HOW TO FIND AND AVOID COLLISIONS FOR THE KNAPSACK HASH FUNCTION

Jacques PATARIN
Bull CP8, 68 route de Versailles - B.P.45 - 78430 Louveciennes - France

Abstract

Ivan Damgård [4] suggested at Crypto'89 concrete examples of hash functions including, among others, a knapsack scheme. In [3], P. Camion and myself have shown how to break this scheme with a number of computations in the region of 2^{32} and about 128 Gigabytes of memory. More precisely in [3] we showed how to find an x such that $h(x) = b$, for a fixed and average b. (1).

But in order to show that h is not collision free, we have just to find x and y, $x \neq y$ such that $h(x) = h(y)$. (2). This is a weaker condition than (1).

We will see in this paper how to find (2) with a number in the region of 2^{24} computations and about 512 Megabytes of memory. That is to say with about 256 times less computation and memory than [3]. Moreover, ways to extend our algorithm to other knapsacks than that (256, 128) suggested by Damgård are investigated.

Then we will see that for solving problems like (1) or (2) for various knapsacks it is also possible to use less memory if we are allowed to use a little more computing time. This is a usefull remark since the memory needed was the main problem of the algorithms of [3].

Finally, at the end of this paper, we will briefly study some ideas on how to avoid all these attacks by slightly modifying the knapsack Hash functions. However some different attacks could appear, and it is not so easy to find a colision free Hash function, both very quick and with very simple Mathematic expression.

The Proposed Knapsack

Let $a_1, \dots, a_s$ be fixed integers of A binary digits, randomly selected. If T is a plaintext of s binary symbols, $T = x_1 \dots x_s$, then $h(x) = \sum_{i=1}^{s} x_i a_i$ will be the proposed hashed value. In paragraph 1 and 2, values assigned are 256 for s and 120 for A, as suggested in [4]. Thus $h(x)$ has at most 120+8=128 binary digits.

1 The general scheme of our modified algorithm

Our algorithm for finding x and y such that $h(x) = h(y)$ will be mainly a variation of the algorithm described in [3] in order to find x such that $h(x) = b$, where b is a fixed and average value. But our modified algorithm will be in $0(2^{24})$ computations instead of $0(2^{32})$, and it will need about 512 Megabytes of memory instead of about 128 Gigabytes. Nowadays it is quite common to have 512 Megabytes but still quite unusual to have 128 Gigabytes of Memory. So our modified algorithm will appear more practical. Our modified algorithm will proceed in 16 steps, plus a step 0 at the beginning.

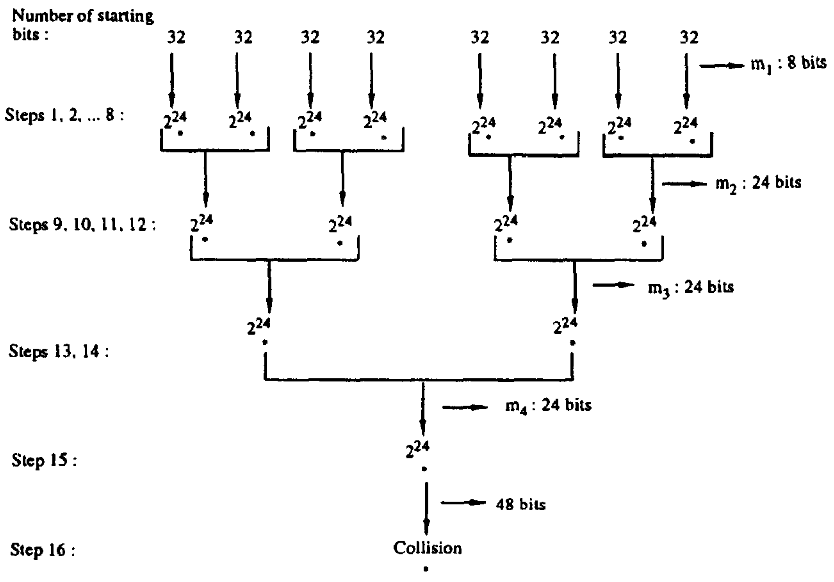
Step 0 : We choose integers m, m_1, m_2, m_3, m_4 and b such that :

a) $m = m_1 m_2 m_3 m_4 > 2^{80}$.

b) The m_i are pairwise coprime, and
• $m_1 \simeq 2^8$, • $m_2 \simeq 2^{24}$, • $m_3 \simeq 2^{24}$, • $m_4 \simeq 2^{24}$.
(So $m \simeq 2^{8+24+24+24} = 2^{80}$.).

c) Let b be a fixed integer, $b \simeq 2^{126}$, for example.
And $\forall i, 1 \le i \le 4$, we define $b_i = b \bmod m_i$.

