(\alpha(j,i)-1)$.

7 The Compression Function

G_β and G'_β are the two encryption permutations obtained in the previous section. Then the compression function C_β is defined by

$$C_\beta(I) = (G_\beta(I) \oplus G'_\beta(I))_{0\dots(\beta-1)}$$

For example, if $\beta = 224$, $C_{224}(I)$ will take the 224 leftmost bits of $G_\beta(I) \oplus G'_\beta(I)$.

Taking the Xor of two secret permutations increases the security properties. It is shown in [7] and [8], that for the Xor of two secret permutations from L bits to L bits the security bound for the number of messages is 2^L.

8 Secure Hash Algorithm

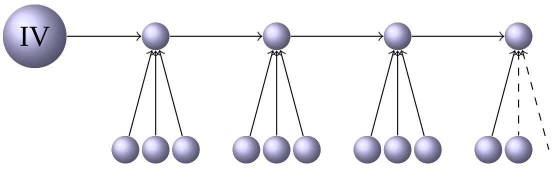
The CRUNCH algorithm is described in this section. For $\beta \in \{224, 256, 384, 512\}$ the final result will be a β-bit digest of the message. M is the message to be hashed. $\tilde{M}$ is the padded message which contains N m-bits blocks. IV is the initialization vector. The compression function C_β is used. The algorithm proceeds as follows (see also Figure 4):

```
$H^{(1)} = C_\beta(I_0)$
For $i = 1$ to $N-1$
    $I_i = H^{(i)} \| \tilde{M}^{(i)}$
    $H^{(i+1)} = C_\beta(I_i)$
EndFor
```

$H^{(N)}$ is the message digest.

The mode of operation of this hash algorithm is shown below:

