Domain Extender for Collision Resistant Hash Functions Using a Directed Acyclic Graph

Palash Sarkar
Cryptology Research Group
Applied Statistics Unit
Indian Statistical Institute
203, B.T. Road
Kolkata 700108
West Bengal, India
palash@isical.ac.in

Abstract

We study the problem of securely extending the domain of a collision resistant compression function. Our first contribution is to show that given an arbitrary directed acyclic graph and a collision resistant compression function, it is possible to construct a collision resistant hash function. Next we introduce a new technique for constructing a hash function which can handle arbitrary length strings. The amount of padding and the number of invocations of the compression function required by our algorithm is asymptotically smaller compared to the Merkle-Damgård algorithm. Our third contribution is to provide some concrete examples and hence derive the foundation for the design of a secure parallel hash algorithm.

Keywords : hash function, compression function, composition principle, collision resistance, preimage resistance, parallel algorithm, directed acyclic graph.

1 Introduction

Hash functions are a basic cryptographic primitive and are used extensively in digital signature protocols. For such applications a hash function must satisfy certain necessary properties, two of which are collision resistance and pre-image resistance. Collision resistance implies that it should be computationally intractable to find two elements in the domain which are mapped to the same element in the range. On the other hand, pre-image resistance means that given an element of the range, it should be computationally intractable to find its pre-image.

Construction of collision resistant and pre-image resistant hash functions are of both practical and theoretical interest. It is possible to construct a hash function such that obtaining a collision implies the ability to solve the discrete logarithm problem in a cryptographically hard group [6]. This solves the problem in a theoretical sense. However, in practice such a hash function would be unacceptably slow.

The current trend in hash function design is to either design them from "scratch" or to design them from other primitives like block ciphers. A recent example of the second technique is the

analysis presented in [3]. In this paper, we concentrate on the theory behind the first mentioned method of constructing hash functions. The advantage of designing a hash function from scratch is that one can use simple logical/arithmetic operations to design the algorithm and hence achieve very high speeds. The disadvantage is that we obtain no proof of collision resistance. Hence a user has to *assume* that the function is collision resistant. A well accepted intuition in this area is that it is more plausible to assume a function to be collision resistant when the domain is fixed (and small) rather than when it is infinite (or very large). A fixed domain function which is assumed to be collision resistant is often called a *compression function*.

For practical use, it is required to hash messages of arbitrary lengths. Hence one must look for methods which extend the domain of a compression function in a "secure" manner, i.e., the extended domain hash function is collision resistant provided the compression function is collision resistant. Any method which achieves this is often called a *composition priniciple*. Early works by Merkle [8] and Damgård [7] provide a secure composition principle which is called the Merkle-Damgård principle. In a recent work, Sarkar and Schellenberg [12] provide a binary tree based parallelizable secure composition principle.

In the following we write $|x|$ for the length of a string and $x_1||x_2$ for the concatenation of two strings x_1 and x_2. Also x^r denotes the reverse of the string x. By an (n, m) function we will mean a function which maps $\{0,1\}^n$ to $\{0,1\}^m$. Also all logarithms in the paper are to the base two.

It is almost impossible to define the collision resistance of a single hash function. Instead we have to consider an infinite family of hash functions and define collision resistance by requiring the success probability of any probabilistic polynomial time adversary to be a negligible function. See [10, 7] for a precise definition of this notion. However, from a practical point of view we need to study the collision resistance of a single function. As mentioned above, the approach is to *assume* a compression function to be collision resistant and then look for a secure composition principle to extend the domain.

We consider a very general class of domain extending algorithms. More precisely, let h be a compression function and $\mathcal{C}$ be a class of algorithms. Any algorithm A in the class $\mathcal{C}$ behaves in the following manner. Given a binary string (the message) x, it does the following.

- Possibly pads some 0's to x.
- Invokes the compression function h a finite number of times. The input to each invocation is constructed from zero or more message bits and zero or more bits of the outputs of some previous invocations of h.
- Each bit of the padded message is provided as input to exactly one invocation of h.
- Each bit of any intermediate invocation of h is provided as input to at most one more invocation of h.
- The output of A consists of the concatenation of the entire outputs of one or more invocations of h.

The structure of any such algorithm A can be described using a directed acyclic graph (DAG) with a node for each invocation of h and an arc from node u to node v if some part of the output of the invocation of h at u is provided as input to the invocation of h at v.

The natural question that arises is whether such an algorithm A preserves collision resistance of h. This motivates us to look at domain extending algorithms using an arbitrary DAG. In Section 2, we provide a construction of a secure domain extending algorithm using an arbitrary DAG. The problem of domain extension consists of two separate problems.

- Extend the domain from short fixed length strings to longer fixed length strings.
- Extend the domain to the set of all non-null binary strings.