In order to have a general view of our algorithm, the diagram given below shows the sequence of operations that will be carried out. Let us outline the meaning of the diagram before going into detail. Each black point represents a step of the algorithm. The number 2^{24} associated with the black point represents the evaluation of the number of partial solution that will be found for this step. Each step will study binary sequences. The length of those sequences is 32 for steps 1, 2, 3, 4, 5, 6, 7, 8. It is 64 for steps 9, 10, 11, 12. It is 128 for steps 13, 14. And then step 15 will produce about 2^{24} sequences of length 256 among which in step 16 we will find a collision with probability close to 1.



We will now go through each step in detail.

Step 1 : Let $b_1 = b \bmod m_1$.

We find all sequences $(x_i), 1 \le i \le 32, x_i = 0$ or 1, such that $\sum_{i=1}^{32} x_i a_i \equiv b_1 [m_1]$.
We will find about 2^{24} such sequences because there are 2^{32} sequences (x_i) of 32 bits, and m_1 is close to 2^8. In fact, we will see in Section 2 the number of solutions that we may expect to obtain when the algorithm is brought to completion.
It is important to notice that it is possible to do this step 1 with a number in the region of 2^{24} operations, with a memory of about 2^{24} words of 32 bits. Indeed, we just have to

do the following :

a) Compute and store all values of $b_1 - \sum_{i=1}^{16} x_i a_i$ modulo m_1.

The "store" is done such that we will have easy access to all the sequences (x_i) such that $b_1 - \sum_{i=1}^{16} x_i a_i$ has a given value modulo m_1.

b) Compute, one by one, all the values of $\sum_{i=17}^{32} x_i a_i$ modulo m_1 and look if there are some sequences $(x_i), 1 \leq i \leq 16$, which gived the same value modulo m_1 in a). If it is so, keep all the pairs of sequences obtained $(x_i), 1 \leq i \leq 32$.

For a) we will need about one field of 2^{16} words of 16 bits. And for b) we will need about one file of 2^{24} words of 32 bits.

Step k, k=2 to 8 : In the same way, we find about 2^{24} sequences (x_i) such that $\sum_{i=32(k-1)+1}^{32k} x_i a_i \equiv 0[m_1]$.

Step 9 : We denote $\sum_{i=1}^{32} x_i a_i$ by s_1 and $\sum_{i=33}^{64} x_i a_i$ by s_2.

From the sequences (x_i) found at Steps 1 and 2, we find about 2^{24} sequences $(x_i), 1 \leq i \leq 64$ such that $s_1 + s_2 \equiv b_2[m_2]$. (where $b_2 = b \bmod m_2$). For there are about $2^{24} \times 2^{24} = 2^{48}$ sequences $(x_i), 1 \leq i \leq 64$ such that $(x_1, \ldots, x_{32})$ is a solution from Step 1 and $(x_{33}, \ldots, x_{64})$ is a solution from Step 2. So if the numbers $s_1 + s_2$ are about equally distributed modulo m_2, $m_2 \simeq 2^{24}$, we find about $\frac{2^{48}}{2^{24}} = 2^{24}$ among those sequences such that $s_1 + s_2 \equiv b_2[m_2]$. All sequences (x_i) to be found in Step 9 also have the following property :

$$s_1 + s_2 \equiv b_2[m_2] \quad \text{and} \quad s_1 + s_2 \equiv b_1[m_1].$$

This is because $s_1 \equiv b_1[m_1]$ and $s_2 \equiv 0[m_1]$. It is important to notice that it is possible to do this step 9 with a number in the region of 2^{24} operations, with a memory of about 2^{24} words of 64 bits. Indeed, we just have to do the following :

a) Compute and store all values of $b_2 - s_1$ modulo m_2, where s_1 has been found in step 1.

b) Compute and store all the values of s_2 modulo m_2, where s_2 has been found in step 2.

c) keep all pairs of sequences (x_i) which give the same value modulo m_2 in a) and b).