CHAINING MODE



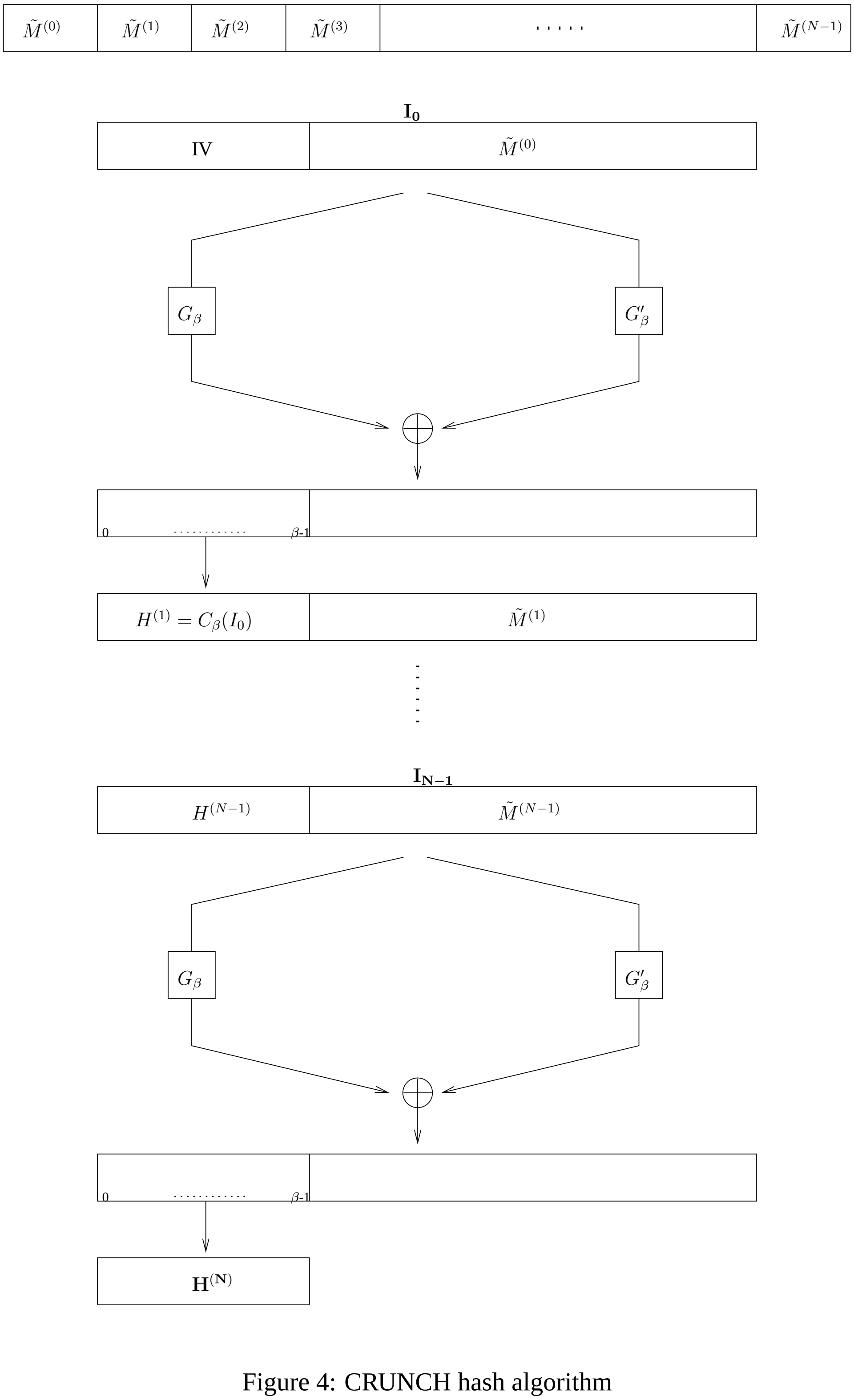


Figure 4: CRUNCH hash algorithm

9 Example

Message $M = abcdefgh$, of length equals to 48 bits.

We show below the first values of I and values of the internal functions.

Round number 0: For G

I_0 **bb**5523c2463dbab4210386db0dee7777abe07d783c3e31564182309bea34a80a
6162636465666768000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000040

$\oplus$

$g_0(0\mathrm{x}bb = 187)$ ..13f8c0b20b9d384d48893331e92c6802b6f5a3b0a72c464c9eefc234962ca3
ec9ebe525b8cb3184bc96d1ffffbfc52594644374cf98b67ea5bae2740b79752
3d402551ac2ca2f80d3ecc40220d807546ad796ebac3eab7010cac68bbe63f24
45cddbee1ff988470e9e0800ab94362b3b3c3381424cb2bbea0d7f0d4b0ff394

For G':

$I'_0 = I_0$ bb5523c2463dbab4210386db0dee7777abe07d783c3e31564182309bea34a80a
6162636465666768000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000040

$\oplus$

$g'_0(187)$ ..d52d5d17bb7f0b92a5f3dcf41b03ce0593a73286e7c464c0a02b87d9b0b7d2
6e36f0ee656c6ada8fd3dada4e31b07be4530ff72457d238d364376f3fe99687
9443a5a1f0befc6c7b7a6a5add59ddab042881e69af40af2fe23fe22081d9cbc
a2d4e666f5fa04d8f76a51d8981452bec505c680cd447f0e80091bd76c421854

Round number 1: For G

I_1 **46**db02f436278c6c4b0fe83c075b1fa95688db8c991d100d1cdf59dea284a98d
fcdd363eead470cbc96d1ffffbfc52594644374cf98b67ea5bae2740b797523d
402551ac2ca2f80d3ecc40220d807546ad796ebac3eab7010cac68bbe63f2445
cddbee1ff988470e9e0800ab94362b3b3c3381424cb2bbea0d7f0d4b0ff3d4**bb**

$\oplus$

$g_1(0\mathrm{x}46 = 70)$ ..a015cf80583afa5839eacf77370bc79dfcacd07143f8aa11f82db17d6a707e
805ab467a6cb2e92afb5fc0629587b046a4d8d8f0d977e69d4d78b7d3943c7ad
ede9bbaca3da9fcfa8ad03c4335824d25fc4acfa1226b5cc4c1168df66f1ecfb
9c60e1c0eee520d896558bd43f77939d365958c3308ebd84c51b4a5cb9c12ddc