The first problem is easier and often forms the basis for a solution to the second problem. A solution to the second problem is usually more difficult. In Section 3, we provide a new solution to the second problem based on any infinite family of single output DAGs. The only previously known solution to the second problem is due to Damgård [7] for the special case of DAGs which are dipaths. Moreover, the algorithm in [7] requires $O(|x|)$ amount of padding for obtaining the digest of a message x. On the other hand, our algorithm requires $O((\log^* |x|)(\log |x|))$ amount of padding to obtain the digest of x and hence is asymptotically more efficient than the Merkle-Damgård algorithm. Furthermore, the number of invocations of the compression function required by our algorithm is asymtotically less than that of the Merkle-Damgård algorithm.

Parallelism in the design of hash functions is important from a practical point of view. Previous work in this area have focussed on parallelism in the compression function [4], parallelism in the composition principle [1, 9, 12] and also parallel hash algorithms [11] and other issues related to parallelism [2]. In Section 4, we provide some concrete examples of hashing structures and show that these can be combined nicely to design a parallel hash function. Issues related to the practicability of the parallel design are not discussed. However, we believe that the design leads to a practical algorithm and the details of an actual practical design is left for future work.

2 DAG Hashing

A directed acyclic graph $D = (V, A)$ where V is a finite non empty set of nodes and A is a set of arcs such that D contains no directed cycles. For any node v of D, we will denote by $\Gamma(v)$ (resp. $\Delta(v)$) the set of all arcs coming into (resp. going out of) v. It is well known (and easy to prove) that any DAG contains at least one node of indegree zero and at least one node of outdegree zero. We make the following definition.

Definition 1 *Let $D = (V, A)$ be a DAG. A node with indegree zero will be called an* exposed *node; a node with outdegree zero will be called an* output *node and all other nodes will be called* internal *nodes.*

If v is an exposed node and u is an output node, we have $\Gamma(v) = \Delta(u) = \emptyset$. Given a DAG D, let $l(D)$ be the maximum number of nodes on any path from an exposed node to an output node (counting both the start and the end nodes). We will call $l(D)$ to be the *depth* of D. To each node v of D we assign a non negative integer called its level in the following manner.

1. Let S be the set of all output nodes and set $\mathsf{level}(v) = l(D) - 1$ for all $v \in S$.
2. For $\mathsf{count} = l(D) - 2$ down to 0 do

```
    T = ∅.
    For v ∈ V do
        if ((v ∉ (S ∪ T)) ∧ (Δ(v) \ S = ∅))
        set level(v) = count;
        set T = T ∪ {v}.
    End do.
    S = S ∪ T.
End do.
```

The level numbers of the nodes partition V into l disjoint subsets $S_0, \ldots, S_{l-1}$, where $l = l(D)$ and $S_i = \{v : \mathsf{level}(v) = i\}$. Note that all output nodes are at the same level, but the exposed nodes can be at different levels. However, all nodes at level zero are necessarily exposed nodes.

An *assignment* α on $D = (V, A)$ is a function $\alpha : A \to \mathbb{N}$ which assigns a positive integer to each arc of D. Let n and m be two positive integers with $n > m$ and D be a DAG. An assignment α is said to be *proper* with respect to (n, m, D) if the following condition holds.

$$\text{For any node } v \text{ of } D, \text{ (a) } \sum_{e \in \Delta(v)} \alpha(e) = m \text{ and (b) } \sum_{e \in \Gamma(v)} \alpha(e) \leq n.$$

For any node v, we define the *fan-in* of v to be $\mu(v) = \sum_{e \in \Gamma(v)} \alpha(e)$. Thus for a proper assignment α on (n, m, D) and any node v, we have $\mu(v) \leq n$. For any exposed node v, we have $\mu(v) = 0$.

A *structure* is a tuple $\mathcal{S} = (n, m, D = (V, A), \alpha)$ where α is a proper assignment on (n, m, D). By an exposed or output node of a structure $\mathcal{S}$ we will mean an exposed or output node of the underlying DAG D. Similarly, by the depth of a structure we will mean the depth of the underlying DAG.

2.1 Construction of $h_{\mathcal{S}}$

Given a structure $\mathcal{S}$ and an (n, m) compression function h, we can define a hash function $h_{\mathcal{S}}$ in the following manner. The hash function takes as input a message x (whose length we specify later) and produces as output a digest $y = h(x)$. The basic idea is to invoke the hash function h for each node v of D. The function h takes n bits as input and produces m bits as output. To ensure this we have to parse (or format) the message x properly. We first describe this formatting procedure. For any node v, the input to v will be written as $z(v)$ and the output of v will be written as $y(v)$. The input $z(v)$ is formed by concatenating a part of the message x and some portions of the outputs of previous invocations of h. The substring of the message which is provided as input to v is denoted by $x(v)$. Note that $|x(v)| = n - \mu(v)$. As a notational convenience, we will assume $V = \{v_1, \ldots, v_t\}$ and write $x_i = x(v_i)$, $z_i = z(v_i)$ and $y_i = y(v_i)$.