Step k, k = 10, 11 and 12 : The same way as step 9, we find about 2^{24} sequences (x_i) such that $\sum_{i=64(k-9)+1}^{64(k-8)} x_i a_i \equiv 0[m_i], i = 1, 2$.

Step 13 : Combining solutions of steps 9 and 10, we find about 2^{24} sequences (x_i) such that $\sum_{i=1}^{128} x_i a_i \equiv b_i[m_i], i = 1, 2, 3$. (This is done with about 2^{24} computations, the same way as step 9).

Step 14 : Combining solutions of steps 11 and 12, we find about 2^{24} sequences (x_i) such that $\sum_{i=129}^{256} x_i a_i \equiv 0[m_i], i = 1, 2, 3$.

Step 15 : Combining solutions of steps 13 and 14, we find about 2^{24} sequences (x_i) such that

$$\sum_{i=1}^{256} x_i a_i \equiv b_i[m_i], i = 1, 2, 3, 4. \tag{1}$$

The m_i are pairwise coprimes, so (1) means : we have about 2^{24} sequences (x_i) such that

$$\sum_{i=1}^{256} x_i a_i \equiv b[m], \quad \text{where } m > 2^{80}. \tag{2}$$

Step 16 : Among the 2^{24} sequences (x_i) found in Step 15, we will have with a "good" probability a collision, that is to say two sequences (x_i) and (y_i) such that :

$$\sum_{i=1}^{256} x_i a_i = \sum_{i=1}^{256} y_i a_i.$$

This is because all the sequences found in step 15 have the same value modulo m, where $m > 2^{80}$. So the probability that $h(x) = h(y)$, where x and y are found in step 15, $x \neq y$, is $\geq \frac{1}{2^{128-80}} = \frac{1}{2^{48}}$. But we have about $\frac{2^{24}.(2^{24}-1)}{2}$ couples (x, y) where x and y are found in step 15. So it is possible to prove (this is a classical "birthday paradox") that with a "good" probability we will obtain an x and an y such that $h(x) = h(y)$. And the collision will be found in about 2^{24} computations after step 15 : we just have to compute and store all the values $\sum_{i=1}^{256} x_i a_i$ where (x_i) has been found in step 15 (about 2^{24} such (x_i) have been found) and this will give us the collisions.

Differences between our algorithm and the original algorithm of [3]

There are three main differences in the design of the algorithm that we have described and the original algorithm of [3] :

1. The number of solutions after each step is about 2^{24} instead of 2^{32}, in order to require less memory and to do less computations.
2. We have one more stage where steps 1 to 8 are done. And for these steps we use reduction modulo m_1 where $m_1 \simeq 2^8$.
3. At the end we find a collision with a "Birthday Paradox" like attack.

We will now give more details about the "good" probability to find a collision with our algorithm, and the memory required.

2 More details and small improvements of our algorithm

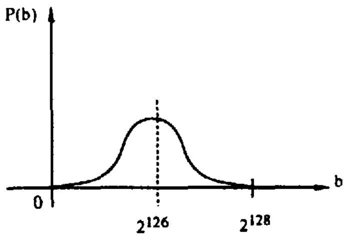
What do we do if at the end of our algorithm no collision is found ? It is possible to use the algorithm again, but with new chosen values. For example we can replace b_1 by $s_1 \equiv b_1 - \lambda[m_1]$ at step 1, and 0 by $s_2 \equiv \lambda[m_1]$ at step 2, where λ is any fixed integer in $[0, m_1 - 1]$. Or we can permute the a_i's. We can also change the value of b or of the m_i's.

But it is much better to keep the same value for b and m : this is because the probability of success in Step 16 depends only on the number of (x_i) found such that $\sum_{i=1}^{256} x_i a_i \equiv b[m]$.

So all the solutions (x_i) found in step 15 with the first application of our algorithm will be useful with the second application of our algorithm. For this second application we can decide to find less solutions in step 15 than 2^{24}, because they will be combined with the "previous" solutions.

"Good" probability

But in fact, even one iteration of our algorithm has a probability of success near 1. This is because $h(x)$ is not equidistributed. If we denote by $P(b)$ the number of (x_i) such that $h(x) = b$, the function $P(b)$ will have a diagram as follows :



And it is possible to prove that, if the a_i's are random numbers of 120 bits, for about 99 % of the (x_i) we will have : $108.2^{119} \leq h(x) \leq 148.2^{119}$.
So for about 99 % of the (x_i), $h(x)$ will have less than 40.2^{119} values, thus less than 2^{125}. So the "collision" in step 16 of our algorithm is easier than expected, and then it is possible to prove that the probability of finding a collision after step 16 is near 1.

