A Design Principle for Hash Functions

Ivan Bjerre Damgård[1]

Abstract

We show that if there exists a computationally collision free function f from m bits to t bits where $m > t$, then there exists a computationally collision free function h mapping messages of *arbitrary* polynomial lengths to t-bit strings.

Let n be the length of the message. h can be constructed either such that it can be evaluated in time linear in n using 1 processor, or such that it takes time $O(log(n))$ using $O(n)$ processors, counting evaluations of f as one step. Finally, for any constant k and large n, a speedup by a factor of k over the first construction is available using k processors.

Apart from suggesting a generally sound design principle for hash functions, our results give a unified view of several apparently unrelated constructions of hash functions proposed earlier. It also suggests changes to other proposed constructions to make a proof of security potentially easier.

We give three concrete examples of constructions, based on modular squaring, on Wolfram's pseudoranddom bit generator [Wo], and on the knapsack problem.

1 Introduction and Related Work

A hash function h is called *collision free*, if it maps messages of any length to strings of some fixed length, but such that finding x, y with $h(x) = h(y)$ is a hard problem. Note that we are concentrating here on publicly computable hash functions, i.e. functions that are not controlled by a secret key.

The need for such functions to ensure data integrity, and for use with digital signature schemes is well known - see for example [Da], [De], [DP].

Several constructions of hash functions have been proposed, based for example on DES [Wi], [DP] or on RSA [DP], [Gir]. The construction in [Da] was the first for which collision freeness could be proved, assuming security of the atomic operation on which it was based - one-wayness of modular squaring in that case, and more generally, the existence of claw free pairs of permutations. A later example is [Gi]. Unfortunately, this pleasant theoretical property meant a decrease in efficiency compared to other proposals: the time needed for these functions is roughly equivalent to applying RSA to the whole message.

The problem of constructing provably collision free AND fast hash functions is therefore still open.

[1]The author is with Mathematical Institute, Aarhus University, Ny Munkegade, DK 8000 Aarhus C, Denmark.

G. Brassard (Ed.): Advances in Cryptology - CRYPTO '89, LNCS 435, pp. 416-427, 1990.

Many of the difficulties with giving proofs for the known constructions arise from the fact that things seem to get more complex as the length of the messages hashed increases. On the other hand, a hash function is of no use, if we are not allowed to hash messages of arbitrary lengths.

In the present paper, we show that this difficulty can be removed, if the right construction is used: it turns out that ability to cut just 1 bit off the length of a message in a collision free way implies ability to hash messages of arbitrary lengths. The proof suggests a basically sound design principle which can be used as a guideline in designing new hash functions and in revising existing ones.

Our construction is very similar to Merkle's "meta-method", discovered independently [Me]; in comparison, the present construction contains several extra elements that make a formal proof possible without any extra assumptions on the parameters of the functions.

There are also similarities with the methods used in independent work by Naor and Yung [NaYu]. They also prove that fixed size hash functions can be composed to obtain compression of arbitrary polynomial length messages. This is done, both for our type of hash function, and for hash functions with a somewhat weaker property: an enemy is allowed to choose x, and is then given a randomly chosen instance from the hash function family. He can now not in polynomial time find another y such that $f(x) = f(y)$. Naor and Yung construct functions of this type from any one-way permutation, and use them to build digital signature schemes.

From a practical point of view, our construction is more efficient and direct. This is because [NaYu] in order to make their construction work for hash functions with the weaker property, must choose a new independent instance of the fixed length hash function for each message block to process.

Finally, some very recent independent work by Impagliazzo and Naor [ImNa] should be mentioned. They prove that a hash function constructed from the knapsack problem in the same way as in Section 4.3 of this paper is secure in the sense of [NaYu] if the knapsack used induces a one-way function.

2 Preliminaries

We first define the concept of a collision free function family:

Definition 2.1

A *fixed size collision free hash function family* $\mathcal{F}$ is an infinite family of finite sets $\{F_m\}_{m=1}^{\infty}$, and a function $t : \mathbf{N} \to \mathbf{N}$, such that $t(m) < m$ for all $m \in \mathbf{N}$.