We associate a non empty string $\beta(e)$ of length at most m to each arc e of D in the following manner. Let $\Delta(v_i) = \{e_{i,1}, \ldots, e_{i,k_i}\}$ and write $y_i = y_{i,1}||\ldots||y_{i,k_i}$, where $|y_{i,j}| = \alpha(e_{i,j})$ for $1 \leq j \leq k_i$. Then $\beta(e_{i,j}) = y_{i,j}$. For any node v_i write $\Gamma(v_i) = \{e_{i,1}, \ldots, e_{i,r_i}\}$. Then the input z_i to v_i is formed by concatenating x_i and $\beta(e_{i,1}), \ldots, \beta(e_{i,r_i})$, i.e., $z_i = x_i||\beta(e_{i,1})||\ldots||\beta(e_{i,r_i})$. For any exposed node v, we have $\Gamma(v) = \emptyset$ and consequently $z(v) = x(v)$ and $|x(v)| = n$. Given a message x, the computation of $h_{\mathcal{S}}(x)$ is described as follows.

Computation of $\mathbf{h}_{\mathcal{S}}(\mathbf{x})$.

1. For $i = 0$ to $l(D) - 1$ do
2. For $v \in S_i$
3. set $y_i = h(z_i)$.
4. End do.
5. End do.
6. $z = \boldsymbol{\lambda}$ (the empty string).
7. For $v \in S_{l(D)-1}$ set $z = z||y(v)$.
8. output z.

We say that the hash function $h_{\mathcal{S}}$ is associated to the structure $\mathcal{S}$ and the compression function h.

Remark : The loop in Steps 2 to 4 involves the invocation of h for each node in S_i. These invocations can be carried out in parallel and hence a parallel execution of the algorithm will require exactly $l(D)$ parallel rounds. Thus the depth of a struture determines the number of parallel rounds required to compute the output of the associated hash function.

2.2 Properties of $h_{\mathcal{S}}$

The following result describes the lengths of the input and output strings of the hash function $h_{\mathcal{S}}$.

Proposition 2 *Let $\mathcal{S} = (n, m, D = (V, A), \alpha)$ be a structure and $h : \{0,1\}^n \to \{0,1\}^m$ be a compression function. Then $h_{\mathcal{S}} : \{0,1\}^N \to \{0,1\}^M$ where $N = t(n-m) + sm$ and $M = sm$, where $t = |V|$ and s is the number of output nodes in D.*

Proof. The outputs of all the output nodes are concatenated and provided as output of $h_{\mathcal{S}}$. The output of each node is m, hence the output of $h_{\mathcal{S}}$ is sm.

The calculation of the input size is as follows. There are t nodes in D. The function h is invoked once for each of these nodes and hence h is invoked a total of t times. Each invocation of h requires an n bit input. Thus a total of tn bits are required as input to all the invocations. An input to an invocation of h either comes directly from the message x or is a part of the intermediate output of some previous invocation of h. There are $(t-s)$ intermediate outputs which provide a total of $(t-s)m$ bits. Hence the message x has to provide a total of exactly $tn - (t-s)m = t(n-m) + sm$ bits. ■

We next prove that the construction described above preserves the property of collision resistance.

Theorem 3 *Let $h_{\mathcal{S}}$ be a hash function constructed from a structure $\mathcal{S} = (n, m, D, \alpha)$ and a compression function h described as above. Then it is possible to find a collision for $h_{\mathcal{S}}$ if and only if it is possible to find a collision for h.*

Proof. If : We have to show that any collision for h can be extended to a collision for $h_{\mathcal{S}}$. Let x_1 and x_1' be n-bit strings which collide for h. Let v be an exposed node of the structure $\mathcal{S}$. We now define two strings x and x' in the domain of $h_{\mathcal{S}}$ such that $x \neq x'$ and $h_{\mathcal{S}}(x) = h_{\mathcal{S}}(x')$. Note that to define x and x' it is enough to define the corresponding inputs $x(u)$ and $x'(u)$ to each node u of $\mathcal{S}$. We do this as follows: Set $x(v) = x_1$, $x'(v) = x_1'$ and for any $u \neq v$, set $x(u)$ and $x'(u)$ both to be equal to an arbitrary binary string of appropriate length. Then it is clear that $x \neq x'$. Moreover,

$h_{\mathcal{S}}(x) = h_{\mathcal{S}}(x')$ since the outputs of the invocation of h at node v are equal and the inputs to all other nodes are equal. Thus x and x' provide a collision for h.