Memory

In order to use less memory, it is useful to begin with steps 1, 2 and 9, then steps 3, 4, 10 and 13. Then steps 5, 6 and 11, then steps 7, 8, 12 and 14. Then steps 15 and 16. Thus with a file of about 2^{24} words of 256 bits it will be possible to do all these steps. This is 512 Megabytes of memory. It is high but 256 less than what was needed in [3]. (In [3] it is explain that 64 Gigabytes are needed for one basic step. But it seams that at least about 128 Gigabytes are needed for all the steps). In paragraph 4 we will see some other ideas in order to use less memory but at the cost of a little more computations.

3 Generalization of the new algorithm for other sizes of Knapsacks

Values for complexity 2^{32}

Let $a_1, \dots, a_s$ be fixed integers of A binary digits.

If x is a plaintext of s binary symbols, i.e. $x = x_1 \dots x_s$, then $h(x) = \sum_{i=1}^{s} x_i a_i$ is the proposed hashed value. The hash value $h(x)$ has less than B binary digits, i.e. $B \simeq A + \log_2 s$.
In [3] some algorithms were given to find an x such that $h(x) = b$ (where b is a fixed and random value in $[1, 2^B]$). In complexity $0(2^{32})$ (in time and memory) these algorithms can find such x in the following cases :

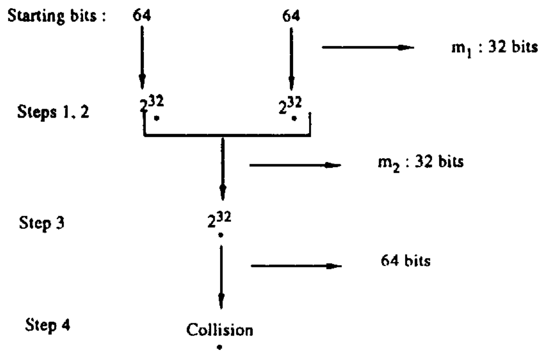
Value of s (or more)	Value of B (or less)
128	96
256	128
512	160
1024	192
2048	224
	etc.

In paragraph 1, we have seen how to find a collision $h(x) = h(y)$ for $s = 256$ and $B = 128$ in $0(2^{24})$ complexity. By using the same ideas, we will now see that, in $0(2^{32})$ complexity our algorithms will find a collision $h(x) = h(y)$ in the following cases :

Value of s (or more)	Value of B (or less)
128	128
256	160
512	192
1024	224
2048	256
	etc.

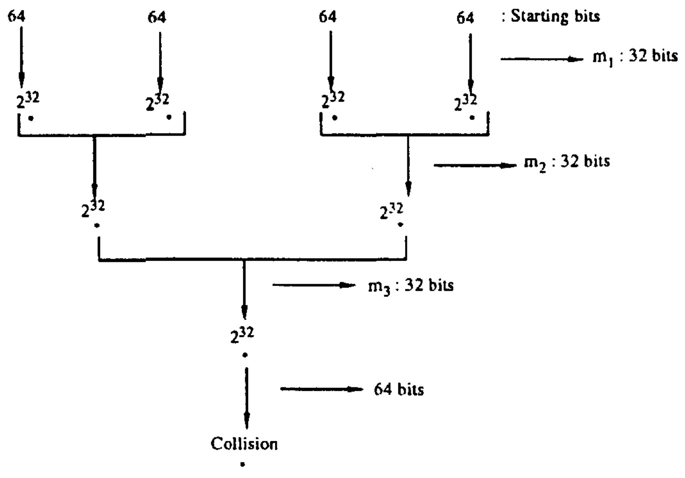
Example 1 : let $s = 128$ and $B = 128$

In this case the function h is not a real "hash" function since from integers of 128 bits, it gives integers of about the same size. But the function is not collision free, even in this case. In the diagram below we will see how to find an x and y such that $h(x) = h(y)$ in $0(2^{32})$ complexity.
We will not give details because this algorithm is similar with algorithm of paragraph 1.