A member of F_m is a function $f : \{0,1\}^m \to \{0,1\}^{t(m)}$, and is called *an instance of* $\mathcal{F}$ of size m.

$\mathcal{F}$ must satisfy the following:

1. There is a probabilistic polynomial (in m) time algorithm Θ which, given a value of m, selects an instance of $\mathcal{F}$ of size m at random.

2. For any instance $f \in F_m$ and $x \in \{0,1\}^m$, $f(x)$ can be computed in polynomial time.

3. Given an instance $f \in \mathcal{F}$ selected randomly as in 1), it hard to find $x, y \in \{0,1\}^m$, such that $f(x) = f(y)$ and $x \neq y$.

   More formally: For any probabilistic polynomial (in m) time algorithm Δ, and any polynomial P, consider the subset of instances f of size m for which Δ with probability at least $1/P(m)$ outputs $x \neq y$ such that $f(x) = f(y)$. Let $\epsilon(m)$ be the probability with which Θ selects one of these instances. Then as a function of m, $\epsilon(m)$ vanishes faster than any polynomial fraction□

Condition 1) and 2) say that $\mathcal{F}$ is useful in pratice: instances can be selected, and function values computed efficiently. 3) states the collision free property.

One basic implication of 3) is that functions in $\mathcal{F}$ have a sort of one-way property:

Lemma 2.1

Let $\mathcal{F}$ be a collision free function family, f an instance of size m. Let P_f be the probability distribution on $\{0,1\}^{t(m)}$ generated by selecting x randomly and uniformly in $\{0,1\}^m$ and and outputting $f(x)$.

Then no algorithm inverting f on images selected according to P_f succeeds with probability larger than $1/2 + 1/P(m)$ for any polynomial P

If P_f is the uniform distribution over the image of f or if $m - t$ is $O(m)$, then no inversion algorithm succeeds with probability larger than $1/P(m)$.

Proof

Assume the Lemma is false. Given an algorithm Δ that inverts f with probability at least $1/2 + 1/P(m)$, we select x uniformly and give $f(x)$ as input to Δ. If Δ is successful, we are given a y, such that $f(x) = f(y)$. Let A be the event that the preimage of $f(x)$ has size at least 2. $\{0,1\}^m$ is at least twice as large as the image of f, and therefore A occurs with probability larger than $1/2$. Hence, by assumption on Δ, it succeeds with probability at least $1/P(m)$ when A occurs. Clearly, Δ's choice of which element in the preimage of $f(x)$ to give us is uncorrelated to the choice of x (given $f(x)$). Hence $x \neq y$ with probability at least $1/2$, given that A occurs and that Δ is successful.

For the second statement, note that if P_f is the uniform distribution, then Δ's success is uncorrelated to occurrence of A. And if $m - t = O(m)$, then A occurs with overwhelming probability. In either case, Δ gives us a collision with probability essentially $1/P(m)$ □

Finally, we define the concept of a collision free hash function family:

Definition 2.2

A *collision free hash function family* $\mathcal{H}$ is an infinite family of finite sets $\{H_m\}_{m=1}^{\infty}$, and a polynomially bounded function $t : \mathbf{N} \to \mathbf{N}$.

A member of H_m is a function $h : \{0,1\}^* \rightarrow \{0,1\}^{t(m)}$, and is called *an instance of* $\mathcal{H}$ of size m.

$\mathcal{H}$ must satisfy the following:

1. Given a value of m, there is a probabilistic polynomial (in m) time algorithm Θ which on input m selects an instance of $\mathcal{H}$ of size m at random.

2. For any instance $h \in H_m$ and $x \in \{0,1\}^*$, $h(x)$ is easy to compute, i.e. computable in time polynomial both in m and $|x|$.

3. Given an instance $h \in \mathcal{H}$ selected randomly as in 1), it hard to find $x, y \in \{0,1\}^*$, such that $h(x) = h(y)$ and $x \neq y$.

   More formally: For any probabilistic polynomial time algorithm Δ, and any polynomial P, consider the subset of instances h of size m for which Δ with probability at least $1/P(m)$ outputs $x \neq y$ such that $h(x) = h(y)$. Let $\epsilon(m)$ be the probability with which Θ selects one of these instances. Then as a function of m, $\epsilon(m)$ vanishes faster than any polynomial fraction□

Note that the essential difference between Definition 2.1 and 2.2 is that in 2.2, we do not place any restrictions on the lengths of the inputs to the functions, other than what follows from the obvious fact that polynomial time algorithms cannot hash messages of superpolynomial length.