Only If : For $0 \leq i \leq l(D) - 1$, we define three sequences of sets $\mathsf{ZList}_i, \mathsf{XList}_i$ and YList_i, where

$$\mathsf{XList}_i = \{x(v) : \mathsf{level}(v) = i\},\ \mathsf{ZList}_i = \{z(v) : \mathsf{level}(v) = i\} \text{ and } \mathsf{YList}_i = \{y(v) : \mathsf{level}(v) = i\}.$$

Note that the message x can be written as a concatenation (in a particular order) of the strings in XList_i for $0 \leq i \leq l(D) - 1$.

For the proof assume that there are two messages x and x' such that $x \neq x'$ but $h_{\mathcal{S}}(x) = h_{\mathcal{S}}(x')$. We show that it is possible to find a collision for h. In the following we will use primed and unprimed notations to denote quantities corresponding to x' and x respectively.

Our proof technique is the following. Assume that there is no collision for any of the invocations of h. We show that this implies $x = x'$ which contradicts the hypothesis that $x \neq x'$. Hence there must be a collision for some invocation of h. We now turn to the proof of the fact that if there is no collision for h, then $x = x'$. This is proved by backward induction on i. More precisely, we show that if there is no collision for h, then for each i, we have $\mathsf{XList}_i = \mathsf{XList}'_i$. Consequently, $x = x'$. We now turn to the actual proof.

We are given that $h_{\mathcal{S}}(x) = h_{\mathcal{S}}(x')$. This implies that $\mathsf{YList}_{l(D)-1}(x) = \mathsf{YList}'_{l(D)-1}(x')$ and consequently for each $v \in S_{l(D)-1}$ we have $h(z(v)) = y(v) = y'(v) = h(z'(v))$. Since there is no collision for h, we must have $z(v) = z'(v)$ and consequently $\mathsf{ZList}_{l(D)-1} = \mathsf{ZList}'_{l(D)-1}$. This in turn implies that for each $v \in S_{l(D)-1}$ we have $x(v) = x'(v)$ and for each $u \in S_{l(D)-2}$ we have $y(u) = y'(u)$. Hence $\mathsf{XList}_{l(D)-1} = \mathsf{XList}'_{l(D)-1}$ and $\mathsf{YList}_{l(D)-2} = \mathsf{YList}'_{l(D)-2}$.

For the induction step assume that we have shown $\mathsf{XList}_{i+1} = \mathsf{XList}'_{i+1}$ and $\mathsf{YList}_i = \mathsf{YList}'_i$ for all $i \geq k+1$. Then using an argument similar to the one given above it follows that $\mathsf{XList}_i = \mathsf{XList}'_i$ and $\mathsf{YList}_{i-1} = \mathsf{YList}'_{i-1}$. This shows that $\mathsf{XList}_i = \mathsf{XList}'_i$ for $1 \leq i \leq l(D) - 1$. Now one more application of the previous argument shows that $\mathsf{XList}_0 = \mathsf{XList}'_0$. Hence $\mathsf{XList}_i = \mathsf{XList}'_i$ for all $0 \leq i \leq l(D) - 1$ as desired. ■

3 Hashing Arbitrary Length Strings

The hash function $h_{\mathcal{S}}$ can handle only strings of one particular length. We would like to obtain a function which can handle strings of any length. Techniques to handle arbitrary length strings have been introduced before by Damgård [7] (see also Stinson [14]) for the special case of structures where the underlying DAG is a directed path. It does not seem to be easy to adapt the technique of [7] to the more general case of DAG that we consider here. Thus, we present a new method for handling arbitrary length strings, which is also of independent interest. To describe the construction of hash function which can handle arbitrary length strings we need to introduce an infinite family of DAGs. The most general situation is the case when there are no restrictions on the DAGs occurring in the family. While this can be done, the description becomes quite complicated. To keep the description reasonably simple, we assume that each DAG in the family has a single output node. The precise definition of the family that we consider is given below.

Let $\{D_k\}_{k \geq 1}$ be a family of DAGs where $D_k = (V_k, A_k)$ is such that $|V_k| = k$ and D_k has exactly one output node. Given positive integers n and m with $n > m$, a family of structures $\mathcal{F}$ is defined

as $\mathcal{F} = \{\mathcal{S}_k\}_{k\geq 1}$ where $\mathcal{S}_k = (n, m, D_k, \alpha_k)$. Given a compression function $h : \{0,1\}^n \to \{0,1\}^m$ and a family of structures $\mathcal{F}$ we define a family of hash functions $\{h_k\}_{k\geq 1}$, where $h_k = h_{\mathcal{S}_k}$.