Example 2 : let $s = 256$ and $B = 160$

We will not give the details because it is just the same algorithm but with one more stage. The diagram is



With the same technique, by using two, three, etc. more stages and still about 2^{32} operations we can solve the cases $s = 512$ and $B = 192$, or $s = 1024$ and $B = 224$, etc. And we obtained in this way the values given in this section before example 1. If we compare these values with the values given in [3], we see that, when s is given, we can obtain a collision on 32 extra bits. Or, when B is given, we can obtain a collision with a length of the Text which is twice as small.

Note. It is not a surprise that to find a collision we need a length of the text twice as small. If $x = y.z$ (that is to say if x is the concatenation of y and z) then
$h(x) = b \quad (1) \Leftrightarrow h(y) = b - h(z) \quad (2)$.
And (2) is similar to the problem of finding collision on texts y and z twice as small as x.

A general formula

With the same technique by using e stages, with at most about 2^m memories and in 2^m operations, and with eventually one extra-stage with a reduction modulo c bits (all the others stages perform reductions modulo m bits), we will obtain these general properties :

• Number of starting bits :	• $2^e(m+c)$	bits (or more).
• It invert h on :	• $(1+e)m+c$	bits (or less).
• It find collisions on :	• $(2+e)m+c$	bits (or less).

With $0(2^m)$ time and at most $0(2^m)$ memory.

Example a. With $m = 24, c = 8, e = 3$, we can find a collision for the (256,128) knapsack in $0(2^{24})$ complexity. This is exactly what we did in paragraph 1.

Example b. With $m = 32, c = 0, e = 3$, we can find a collision for the (256, 160) knapsack in $0(2^{32})$ complexity. This is what we did in example 2 above.

4 Time-Memory trade off (for 3 or 4 stages)

All the algorithms given in [3] and in paragraphs 1, 2, 3 above have been designed in order to minimize the time of computing. But the main practical problem of these algorithms is the memory needed, and not the time needed. This is because to perform about 2^{32} basic operations is faisable with modern computers quite easilly, but to find hundred of Gigabytes of memory is still less easy.

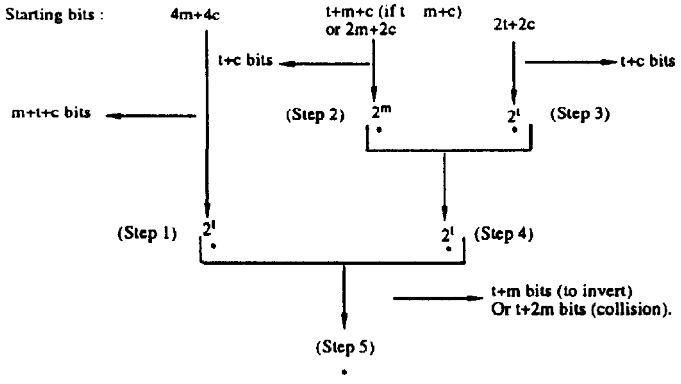
However, we will see in this paragraph that all our algorithms can be modify in order to use less memory, at the cost of a little more computing time. So it will be possible to adapt the algorithm to the memory available.

Due to the lack of space, we will explain how to do this just in the cases with three of four stages, because this is a very good number of stages for a lot of practical knapsacks. However for others knapsacks less or more stages will be better. (For example if the length of the text is appreciably more that the double of the length of the hash value, more stages will be better).

We will denote by t and m two integers such that :

- the memory available is on the order of 2^m.
- the time for computation available is on the order of 2^t.

We will assume that $\frac{t}{2} \leq m \leq t$. And we will denote by c a parameter such that : $t/2 + c/2 \leq m$ and $c \geq 0$. Before going into details, we give the diagram of the steps :
The **general diagram** is :



We now go through each step in more details.

Step 0 : We choose integers m_1, m_2, m_3, m_4 such that :

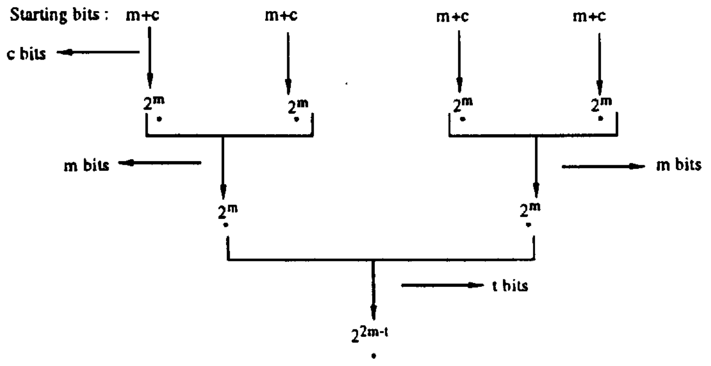
a) m_1, m_2, m_3, m_4 are pairwise coprime, and $m_1 \simeq 2^c$, $m_2 \simeq 2^m$, $m_3 \simeq 2^t$.