3 Basic Constructions

The main result in this section is that collision free hash function families can be constructed from fixed size collision free hash function families:

Theorem 3.1

Let $\mathcal{F}$ be a fixed size collision free hash function family mapping m bits to $t(m)$ bits. Then there exists a collision free hash function family $\mathcal{H}$ mapping strings of arbitrary length to $t(m)$-bit strings.

Let h be an instance in $\mathcal{H}$ of size m Then evaluating h on input of length n can be done in at most $n/(m - t(m) + 1) + 1$ steps using 1 processor (we count evaluation of functions in $\mathcal{F}$ as 1 step).

Proof

For each instance $f \in \mathcal{F}$ of size m, we will construct an instance $h \in \mathcal{H}$ of size m. Put $t = t(m)$. For bit strings a, b, we let $a||b$ denote the concatenation of a and b.

The construction will be divided into two cases: first we will discuss the case where $m - t > 1$, and take the $m - t = 1$ case later.

We describe how to compute the value of h on input $x \in \{0,1\}^*$:

Split x in blocks of size $m-t-1$ bits. If the last block is incomplete, it is padded with 0's. Let d be the number of 0's needed. Let the blocks be denoted by $x_1, x_2, ..., x_{n/(m-t-1)}$, where $n = |x|$ (the length after padding).

We append to this sequence one extra $m-t-1$-bit block $x_{n/(m-t-1)+1}$, which contains the binary representation of d, prefixed with an appropriate number of 0's.

Then define a sequence of t bit blocks $h_0, h_1, ...$ by:

$$h_1 = f(0^{t+1}||x_1)$$

$$h_{i+1} = f(h_i||1||x_{i+1})$$

Finally, put $h(x) = h_{n/(m-t)+1}$.

Checking that $\mathcal{H}$ satisfies condition 1 and 2 in Definition 2.2 is easy. For condition 3, assume for contradiction that we are given an algorithm Δ which finds $x \neq x'$ such that $h(x) = h(x')$.

Let h_i, x_i (resp. h'_i, x'_i) be the intermediate results in the computation of $h(x)$ (resp. $h(x')$).

If $|x| \neq |x'|$ *mod* $(m-t)$, then certainly $x_{n/(m-t)+1} \neq x'_{n'/(m-t)+1}$, so that $h(x) = h(x')$ gives us immediately a collision for f. So we may now assume that $|x| = |x'|$ *mod* $(m-t)$ and without loss of generality that $|x'| \geq |x|$.

Consider now the equation

$$h(x) = f(h_{n/(m-t)}||x_{n/(m-t)+1}) = f(h'_{n'/(m-t)}||x'_{n'/(m-t)+1}) = h(x')$$

If $h_{n/(m-t)}||x_{n/(m-t)+1} \neq h'_{n'/(m-t)}||x'_{n'/(m-t)+1}$, we have a collision for f, and are done. If not, we may consider instead the equation

$$f(h_{n/(m-t)-1}||x_{n/(m-t)}) = f(h'_{n'/(m-t)-1}||x'_{n'/(m-t)})$$

and repeat the argument. Clearly this process must stop either by creating a collision for f, or (since $x \neq x'$) by establishing the equation

$$0^{t+1}||x_1 = h'_i||1||x'_{i+1},$$

which is clearly impossible.

Summarizing, we have now a reduction that transforms Δ into an algorithm that finds a collision for f. Suppose Δ takes at most $T(m)$ bit operations. Then x and x' must be of length less than $T(m)$.

Therefore the whole reduction takes time $O(T(m)F(m))$, where $F(m)$ is the time needed to compute 1 f-value, in particular if T is a polynomial, then the whole reduction is in polynomial time. This finally establishes a contradiction with condition 3 in Definition 2.1

Finally, we discuss the case where $m-t = 1$. An easy, but not very efficient solution is prefix-free encode all messages before they are hashed, and then use a construction similar to the above, changing the definition of the h's to:

$$h_1 = f(0^t||x_1)$$

$$h_{i+1} = f(h_i||x_{i+1})$$

Here, of course the x_i's are 1-bit blocks. This can be proved secure in much the same way as above.