For G':

I'_1 **80**0e9f5186c5bfb3a67507f9f574b9ae73da4abad9f53281221b1c33841fd80f
54938a000a0db20fd3dada4e31b07be4530ff72457d238d364376f3fe9968794
43a5a1f0befc6c7b7a6a5add59ddab042881e69af40af2fe23fe22081d9cbca2
d4e666f5fa04d8f76a51d8981452bec505c680cd447f0e80091bd76c421814bb

$\oplus$

$g'_1(0\mathrm{x}80 = 128)$ ..d131eb2f92e83c5d4aa942c8280dadcc7c1e0978279927c1e8ba4910f43bb7
a446ff06d03f50bf011e7943653a5f1a4453581260630fa9eeee4a35b29b8846
a34690e933ff6d45ce58185fddb5c2fff03e9545254b9062ae20bfedaf2db92e
9b24075425d050b2ce54e8fd83aac139aeeeda507b6303adbab56618f0c9df52

Round number 255:	For G
I_{255}	$5e\mathbf{48}8dffcd0874a7f0f73ffec4b29ef61421397c712dea6f4806f37130e79a56$ $2d18938eda567a836366d8d5b40d38a3af22d8ccf972983e153454c27b7ea6e9$ $0bab2f788bdf69dd8d359b1187e626006bf6367148b705a7d091d52256bf93c2$ $4cd7119fb0ba7c2561bbbf18d8a46f7c262d8a72b60f7c7852d21051af64a710$
$\oplus$	
$g_{255}(0\text{x}5e)$	$00\mathbf{ed}47461030e8c0e45140c543b1fb2b74ff84871336b192504f4131baaf648d$ $5b15ea6d6fc71334bc3d03abc30b4d02611c2af7c9a58c3ccc272ad6c951f42e$ $6ce6e6a38f380093126c7af579ffc4041d385840be70571627efa4f74823106a$ $7362947d8cb6dc1768f37665f7489960fa568c5924a55e666b280a93c6d0a338$
	For G'
I_{255}	$58\mathbf{a1}7b94a9b9eab01df56be1b7c1feaad1d1068d5937358f431908ebf9ddb854$ $72efe6bd29d720b1e718b338fb28e45c3663f6f503ddd49033a6a870a1146053$ $0e88bd1fb78b08a34adaced51584a0c0157e85293864b670d413ba5044ae8298$ $a27f9cabde29b3b34f57584dc38063ae00e1daa6173101909b2cca17a9499717$
$\oplus$	
$g'_{255}(0\text{x}58)$	$00\mathbf{63}da77d68354fd216d947c9683862f3ccad1d37fc5ddab239b7049d8f4a0ee$ $2ff65317a8a5fe8ae1050bc845fdfc76b89ddb285400ddf719dcb286937e1021$ $07ed4c9855931fd8464220a7fdcfdba34a22bd65c0e62022a351be59c38dcb06$ $3866c6444771b57f82d743e7adc17fdfa08a4650b402cf487ef47c185a95d33a$

Finally the hash value of ”abcdefgh”:

$$\mathbf{67}6b5aa202222a283e80a6a6411d588dc56aa544e9b3d978cbcae2ab61e6612b$$

We can notice that $67 = 48 \oplus ed \oplus a1 \oplus 63$

Part II

Computational efficiency

10 Memory Size/Speed

All of the constants to be stored (namely the S boxes) require around $1MB$ of storage. It is small enough to fit in the L2 (L3) caches of most of the recent general purpose microprocessors (x86 for example). Fitting the S boxes in the L1 caches might be much more difficult due to the very limited size of the L1 Dcache (typically 32 KB). However, experimentally we have checked that fitting in L1 cache will only give marginal performance improvement due to the good bandwidth of L2 caches and efficient prefetch mechanisms from L2 to L1. At the other end of the spectrum (on smart cards), 1 MB of storage requirements might be hard to accomodate within the current generation but we first believe that there will be a general trend to improve storage capacity and second we describe a mechanism to compute on the fly the S boxes allowing to reduce the storage requirements.