b) $m_4 \simeq 2^{t+2m}$ if we want a collision, or $m_4 \simeq 2^{t+m}$ if we want to invert (i.e. to find an x such that $h(x) = b$ where b is given).

Step 1 : The aim of step 1 is to find, and store about 2^m sequences of $4m + 4c$ bits such that $\sum_{i=1}^{4m+4c} x_i a_i \equiv 0[m_1 m_2 m_3]$ in $0(2^t)$ time and $0(2^m)$ memory.

Note. There are about $2^{4m+4c}/2^{t+m+c} \simeq 2^{3m-t+3c}$ such solutions. Among these we want only 2^m such solutions, and this is in fact less or equal because $m \leq 3m - t + 3c$ since $t \leq 2m$ and $c \geq 0$.

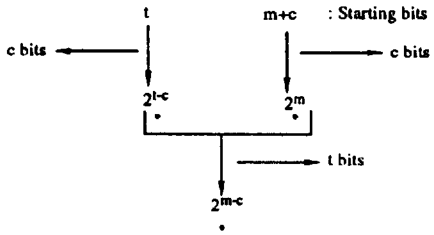
For step 1, we proceed in 2^{t-m} cases. Each of these 2^{t-m} cases follows a diagram like that :



So for each case 2^{2m-t} solutions are found. But all the solution of one case are distincts from the solutions of another case because for example in each case we found solutions such that $\sum_{i=1}^{2m+2c} x_i a_i \equiv \lambda[m_1 m_2]$ where λ is a parameter distinct in each cases. So in this step 1 we will find $2^{t-m} \times 2^{2m-t} = 2^m$ solutions as claimed. And the total time for this step 1 in on the order of $2^{t-m} \times 2^m = 2^t$. (Note that the time for each reduction modulo c bits is about $2^{\frac{m}{2}+\frac{c}{2}} \leq 2^m$ since $c \leq m$ since $c/2 \leq m - \frac{t}{2} \leq \frac{m}{2}$). And the total memory is on the order of 2^m.

Step 2 : The aim of step 2 is to find, and store, about 2^m sequences of $\ell = t + m + c$ bits (if $t \geq m + c$) or of $\ell = 2m + 2c$ bits (if $t \leq m + c$) such that $\sum_{i=\alpha}^{\ell+\alpha} x_i a_i \equiv 0[m_1 m_3]$ in $0(2^t)$ time and $0(2^m)$ memory, where $\alpha = 4m + 4c + 1$.

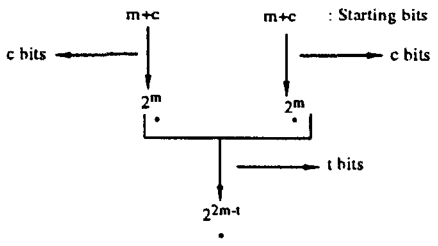
Case 1 : $t \geq m + c$
Then for step 2 we proceed in 2^c cases. Each of these 2^c cases follows a diagram like that :



So we will found about $2^c \times 2^{m-c} \simeq 2^m$ solutions as wanted.

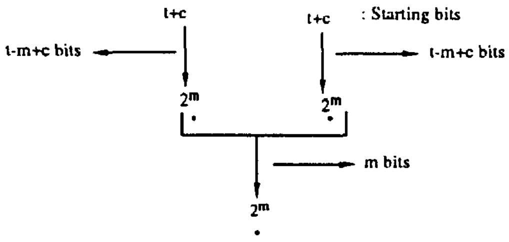
Note. Here we have 2^m solutions for step 2 and we found all of them. The time on the right side was $2^c \times 2^m = 2^{m+c}$ and this is $\leq 2^t$ because here $m + c \leq t$. So for step 2 the total time is in $0(2^t)$. And the value 2^{t-c} in the diagram is a number of sequences that are found one by one, and not stored. So the total memory is in $0(2^m)$.