Proposition 4 *Let $\{h_k\}_{k\geq 1}$ be a family of hash functions defined as above. Then*

$$h_k : \{0,1\}^{k(n-m)+m} \to \{0,1\}^m.$$

Remark : Note that $h_1 = h$. From Theorem 3 we know that the ability to find a collision for any h_k implies the ability to find a collision for h.

We want to define a hash function which can handle strings of any length. Each h_k can handle only fixed length strings. More precisely, h_1 can handle strings of length n, h_2 can handle strings of length $2n - m$, h_3 can handle strings of length $3n - 2m$ and so on. First we need to "fill the gaps" in the lengths. For this we define a function $h^* : \cup_{i\geq 1}\{0,1\}^i \to \{0,1\}^m$ in the following manner.

$$\left.\begin{array}{rcll} h^*(x) & = & h_1(x||0^{n-|x|}) & \text{if } 1 \leq |x| \leq n; \\ & = & h_{k+1}(x||0^{(k+1)(n-m)+m-|x|}) & \text{if } k(n-m)+m < |x| \leq (k+1)(n-m)+m. \end{array}\right\} \quad (1)$$

Note that the amount of padding done to x in the definition of h^* is at most $(n-1)$ in the first case and at most $(n-m-1)$ in the second case. The function $h^*(x)$ is not collision resistant. For example, the images of the strings 1 and 10^{n-1} are same, since $h^*(1) = h(10^{n-1}) = h^*(10^{n-1})$. We modify the function $h^*(x)$ to a function $h^\infty(x) : \cup_{i\geq 1}\{0,1\}^i \to \{0,1\}^m$ which is collision resistant (assuming that h is collision resistant). To do this we first need to introduce a length extracting function.

Given a binary string x, let $\chi(x)$ denote the minimum length binary representation of the length of x. For example, if $x = 110001101010$, then $\chi(x) = 1100$, since the length of x is 12. The iterates of $\chi()$ are defined as usual: $\chi^0(x) = x$ and for $i > 0$, $\chi^i(x) = \chi(\chi^{i-1}(x))$. The following result states some simple properties of the function $\chi()$.

Proposition 5 *Let x be a binary string. Then*

1. *The first bit of $\chi(x)$ is 1.*
2. *$\chi(x) = x$ if and only if $x = 1$ or $x = 10$.*
3. *$|\chi(x)| = 1 + \lfloor \log|x| \rfloor = \lceil \log(|x|+1) \rceil$.*
4. *If $|x| > 1$, then there is a positive integer j, such that $\chi^j(x) = 10$.*

Now we are in a position to define the function h^∞. Recall that x^r denotes the reverse of the string x. Let IV be an initialization vector, i.e., a string of length m.

Computation of $\mathbf{h^\infty(x)}$.

1. Define $X_0 = x$ and for $i > 0$, define $X_i = \chi^i(X_0) = \chi(X_{i-1})$.
2. Let j be the least positive integer such that $|X_j| \leq n - m$.
3. Define $Y_0 = h^*(\mathsf{IV}||0||X_0)$.
4. For $1 \leq i \leq j-1$, define $Y_i = h^*(Y_{i-1}||1||X_i)$.
5. $Y_j = h^*(Y_{j-1}||X_j^r)$.
6. Output Y_j.

Remark : Note that for the above construction to work, there must exist a j such that $|X_j| \leq n-m$. If $n-m=1$ and $|x|>1$, then this cannot be achieved. Thus henceforth we will assume $n-m \geq 2$. From a practical point of view, this is not really a constraint since all known compression functions satisfy this condition.

We next prove that h^∞ is collision resistant if h is collision resistant.

Theorem 6 *Let h^∞ be defined as above. Then a collision for h^∞ yields a collision for h.*

Proof. Given any message x, the computation of the digest involves several invocations of the function h^*. At each stage, the function h^* in turn invokes h_k on a suitably padded string. There are $(j+1)$ invocations of h^*. Suppose that at the ith $(0 \leq i \leq j)$ invocation of h^*, the function h_{k_i} is invoked. Also denote the padded input to h_{k_i} by W_i. Thus $Y_0 = h^*(\mathsf{IV}||0||X_0) = h_{k_0}(W_0)$, for $1 \leq i \leq j-1$, $Y_i = h^*(Y_{i-1}||1||X_i) = h_{k_i}(W_i)$ and $Y_j = h^*(Y_{j-1}||X_j^r) = h_{k_j}(W_j)$. Further, we have $|W_i| = k_i(n-m)+m$ and $0 \leq i \leq j$, $|Y_i| = m$.

Assume $h^\infty(x) = h^\infty(x')$ and $x \neq x'$. We show that this implies that there is a collision for h. The proof is by backward induction. We will use primed and unprimed notation to denote the quantities corresponding to x and x' respectively.