One excellent feature of CRUNCH is that besides data access (reading values from S boxes), the computational structure is remarkably simple: a loop around XOR operations (which is one of the simplest operations to perform, much simpler and efficient than an add or a shift operation). There is no complex control structure (which could lead to branch misprediction). In fact the key performance limiting factor of CRUNCH is data access.

11 Implementation

For the implementation, it is far better to avoid shifting data between each round of the permutations: this can be simply done by a clever adressing. Except maybe for 8 bit processors, because data is already divided into 8 bit-blocks, so it might be easier to shift them.

12 64-bit processors

Machine 1:
Processor: Intel Core 2 Duo E6400 @2.13GHz (cache L1 data 32KB, cache L2 2MB)
RAM: 4GB DDR2 dual channel
OS: kubuntu 8.04.1 64bits with KDE 3.5
compiler: icc v10.1
compilation options: -fast

Message digest (bits)	Message Size (MB)	Number of cycles	Speed (MB/s)
256	100	$16,95 * 10^9$	12,59
384	100	$29,62 * 10^9$	7,24
512	100	$46,97 * 10^9$	4,55

13 32-bit processors

Machine 1:
Processor: Intel Core 2 Duo E6400 @2.13GHz (cache L1 data 32KB, cache L2 2MB)
RAM: 4GB DDR2 dual channel
OS: kubuntu 8.04.1 64bits with KDE 3.5
compiler: icc v10.1
compilation options: -O3 -march=pentium4

Message digest (bits)	Message Size (MB)	Number of cycles	Speed (MB/s)
224	100	$25,16 * 10^9$	8,48
256	100	$29,87 * 10^9$	7,15
384	100	$52,36 * 10^9$	4,08
512	100	$86,42 * 10^9$	2,47

Machine 2:
Processor: Intel Core Duo T2300e @1.66GHz (cache L1 data 32KB, cache L2 1MB)
RAM: 1GB DDR2 dual channel
OS: kubuntu 8.04.1 32bits with KDE 3.5
compiler: icc v10.1
compilation options: -O3 -march=pentium4

Message digest (bits)	Message Size (MB)	Number of cycles	Speed (MB/s)
224	100	$29,26 * 10^9$	5,68
256	100	$34,23 * 10^9$	4,88
384	100	$60,55 * 10^9$	2,74
512	100	$100,38 * 10^9$	1,66

14 8-bit processors

The following estimate has been obtain on an 8-bit simulator of a smart-card, using the compiler IAR / AVR. (AVR is the standard 8 bits atmel)

Message digest (bits)	Message Size (**bits**)	Number of cycles	Speed (KB/s)
224	800	535585	3,82
256	768	612097	3,21

Part III

Known Answer Tests and Monte Carlo Tests

We reproduce here some results of CRUNCH with a digest size equal to 256 bits. For the complete results, see the appropriate file.

```
Len = 5
Msg = 48
MD = D609F9048F694543DD90603ABD8A4D590E2E77FB953E90FCC43FE381F9CE3D28

Len = 6
Msg = 50
MD = 04BB153937A7AEAA2060C3FFC2E545C357FBE3DFB60CD14B9A053717F6FD7441

Len = 7
Msg = 98
MD = B152F5410BFA280431D520806C1D6443A3ACD0D49D3E314257AA2DCDC1B74EE5

Len = 8
Msg = CC
```

Algorithm	Message Size (bits)	Block Size (bits)	Word Size (bits)	Message Digest Size (bits)	Security [a] (bits)
CRUNCH-224	$< 2^{64}$	1024	32	224	112
CRUNCH-256	$< 2^{64}$	1024	32 or 64	256	128
CRUNCH-384	$< 2^{128}$	1024	64	384	192
CRUNCH-512	$< 2^{128}$	1024	64	512	256

---

[a] In this context, "security" refers to the fact that a birthday attack [HAC] on a message of size n produces a collision with a factor of approximately $2^{n/2}$.