Case 2 : $t \leq m + c$
Then for step 2 we proceed in 2^{t-m} cases. Each of these 2^{t-m} cases follows a diagram like that :



Here we have about $2^{2m+2c}/2^{t+c} = 2^{2m+c-t}$ solutions such that $\sum_{i=\alpha}^{\alpha+\ell} x_i a_i \equiv 0[m_1 m_3]$ and with the diagram above we find $2^{t-m} \times 2^{2m-t} = 2^m$ such solutions. ($m \leq 2m + c - t$ since $t \leq m + c$ here).

Step 3 : The aim of step 3 is to find, one by one, about 2^t sequences of $2t + 2c$ bits such that $\sum_{i=\beta}^{\beta+2t+2c} x_i a_i \equiv \xi[m_1 m_3]$ in $0(2^t)$ time and $0(2^m)$ memory, where $\beta = \ell + \alpha + 1$ (ℓ and α as in step 2) and where ξ is a fixed and given value. For step 3 we proceed in 2^{t-m} cases. Each of these 2^{t-m} cases follows a diagram like that :



Note. Here, at the begining of this diagram, at least $2^{t/2+c/2}$ computations are done in each one of the 2^{t-m} cases. So we want that $t/2 + c/2 + t - m \leq t$. That is to say : $t/2 + c/2 \leq m$. And this was exactly a property given at the begining for our parameters.

Step 4 and 5 : Each time a solution of step 3 is found, that solution is combined with the solutions of steps 2 and 1, as shown in the general diagram. The property of our algorithm (with 3 or 4 stages) is finally :

Number of starting bits (or more) :	• $3t + 5m + 7c$ if $t \geq m + c$
	• $2t + 6m + 8c$ if $t \leq m + c$
It invert h on	• $2t + 2m + c$ bits (or less).
It find collision on	• $2t + 3m + c$ bits (or less).

With $0(2^t)$ time, $0(2^m)$ memory, and $t \geq m \geq t/2 + c/2$.

Examples.

Example 1. With $m = 32, t = 32, c = 0$ we can invert the (256, 128) knapsack. This was done in [3]. But with $m = 24, t = 39, c = 2$ we can solve the same problem with less memory (and a little more time).

Example 2. With $m = 48, t = 48, c = 16$, or with $m = 42, t = 56, c = 18$, we can find a collision for the (512, 256) knapsack. (However here the algorithm still need a huge amount of time and memory).

5 How to avoid these attacks

In [4], I. Damgård gave a great theorem which shows that if we can have a collision free hash function f from 256 to 128 bits (for example), then we can design a collision free hash function h from any size to 128 bits. And it will be possible to calculate h very quickly if we can calculate f very quickly. The Knapsack Hash function with $s = 256$ and $B = 128$ is quick to calculate. Moreover for example on a 32 bit computer it seems that it will be about 4 times slower than MD4. (MD4 is a concrete example of really used hash function, see [5] for details). Another problem of this Knapsack Hash function is that it requires an array of about 256 words of 120 bits for the numbers $a_1, 1 \le i \le 256$, (this is 3.75 Kbytes), and MD4 doesn't need this. However the main problem, of course, is that we have seen that this Knapsack Hash function is not collision free. But most of the hash functions that are used today (as MD4, MD5 or SHS) do not have a very simple mathematical description. So, is it possible to describe a candidate hash function which will be :

1. Very quick to calculate.
2. Collision free.
3. With a very simple mathematical description.
4. With about 128 bits in output.

In [6], G. Zémor suggested a candidate for points 1, 2, 3 above based on multiplication in the group $G = SL_2(\mathbb{F}_p)$ of 2×2 matrices of determinant 1 over $\mathbb{F}_p$.

However, in order to avoid some potential attacks, G. Zémor suggested to take for p a prime of about 150 bits. So the hash value will be a hash of about 450 bits (instead of 128 for MD4 for example).

It is possible to suggest many different candidates (for example with modular multiplication, but then the function will be much slower than MD4). We will now give some example of functions obtained by modifying just a little the Knapsack Hash function. The hash values of these functions will be 128 bits long. However, in all the examples below our functions (designed in order to avoid the attacks of paragraphs 1-4) will not be collision free, due to other attacks. Nevertheless, we think that, from a theoretical point of view, it is very interesting to study simple mathematical hash functions, in order to gain a better understanding of what makes such a function "collision free" and what doesn't.