By hypothesis, we have $h^\infty(x) = Y_j = Y'_{j'} = h^\infty(x')$. From the definition of h^∞ we have

$$h^*(Y_{j-1}||X_j) = h_{k_j}(W_j) = Y_j = Y'_{j'} = h_{k'_{j'}}(W'_{j'}) = h^*(Y'_{j'-1}||X'_{j'}).$$

By definition of j and j', we have $|X_j|, |X'_{j'}| \leq n-m$ and hence $|Y_{j-1}||X_j|, |Y'_{j'-1}||X'_{j'}| \leq n$. From the definition of h^* it follows that $k_j = k'_{j'} = 1$ and $|W_j| = |W'_{j'}| = n$. If $W_j \neq W'_{j'}$ then we obtain a collision for h_1 and hence for h (since $h = h_1$) and we are done. On the other hand, if there is no collision for h, we must have $W_j = W'_{j'}$. Hence

$$W_j = Y_{j-1}||X_j^r||0^{n-m-|X_j|} = Y'_{j'-1}||X_{j'}^{\prime r}||0^{n-m-|X'_{j'}|} = W'_{j'}.$$

Since both X_j^r and $X_{j'}^{\prime r}$ end with an 1, by the above condition we must have $|X_j^r| = |X_{j'}^{\prime r}|$. This implies $Y_{j-1} = Y'_{j'-1}$ and $X_j = X'_{j'}$. Now there are two cases to consider.

Case $\mathbf{j = j'}$: We have $\chi(X_{j-1}) = X_j = X'_{j'} = \chi(X'_{j'-1})$ and hence $|X_{j-1}| = |X'_{j'-1}|$. Thus $|W_{j-1}| = |W'_{j'-1}|$ and consequently $k_{j-1} = k'_{j'-1}$. Also we have

$$h_{k_{j-1}}(W_{j-1}) = Y_{j-1} = Y'_{j'-1} = h_{k'_{j'-1}}(W'_{j'-1}).$$

Using Theorem 3 we obtain that either $W_{j-1} = W'_{j'-1}$ or we obtain a collision for h. Since we are assuming that there is no collision for h, we obtain $W_{j-1} = W'_{j'-1}$ and consequently $Y_{j-2} = Y'_{j'-2}$ and $X_{j-1} = X'_{j'-1}$.

Repeating the above argument for $i = j-2, \ldots, 1$, we obtain that $W_{j-2} = W'_{j'-2}$, $W_{j-3} = W'_{j'-3}$, $\ldots$, $W_1 = W'_1$ and consequently $Y_{j-3} = Y'_{j'-3}$ and $X_{j-2} = X'_{j'-2}$, $Y_{j-4} = Y'_{j'-4}$ and $X_{j-3} = X'_{j'-3}$, $\ldots$, $Y_0 = Y'_0$ and $X_1 = X'_1$. Now we have

$$\chi(X_0) = X_1 = X'_1 = \chi(X'_0).$$

Consequently, $|X_0| = |X'_0|$ and so $|W_0| = |W'_0|$. This forces $k_0 = k'_0$. Thus we have

$$h_{k_0}(W_0) = Y_0 = Y'_0 = h_{k_0}(W'_0).$$

Again using Theorem 3, we have that either there is a collision for h or $W_0 = W_0'$. In the first case we are done and in the second case, we have $W_0 = W_0'$ and hence $x = X_0 = X_0' = x'$ which contradicts the hypothesis. Hence there is a collision for h.

Case $\mathbf{j \neq j'}$: Without loss of generality assume $j' > j$ and $j' - j = l > 0$. Proceeding as in the above case, we have $Y_0 = Y_l'$ and $X_1 = X_{l+1}'$. Again $\chi(X_0) = X_1 = X_{l+1}' = \chi(X_l')$ and hence $|X_0| = |X_l'|$ which implies $|W_0| = |W_l'|$. This forces $k_0 = k_l'$. Thus we have

$$h_{k_0}(W_0) = Y_0 = Y_l' = h_{k_l'}(W_l').$$

The string W_0 is formed by padding 0's to the end of $\mathsf{IV}||0||X_0$ and the string W_l' is formed by padding 0's to the end of $Y_{l-1}'||1||X_l'$. Thus W_0 and W_l' differ in the $(m+1)$th bit position and hence $W_0 \neq W_l'$. Hence by Theorem 3 there must be a collision for h.

This proves the result. ■

Remark : Note that it is easy to find a collision for h given a collision for h^∞. The task involved is to invoke h^∞ for both x and x' and then "scan backwards" until a collision for h is found.