If f satisfies the second condition in Lemma 2.1, there is a more efficient solution: we choose a t bit string y_0 uniformly, and define

$$h_1 = f(y_0||x_1)$$

$$h_{i+1} = f(h_i||x_{i+1})$$

this time without doing the prefix free encoding of x. The argument from above will now show that a collison for h will either give us a collision for f or a preimage of y_0. But then we are done by Lemma 2.1. □

Remarks

- The last version of the construction using Lemma 2.1 will not only work when $m - t = 1$, but will work whenever P_f is (close to) the uniform distribution, or $m - t = O(m)$. This should be noted because this version allows hashing of 1 bit more per application of f than the general construction, and is therefore slightly more efficient.

- The trick in the proof above of appending an extra block to the message is only necessary to ensure that we can recognize the difference between messages that need to be padded with d 0's, and messages that simply end with d 0's before padding. In many applications, it is perfectly acceptable that trailing 0's in the last block are ignored, in which case this part of the construction can be skipped.

Let us look at the connection between Theorem 3.1 and previously known hash functions. In [Da], hash functions are constructed based on claw-free pairs of permutations, i.e. pairs of permutations (f_0, f_1) with the same domain D, for which finding $x \neq y$ such that $f_0(x) = f_1(y)$ is a hard problem. We can construct an instance in a collision free function family from the pair (f_0, f_1) by defining a function $f : D \times \{0,1\} \to D$ by:

$$f(x, b) = f_b(x)$$

for $x \in D$ and $b \in \{0,1\}$. Using Theorem 3.1 on the function family thus defined will yield exactly the hashfunctions presented in [Da], except that we have removed the need for the prefix free encoding of messages used there (P_f is uniform in this case).

As a second example, consider one of the first ideas for constructing a hash function from a conventional cipher, due originally to Rabin: Let E be an encryption algorithm that encrypts messages of size t bits, using a key of size k bits. Put $m = t + k$. We split the message x in blocks of size k bits $x_1, x_2, ..., x_n$. We then choose a fixed t bit block h_0 at random and let $h_{i+1} = E_{x_{i+1}}(h_i)$. $h(x)$ is defined to be h_{n+1}.

This fits into the framework of Theorem 3.1 by letting $f(a,b) = E_a(b)$ for a t-bit block b and k-bit block a. Unfortunately, this f is NOT collision free: enciphering any message with an arbitrary key and deciphering with a *different* key will yield a collision for f with high probability. This does not necessarily mean that the function is weak. It does mean, however, that a proof of security cannot be based only on properties of f itself, but must depend on the global structure of the function.

For concrete examples of f, weaknesses have been found, however: if DES is used as E, such that t is only 64, it is well known that the function h constructed as above, permits an enemy which is given x and $h(x)$ to find a $y \neq x$ such that $h(x) = h(y)$, using the "birthday paradox". This is in fact a stronger statement than saying that the hash function is not collision free.

Within ISO, it is currently proposed to standardize a modification of this scheme, where f is defined as $f(a,b) = E_a(b) \oplus b$. For this version of f, there is in fact hope that we can use Theorem 3.1 to prove collision freeness of the entire function by looking only at f: Given c, it is not easy to solve $f(a,b) = c$ for a and b, if E is a strong encryption algorithm, and thus there good reason to believe that this version of f is collision free.

3.1 Parallellizing Hash Functions

Based on Theorem 3.1, we can give alternative constructions that allow computation in parallel of a hashvalue:

Theorem 3.2

Let $\mathcal{F}$ be a collision free function family mapping m bits to $t(m)$ bits. Then there exists a collision free hash function family $\mathcal{H}$ mapping arbitrary strings to $t(m)$-bit strings with the following property:

Let h be an instance in $\mathcal{H}$ of size m. Put $t = t(m)$. Then evaluating h on input of length n can be done in $O(log_2(n/t)t/(m-t))$ steps using $n/2t$ processors (we count evaluation of functions in $\mathcal{F}$ as 1 step).