Figure 5: Secure Hash Algorithm Properties.

```
MD = A819196D71E8CDFABEA307A61A59302DD3FB71FCE0E0D84B0BF656E8FA36D180

Repeat = 16777216
Text = abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmno
MD = 6521EDFAD4166903A03239D021DFC77CA5CBB44D4AA45D90CDD336B91CF17C82
```

Part IV

Expected Strength

The expected strength of the CRUNCH algorithm is summarized in Figure 5.

Part V

Analysis of Known Attacks

15 Background

The design of CRUNCH is based on the XOR of two (fixed) permutations.

The idea of using a block cipher goes back to Preneel, Govaerts and Vandewalle [10] and further analyzed by Black, Rogaway and Shrimpton [2] who proved that among 64 possible constructions, 20 of them are collision-resistant up to the birthday bound in the black-box model.

However, in all these constructions, the key is changed every round, which is usually a serious drawback as concerns efficiency. Hence the idea of building a hash function with block ciphers whose keys are fixed.

The possibility of designing a secure hash function whose underlying compression function uses exactly one call to a (fixed key) block cipher was studied by Black, Cochran and Shrimpton [1]. They essentially proved that such a construction cannot reach a *proven* level of security, by exhibiting a collision attack with $\mathcal{O}(n)$ oracle accesses to the block cipher (modelized as an ideal cipher). Even if this attack is not practical (it requires building a tree with $\Omega(2^n)$ nodes, where n is the bit size of the blocks), it shows that it is not possible to obtain a proof of security against adversaries with unlimited computational abilities.

As an example, we could consider a variant where, instead of Xoring two permutation, the compression function with one encryption permutation G_β is defined by

$$\mathcal{C}_\beta(I) = (G_\beta(I) \oplus I)_{0\ldots(\beta-1)}$$

The obtained scheme will be approximately twice faster than CRUNCH, but cannot be proven secure.

As a consequence, hash functions using with a compression function using two calls to (fixed key) block ciphers are worth considering.

16 Collision attacks

Rogaway and Steinberger [12] investigated the case of hash functions whose compression function uses two calls to fixed key block ciphers. They describe a generic attack for collision finding, which gives an upper bound for the security. More precisely, the best known attack requires $\mathcal{O}(2^{n/2})$ oracle accesses to the permutations and a time complexity $\mathcal{O}(n.2^{n/2})$. This means that – for the best constructions based on 2 permutations – one cannot have a security, against collisions, better than $\mathcal{O}(2^{n/2})$. Note also that [5] gives an attack in $\mathcal{O}(2^{3n/8})$ oracle accesses and time complexity $\mathcal{O}(2^{3n/8})$.

For CRUNCH-β, we have $n = 1024$, and the best known attack is the birthday attack, whose complexity is in $\mathcal{O}(2^{\beta/2})$.

Moreover, for similar constructions (see [5]), Fouque, Stern and Zimmer proved that finding a collision on the compression function (and thus for the whole hash function) requires $\Omega(2^{n/4})$ oracle accesses to the permutations.

17 Preimage attacks

In [12], Rogaway and Steinberger also investigated preimage attacks for hash functions based on a compression function using two calls to (fixed key) block ciphers. They describe a generic attack for preimage, which gives an upper bound for the security. More precisely, the best known attack requires $\mathcal{O}(2^{n/2})$ oracle accesses to the permutations (and a time complexity $> 2^n$). This means that – for the best constructions based on 2 permutations – one cannot have a security, against preimage, better than $\mathcal{O}(2^{n/2})$. Note also that [5] gives an attack in $\mathcal{O}(2^{3n/4})$ oracle accesses, time complexity $\mathcal{O}(n.2^{3n/4})$ and space $\mathcal{O}(2^{3n/4})$.