3.1 Padding Efficiency

The function h^∞ performs some amount of padding to the string x before hashing it. We determine the maximum amount of padding that is done and show that this is (asymptotically) less than the amount of padding performed in the Merkle-Damgård construction. Given integer i, we define $\log^*(i)$ to be the least integer k such that

$$\underbrace{\log(\log(\ldots(\log(}_{k}|x|)\ldots)) \leq 1.$$

Note that the parameters n and m of the compression function h are independent of the message length $|x|$ and can be assumed to be constant in an asymptotic analysis.

Proposition 7 *Let x be a binary string with $|x| > n$. Then the maximum amount of padding done to the string x in the computation of $h^\infty(x)$ is*

$$n + j(n-m) + |\chi(x)| + |\chi^2(x)| + \cdots + |\chi^{j-1}(x)|$$

where j is the minimum positive integer such that $|\chi^j(x)| \leq n - m$.

Proof. The maximum amount of padding in Step 3 is $m + 1 + (n - m - 1)$. In Step 4, there is a loop; for each value of i $(1 \leq i \leq j-1)$ the maximum amount of padding is $1 + |X_i| + (n-m-1) = (n-m) + |\chi^i(x)|$. The padding in Step 5 is equal to $(n-m)$. Adding up all these gives the required result. ■

The maximum amount of padding done to x in the Merkle-Damgård algorithm is $(n-1) + (n - m - 1) + \left\lceil \frac{|x|}{n-m-1} \right\rceil$ (see [14]). Assuming n and m to be constants, the amount of padding is $O(|x|)$. On the other hand, assuming n and m to be constants, the maximum amount of padding in our algorithm is $O(\log^* |x| \log |x|)$. Hence in an asymptotic sense our padding scheme is more efficient than the Merkle-Damgård padding scheme. The asymptotic inefficiency in the Merkle-Damgård construction arises due to the fact that one bit of padding is done to each message block. However, we note that for practical sized messages, the amount of padding in both our scheme and the Merkle-Damgård scheme will roughly be the same.

3.2 Invocation Efficiency

We compare the invocation efficiency of our algorithm to the Merkle-Damgård algorithm, i.e., we compare the number of invocations of the compression function h for a message x by the Merkle-Damgård algorithm and our algorithm.

We first compute the number of invocations of h made by our algorithm. The algorithm to compute h^∞ invokes h^* exacty $j+1$ times, i.e., for $i = 0, \ldots, j$. Suppose as in the proof of Theorem 6 that the ith invocation of h^* is made on the string W_i which is obtained by possibly padding 0s to $\mathsf{IV}||0||X_0$ if $i = 0$; to $Y_{i-1}||1||X_i$ if $1 \leq i \leq j-1$; and to $Y_{j-1}||X_j$ if $i = j$. Again as in the proof of Theorem 6 assume that h^* invokes the function h_{k_i} on the string W_i. Then from Proposition 2 it follows that $|W_i| = (n-m)(k_i - 1) + n$. Now $|W_i| = m + 1 + |X_i| + |z_i|$ for $0 \leq i \leq j-1$ and $|W_j| = m + |X_j| + |z_j|$, where z_is are the all zero strings which are used as pads to obtain the W_is. Further, $|z_i| \leq n - m - 1$ for all $0 \leq i \leq j$. Thus we obtain

$$\left.\begin{aligned} k_i &= \left\lceil \frac{|X_i|+1}{(n-m)} \right\rceil \quad \text{if } 0 \leq i \leq j-1; \\ &= \left\lceil \frac{|X_i|}{(n-m)} \right\rceil \quad \text{if } i = j. \end{aligned}\right\} \tag{2}$$

The value of k_i is the number of invocations of h made by h_{k_i}. Note that $k_j = 1$. Hence the total number of invocations of h made in the computation of h^∞ is obtained by adding all the k_is and is given in the following result.

Proposition 8 *The total number of invocations of h made in the computation of h^∞ is equal to*

$$\left\lceil \frac{|x|+1}{n-m} \right\rceil + \left\lceil \frac{|\chi(x)|+1}{n-m} \right\rceil + \cdots + \left\lceil \frac{|\chi^{j-1}(x)|+1}{n-m} \right\rceil + 1$$

In the Merkle-Damgård algorithm, the number of invocations A of the compression function h is equal to

$$A = 1 + \left\lceil \frac{|x|}{n-m-1} \right\rceil. \tag{3}$$

On the other hand, the number of invocations B of h in our algorithm is given by Proposition 8. Note that $j \leq \log^* |x|$. Using this fact and some simple algebraic simplification we obtain

$$\begin{aligned} A - B &> \frac{|x|}{(n-m)(n-m-1)} - \frac{n-m+1+\log^* |x|(n-m+2+\log|x|)}{n-m} \\ &> 0 \end{aligned}$$

for sufficiently large $|x|$. Thus in an asymptotic sense, our algorithm is more efficient than the Merkle-Damgård algorithm. Again we note that for practical sized messages both algorithms have roughly the same performance.