Proof

We are given an instance $f \in \mathcal{F}$ of size m. By Theorem 3.1, we can construct a hash function h', which maps $2t$ bits to t bits in $t/(m-t)$ steps. Note, that since the length of the input is fixed, we do not need to append an extra input block in the construction of h'.

We then construct an instance $h \in \mathcal{H}$ as follows:

Let a message x of length n be given. We pad x with a number of 0's, such that the resulting bit string x_0 has length equal to $2^j t$ for some j. Now construct a sequence $x_0, x_1, ..., x_j$ by defining x_{i+1} in terms of x_i: split x_i in blocks of length $2t$, apply h' to each block and concatenate the results to obtain x_{i+1}.

The sequence stops at x_j, which has length t. We then hash the binary representation of the length n of x using Theorem 3.1 to obtain a t bit block len_x. Finally we

put

$$h(x) = h'(x_j||len_x).$$

The statements on the time and processors needed to compute h are trivial to verify.

As for collision freeness, suppose we could produce $x \neq x'$ with $h(x) = h(x')$ in expected polynomial time.

If $x'_{j'}||len_{x'} \neq x_j||len_x$, we have a collision for h', contradicting Theorem 3.1. Hence we may also assume that $n = n'$, since otherwise $len_{x'} = len_x$ would imply a collision.

Now, $x \neq x'$ implies that we may choose an i such that $x_i \neq x'_i$, but $x_{i+1} = x'_{i+1}$. This clearly implies a collision for h'□

Finally, it is also easy to see how to make a construction that allows c processors to cooperate in computing a hashvalue, acheiving a speed up by a factor of c for long messages. Loosely speaking, we split the message in c parts of roughly the same size, hash each part in parallel using Theorem 3.1, and finally hash the c output blocks using Theorem 3.1 once again. Formalizing this and proving collision freeness is left to the reader.

4 Concrete Constructions

In the following we propose three concrete constructions of collision free functions with fixed inputsize. These functions can then be turned into hashfunctions by a straightforward application of Theorem 3.1.

4.1 Based on Modular Squaring

We give first a construction based on the hardness of extracting square roots modulo large numbers with two prime factors. The construction bears some similarities with the functions considered in [Gir], but is fundamentally different in that the functions from [Gir] do not allow for application of Theorem 3.1.

Let $n = pq$, where p and q are large primes. Let the length of n be s bits. For concreteness, one can think of $s = 512$. Next, choose I, a proper subset of the numbers $1, 2, ..., s$. For any s-bit string $y = y_1, y_2, ..., y_s$, let $f_I(y)$ be the concatenation of all y_j for which $j \in I$.

Finally, we can define our candidate collision free function f from m bits to t bits by setting $m = s - 8$, $t = |I|$, and defining

$$f(x) = f_I((3F|x)^2 \ mod \ n),$$

where $3F|x$ denotes the concatenation of the byte $3F$ (in hex) and x. This concatenation implies that all inputs to the modular squaring are less than $n/2$, but large enough to guarantee that modular reductions always take place. We therefore prevent trivial collisions implied by $x^2 = (-x)^2 \ mod \ n$, and also attempts to find collisions by choosing for example small 2-powers as input.

We also need to specify choices of I that will make f secure and efficient. The problem of finding a collision for f can be reformulated: find numbers $x \neq y$, such that their squares modulo n match at the positions designated by I. Girault [Gir] shows how to do this if I designates a reasonably small (less than 64) number of the least significant, or the most significant bits.

This suggests that we should choose the positions to be spread evenly over the s possible ones, since Girault's method and related ones [GTV] will then fail. On the other hand there are good practical reasons for not using completely random positions, but at least lumping them together in bytes. Moreover, $|I|$ should not be chosen too small, to prevent "birthday collisions".

To be concrete, if $s = 512$, the above suggests that good choices would be $|I| = 128$, and letting f_I extract every 4'th byte.

This function will hash up to 376 bits pr. modular squaring, which on for example an IBM P/S II model 80 will give a speed of about 100 Kbits/sec. Special purpose hardware will give speeds in the Mbit/sec area.