For CRUNCH-β, we have $n = 1024$, and the best known attack has a complexity $\mathcal{O}(2^\beta)$.

Moreover, for similar constructions (see [5]), Fouque, Stern and Zimmer proved that finding a preimage for hash function requires $\Omega(2^{n/2})$ oracle accesses to the permutations.

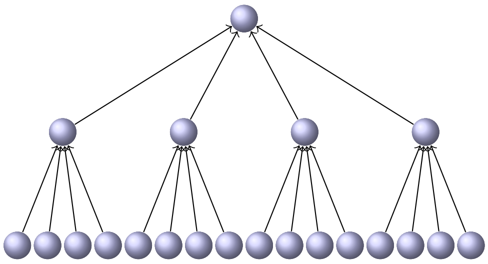
It should be noted that we carefully studied preimage attacks for many variants of CRUNCH. In particular, some variants lead to very efficient hash functions, but unfortunately are trivially broken. For instance if, instead of Xoring permutation, only one encryption permutation G_β is considered and if the compression function is defined by dividing $G_\beta(I)$ into two parts and then Xoring them, the following attack will be possible. Given a digest H, it is possible to obtain I so that $C_\beta(I) = H$. Since the internal function are public, it is enough to go backward from the element $[S, S \oplus H] \in \{0,1\}^{1024}$ where S is any element of $\{0,1\}^{512}$.

Part VI
Advantage and limitations

18 Parallelization

Our algorithm can greatly benefit from the new coming multicore organization which is becoming a de facto standard on most of the general purpose microprocessors.

First of all, the way the message is compressed can be easily changed: for example, the message can be compressed level by level, as shown on the figure below (ratio 4 to 1) :



This is a real advantage when the message to be hashed is very long. The computational time becomes then proportional to the logarithm of the size of the message.

Second, the two permutation functions can be evaluated in parallel, since their computations are independent. This gives an extra performance gain of 2.

19 Vectorization

Our CRUNCH algorithms lends itself very well to "vectorization": 128 bits long XOR operations can be easily used for performing the operations of the innermost loop. Such operations are available on a large number of modern general puropose microprocessors (SSE, Altivec etc.). Future extensions of these instruction to 256 or 512 bit long operations can also be easily used by our algorithm. Tests performed using a state of the art compiler such as ICC (Intel C Compiler) V10 has shown that the innermost loop can be easily fully vectorized and optimized using the full set of registers available (without having to hand code in assembly language to get peak performance).

20 8-bit processors

We can easily implement the algorithm in a 8-bit processor machine, for there are no complicated operation. The figure 1 show that it works well in a 8-bit environment. Nevertheless there is a difficulty to store all the S-boxes (1 MB). Such a difficulty could be avoided if we implement a little algorithm to generate only the values of the S-boxes that we need (on the fly). There, we need a way to generate efficiently and exactly the 35 first decimals of the sinus of an integer between 1 and 262173. Of course, we can imagine other ways to generate these boxes. For example, as the AES is often implemented on smart cards, we could replace the S-boxes by a random number generated by the AES. For this purpose, the 128 bits of $AES(i)$ could be used to generate 4 constants. This surely will slow the process, although this also a way to increase the security as 1024 different S-Boxes instead of only 32 can be generated in the same way .

21 Other Digest Size

Any other size of digest message smaller than 512 bits. If we want a digest size bigger than 512 then we have to change the size of the message block, which has to be at least twice the digest size. For security, we recommend a number of rounds equal to the maximum between the digest size and 224.

Part VII

Variants

There are several possible variants for the CRUNCH hash function.

22 Variants on the S-Boxes

- To implement the CRUNCH hash function on smart cards, the AES block cipher (instead of the sine function) can be chosen to generate the constants needed for the internal functions.
- Another variant is to have true random internal functions.
- In order to construct the encryption permutation, it is also possible to have 32 instead of 16 internal functions. This choice of 16 functions is due to the L2 cache memory.