Example 1

Let x be a plaintext of 256 binary symbols. Let $h(x) = \sum_{i=1}^{256} x_i a_i$ be the (256,128) Knapsack Hash function that we have studied in paragraph 1. Let $x = (x_i), 1 \le i \le 256, x_i = 0$ or 1, and let $y = (x_i), 1 \le i \le 128$ and $z = (x_i), 129 \le i \le 256$. (So x is the concatenation of y and z). Then let $H(x) = h(x) + y + z$.

Is H collision free ? In fact, we will see that H is not collision free. If $1 \leq i \leq 128$, let $b_i = a_i + 2^{i-1}$. And if $129 \leq i \leq 256$, let $b_i = a_i + 2^{i-129}$. Then $H(x) = \sum_{i=1}^{256} x_i b_i$, so H is just another Knapsack like h. So H is not collision free as explained in [3] and in paragraph 1.

Example 2
Let $x, h(x)$, y and z be as in example 1. We now define an auxiliary function F that takes as input three 128-bit words and produces as output one 128-bit word : $F(X,Y,Z) = YX \vee (\rceil X)Z$.

(In this formula XY denote the bit-wise AND of X and Y, $X \vee Y$ denote the bit-wise OR of X and Y, and $\rceil X$ denote the bit wise complement of X).

Each bit position F acts as a conditional : if x then y else z. (In MD4 a similar function is used, but for 32 bit words). Finally, we define

$$H(x) = F(h(x), y, z).$$

Then H is just a slight variation of the Knapsack. The time needed to calculate H and h is about the same and all the attacks described for the Knapsack seem to be ineffective for H. However, this function is not collision free. In fact if $y = z = b$, then $H(x) = b$. So H is really easy to invert !

Example 3
Let $x, h(x), y, z$ and F be as in example 2.
And let $H(x) = F(h(x), y, z) + h(x)$.
This function H is defined in order to avoid the attacks of paragraphs 1-4 and the attack of example 2. (H computes only additions except one single use of F in order to have a simple mathematical description and be quick to calculate).
However, H is not collision free.
In fact, if $y = 0$ and $z = 1$, then $H(y, z) = 1$.
And more generally, if $y = 0$ then $H(0, z) = h \vee z$.
So by chosen a z with only one zero, the probability that $H(0, z) = 1 = H(0, 1)$ is about 1/2. This property will give easily a collision.
As all our three examples show it can be very dangerous to add an extra composition of functions. It could be a good idea to re-use the input y and z with h in order to design an hash function H to avoid the attacks of paragraph 1-4, but this must be done very carefully.

6 Conclusion

We have seen how to modify the algorithms described in [3] in order to find collisions with less memory, or in order to find collisions for stronger Knapsack. For example, with about 512 Megabytes of memory (instead of about 128 Gigabytes) is it possible to find a collision for the (256,128) Knapsack. Or, in complexity $0(2^{32})$ it is possible to find a collision for the (128,128) Knapsack. The technique is very efficient for various values of the Knapsack. Moreover it is possible to adapt the algorithms by using less memory if a little more computing time is allowed. Finally, we have study some slight modifications of the Knapsack in order to avoid these attacks. Although these modifications did not avoid collisions we think that it is interesting to study simple mathematical candidate hash functions.

Acknoledgements

I want to thank you S. Vaudenay and J.S. Lair for usefull comments.

References

[1] P. Camion, *"Can a Fast Signature Scheme Without Secret Key be Secure ?"*, in AAECC-2, Lecture Notes in Computer Science n° 228, Springer Verlag.

[2] P. Camion and Ph. Godlewski, *"Manipulation and Errors, Localization and Detection"*, Proceedings of Eurocrypt'88, Lecture Notes in Computer Science n° 330, Springer Verlag.

[3] P. Camion and J. Patarin, *"The Knapsack Hash Function proposed at Crypto'89 can be broken"*, Proceedings of Eurocrypt'91, pp. 39-53, Springer Verlag.

[4] I. Damgård, *"Design Principles for Hash Functions"*, Proceedings of Crypto'89, Springer Verlag.

[5] R.L. Rivest, *"The MD4 Message Digest Algorithm"*, Crypto'90, Springer Verlag, pp. 303-311.

[6] G. Zémor, *"Hash Functions and Graphs with Large Girths"*, Proceedings of EUROCRYPT'91, pp 508-511, Springer Verlag.