4.2 Based on Wolfram's Pseudorandom Bit Generator

The second suggestion bases itself on the pseudorandom bit generator proposed by Wolfram [Wo]. In general, a pseudorandom bit generator is an algorithm that takes as input a short, truely random seed, and stretches this into a long, seemingly random output string. It is intuitively clear that such a generator must in some sense implement a one-way function from its seed to its output: if the seed was easy to find given some part of the output, then the whole output could be predicted and hence be recognized as being non-random.

However, this one-way property does NOT in general imply collision freeness of a function constructed directly from the generator - an analysis of the concrete instance is therefore very much required.

Let us therefore have a look at the algorithm suggested by Wolfram: We define a function g from n bit strings to n bit strings: let $x = x_0, x_2, ..., x_{n-1}$, then the i'th bit of $g(x)$ is

$$g(x)_i = x_{i-1} \oplus (x_i \vee x_{i+1}),$$

where addition and subtraction of 1 are modulo n. One can think of this as a register R in which the bits are updated in parallel by setting $R := g(R)$. This is known as a one-dimensional cellular automat.

To use this for pseudorandom bit generation, one does the following:

1. Choose x at random.
2. $x := g(x)$.
3. Output x_0. Go to 2.

In [Wo], this pseudorandom bitgenerator is analyzed, and results of a large number of statistical tests are given. Its security is proved against enemies restricted to certain types of computations, but its ability to foil arbitrary polynomial time enemies

remains a conjecture. All the known evidence, however, indicates that the generator is in fact very strong.

Let the bits produced by the algorithm on input x be denoted by $b_1(x), b_2(x), ...$

The natural way to use this to construct a collision free function is to choose two natural numbers $c < d$, and let a function f_0 be defined by

$$f_0(x) = b_c(x), b_{c+1}(x), ..., b_d(x).$$

There are two possible flaws in this idea, which must be taken care of:

First, a natural demand to a function like this is that all output bits depend on all input bits. It is easy to see that changing 1 bit of x will eventually after many executions of $x := g(x)$ affect all of x - but the effect only propagates slowly through the register: 1 bit to the right, and about .25 bit to the left pr. application of g [Wo]. Hence choosing c too small will clearly be dangerous. A natural minimum value of c would therefore be one that guaranties that all output bits depend on all input bits.

The second problem is that g itself is not collision free: for example, $g(1^n) = g(0^n) = 0^n$. And clearly, $g(x) = g(y)$ implies $f_0(x) = f_0(y)$. More generally, if $g^v(x) = g^v(y)$ for small v, then there is at least a nonnegligible chance that $f_0(x) = f_0(y)$.

One natural way to get rid of this difficulty is to restrict f to a subset of the n-bit strings, thereby lowering the chance that pairs x, y of the above form exist, or at least making them harder to find. One concrete possibility is to restrict to strings of the form $x||z$, where z is a randomly chosen constant string in $\{0,1\}^r$, $r < n$.

Thus we will define our final candidate f so that it maps an $n - r$ bit string x to

$$f(x) = f_0(x||z).$$

The following lemma provides evidence in favor of this approach:

Lemma 5.1

If $g^v(x) = g^v(y) = z$, and $2v$ consecutive bits of x and y are equal, then $x = y$.

Proof

Let j be the index of the position immediately to the right of the $2v$ bits we know to be equal. Each bit of z depends on at most $2v+1$ bits of x(or y). Let l be chosen such that z_l is a function of bits $j, ..., j + 2v$ of x(or y). Now, since inverting x_j will invert z_l, our assumptions imply that $x_j = y_j$. We can now "slide" the same argument one position to the right□

This provides good evidence that choosing a relatively large r will make "trivial" collisions more sparse and harder to find.

As a concrete example, suppose we choose $n = 512$, $r = 256$, $c = 257$ and $d = 384$. The resulting f will map 256-bit strings to 128-bit strings, and thus the hash function constructed from f by Theorem 3.1 will hash messages in blocks of 128 bits, and produce 128 bit outputs.

This function will be extremely well suited for a hardware implementation: in VLSI, we can compute g by updating all bits in parallel, and thus one application of g would only take 1 or 2 clock cycles, *independently* of n. This will produce speeds in the Mbit/sec area even with quite modest clock speeds. Moreover such a hardware implementation would be extremely easy and cheap to build.