23 Variants on the design

A variant is to consider only one encryption permutation G and to compute $G(I) \oplus I$ instead of taking the Xor of two permutations as explained in Part V. However we do not recommend this variant.

24 Variants on the encryption permutations

In the design of the encryption permutations, it is also possible to choose other group laws (e.g. addition modulo 2^{1024}) and to have different laws for each encryption permutation.

25 Variants on the Merkle-Damgård construction

As explained in Section 9, the CRUNCH algorithm is parallelizable.

26 The Future

With more memory space, instead of having internal functions from 8 bits to 1024 bits, it could be possible to choose internal function from 16 bits to 1024 bits with an analogous structure (unbalanced Feistel schemes with expanding functions).

Part VIII

Conclusion

The proposed hash algorithm (CRUNCH) has an extremely simple structure: basically the innermost loop amounts to accessing S-boxes and XORing the data accessed. Its simplicity is key to our design because it allows simple and efficient implementation on almost any microprocessor, it simplifies its protection and finally it makes easier to establish a direct relation between CRUNCH security and a generic (well known) security problem. The simplicity of its computational structure is compensated by the requirement of accessing (and storing) S-boxes whose total size is around 1 MB. This storage requirement can be lifted by computing on the fly the S-boxes. Although it increases the computational requirements, it does not alter any properties on the security of CRUNCH.

References

[1] J. Black, M. Cochran, and T. Shrimpton. On the Impossibility of Highly-Efficient Blockcipher-Based Hash Functions. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 526–541. Springer-Verlag, 2005.

[2] J. Black, P. Rogaway, and T. Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from PGV. In M. Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 320–335. Springer-Verlag, 2002.

[3] J.S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In V. Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer-Verlag, 2005.

[4] J.S. Coron, J. Patarin, and Y. Seurin. The Random Oracle and the Ideal Cipher are Equivalent. In D. Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2008.

[5] P.A. Fouque, J. Stern, and S. Zimmer. Two-Permutation Collision-Resistant Hash Function. In R. Avanzi, editor, *SAC 2008*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

[6] U. Maurer, R. Renner, and C. Holenstein. Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In M. Naor, editor, *TCC 2004*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39. Springer-Verlag, 2004.

[7] J. Patarin. A Proof of Security in $O(2^n)$ for the Xor of Two Random Permutations. In R. Safavi, editor, *ICITS '2008*, volume 5155 of *Lecture Notes in Computer Science*, pages 232–248. Springer-Verlag, 2008.

[8] J. Patarin. Generic Attacks for the Xor of k Random Permutations. eprint, 2008.

[9] J. Patarin, V. Nachef, and C. Berbain. Generic Attacks on Unbalanced Feistel Schemes with Expanding Functions. In K. Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 325–341. Springer-Verlag, 2007.

[10] B. Preneel, R. Govaerts, and J. Vandewalle. Hash functions based on block ciphers: A synthetic approach. In D. Stinson, editor, *CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer-Verlag, 1994.

[11] P. Rogaway and J. Steinberger. Constructing cryptographic hash functions from fixed-key blockciphers. In D. Wagner, editor, *CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 433–450. Springer-Verlag, 2008.

[12] P. Rogaway and J. Steinberger. Security/Efficiency Tradeoffs for Permutation-Based Hashing. In N. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 220–236. Springer-Verlag, 2008.

[13] T. Shrimpton and M. Stam. Building a Collision-Resistant COmpression Function from Non-Compressing Primitives. In I. Damgård, editor, *ICALP 2008*, volume 5126 of *Lecture Notes in Computer Science*, pages 643–654. Springer-Verlag, 2008.

[14] S.S. Thomsen. *Cryptographic Hash Functions.* PhD thesis, Technical University of Denmark, N. Smart.

[15] S. Zimmer. *Mécanismes cryptographiques pour la génération de clefs et l'authentification.* PhD thesis, École Polytechnique, N. Smart.