4.3 Based on the Knapsack Problem

Although the knapsack problem is NP-complete, and therefore probably very hard in the worst cases, making use of this hardness in cryptography is not easy, as shown by the fate of many public-key systems based on this problem.

The difficulty, however, is largely due to the fact that an encryption function must be invertible, and that the knapsacks used must therefore have some built-in structures, which in many cases turn out to be useful to a cryptanalyst. A hash function, on the other hand, never has to be inverted, and therefore completely randomly generated knapsacks can be used.

The naive way of doing this is to choose at random numbers $a_1, ..., a_s$ in the interval $1..M$, where s is the maximal length of a message to be expected. We can then hash the binary message $m_1, m_2, ..., m_s$ to

$$f(m_1, ..., m_s) = \sum_{i=1}^{s} m_i a_i$$

As shown in [GC], this is completely insecure for large s, more precisely when $M \leq s^{log(s)/4}$. To solve this, we propose to fix s to something reasonably small compared to M, and use Theorem 3.1 to construct the actual hash function. As an example, one could choose s to be about $2log(M)$, which implies that f compresses s bits blocks to $s/2$ bit blocks, and that the condition of [GC] is very far from being satisfied.

Concrete choices could be $s = 256$ and $M = 2^{120} - 1$. This would give an output from the final hash function of length 128 bits. On an IBM P/S II Model 80, this version will run at a speed of about 250 Kbits/sec.

To specify the function one needs to specify about 4 Kbytes of data. On e.g. a PC with 640K of RAM, this does not seem excessive. But in situations with a smaller memory, one can trade time for memory and generate the a's pseudorandomly in stead of remembering all of them.

The result of [ImNa] is a strong indication that this function is indeed collision free, although the security property proved is weaker than what we need here (see Section 1).

Another indication of the strength of the function is the fact that the problem of deciding in general whether a given knapsack induces an injective mapping is co-NP complete. A collision is of course a witness of non- injectiveness (the decision problem is clearly trivial if the knapsack compresses its input, but this does not imply that a witness is easy to find).

We remark that even knapsacks that expand their input slightly (and therefore cannot be used by [ImNa]) can be used to build hash functions secure in the sense

of [NaYu], if one is willing to assume that they induce collision free mappings. This seems reasonable in view of the co-NP completeness of the problem involved and the fact that the decision problem is non trivial in this case. The construction and proof can be obtained by adapting the techniques of [NaYu].

References

[Da] Damgård: "Collision Free Hash Functions and Public Key Signature Schemes", Proceedings of EuroCrypt 87, Springer.

[De] D. Denning: "Digital Signatures with RSA and other Public Key Cryptosystems", CACM, vol.27, 1984, pp.441-448.

[DP] Davis and Price: "The Application of Digital Signatures Based on Public Key Crypto-Systems", Proc. of CompCon 1980, pp.525-530.

[GC] Godlewski and Camion: "Manipulation and Errors, Localization and Detection", Proceedings of EuroCrypt 88, Springer.

[Gi] Gibson: "A Collision Free Hash Function and the Discrete Logarithm Problem for a Composite Modulus", Manuscript, 1/10/88, London, England.

[Gir] Girault: "Hash Functions Using Modulo-n Operations", Proceedings of EuroCrypt 87, Springer.

[GTV] Girault, Toffin and Vallée: "Computation of Approximate L-th Roots Modulo n and Application to Cryptography", Proceedings of Crypto 88, Springer.

[ImNa] Impagliazzo and Naor: "Efficient Cryptographic Schemes Provably as Secure as Subset Sum", Proc. of FOCS 89.

[Me] Merkle: "One Way Hash Functions and DES", these proceedings.

[NaYu] Naor and Yung: "Universal One-Way Hash Functions", Proc. of STOC 89.

[Wi] Winternitz: "Producing a one-way Hash Function from DES", Proceedings of Crypto 83, Springer.

[Wo] Wolfram: "Random Sequence Generation by Cellular Automata", Adv. Appl. Math., vol 7, 123-169, 1986.