4 Concrete Examples

In this section, we provide some examples of DAGs which can be used to extend the domain of a collision resistant compression function. To do this it will be easier to define a notion of composition of structures in the following manner.

Let $\mathcal{S}_1 = (n, m, D_1 = (V_1, A_1), \alpha_1)$ and $\mathcal{S}_2 = (n, m, D_2 = (V_2, A_2), \alpha_2)$ be two structures such that the number of output nodes of D_1 is at most equal to the number of exposed nodes of D_2. Let $\{u_1, \ldots, u_r\}$ be the output nodes of D_1 and $\{v_1, \ldots, v_s\}$ $(r \leq s)$ be the exposed nodes of D_2. Define a DAG $D = (V, A)$, where $V = V_1 \cup V_2$ and $A = A_1 \cup A_2 \cup \{(u_1, v_1), \ldots, (u_r, v_r)\}$. Define a proper assignment α on D in the following manner: $\alpha(e) = \alpha_i(e)$ if $e \in A_i$, $i = 1, 2$ and $\alpha(e) = m$ otherwise. We define $\mathcal{S} = \mathcal{S}_1 \bullet \mathcal{S}_2$ to be the *partial composition* of $\mathcal{S}_1$ and $\mathcal{S}_2$ where $\mathcal{S} = (n, m, D, \alpha)$. In case $r = s$, i.e., the number of output nodes of D_1 is equal to the number of exposed nodes of D_2 we will say that $\mathcal{S}$ is the total composition or simply the composition of $\mathcal{S}_1$ and $\mathcal{S}_2$. Also we will denote the total composition by the symbol $\circ$. Note that $\circ$ is an associative operation while $\bullet$ is not and neither of the two operations are commutative.

From now on we will explicitly write a structure as $\mathcal{S} = (n, m, r_1, r_2, D, \alpha)$ where r_1 (resp. r_2) is the number of exposed (resp. output) nodes of D. Thus we can compose $\mathcal{S}_1 = (n, m, r_1, r_2, D_1, \alpha_1)$ and $\mathcal{S}_2 = (n, m, r_2, r_3, D_2, \alpha_2)$ to obtain $\mathcal{S} = \mathcal{S}_1 \circ \mathcal{S}_2 = (n, m, r_1, r_3, D, \alpha)$. Let $h_{\mathcal{S}_1}, h_{\mathcal{S}_2}$ and $h_{\mathcal{S}}$ be the hash functions associated with the structures $\mathcal{S}_1, \mathcal{S}_2$ and $\mathcal{S}$ respectively. Then $h_{\mathcal{S}_1}$ is an $(t_1(n-m) + r_2 m, r_2 m)$ function, $h_{\mathcal{S}_2}$ is an $(t_2(n-m) + r_3 m, r_3 m)$ function and $h_{\mathcal{S}}$ is an $((t_1 + t_2)(n-m) + r_3 m, r_3 m)$ function where t_1 and t_2 are the numbers of nodes in D_1 and D_2 respectively.

We now provide some examples of structures. In each of the cases below we assume the existence of a suitable (n, m) compression function h.

Example 1 (isolated nodes): For $i \geq 1$, define $\mathcal{K}_i = (n, m, i, i, D = (\{1, \ldots, i\}, \emptyset), \alpha)$ to be the structure corresponding to the digraph consisting of i nodes and no arcs. Hence each node is both an exposed and an output node. The depth of $\mathcal{K}_i$ is one. The associated hash function $h_{\mathcal{K}_i}$ is an (in, im) function.

Example 2 (dipath): For $r \geq 1$, define $P^{(r)}$ to be the directed path on r nodes and α assigns m to each arc of $P^{(r)}$. This defines a structure $\mathcal{P}^{(r)} = (n, m, 1, 1, P^{(r)}, \alpha)$. The depth of $\mathcal{P}^{(r)}$ is r. The associated hash function $h_{\mathcal{P}^{(r)}}$ is an $(r(n-m) + m, m)$ function. A variation of this structure (which includes an initialization vector) is used in the Merkle-Damgård construction [7, 8].

Example 3 (parallel dipaths): For $r, q \geq 1$, define $P_q^{(r)}$ to be the union of q copies of $P^{(r)}$ and the corresponding structure is denoted by $\mathcal{P}_q^{(r)} = (n, m, q, q, P^{(r)}, \alpha)$, where α again assigns m to each arc of $P_q^{(r)}$. The depth of $\mathcal{P}_q^{(r)}$ is r and also note that $\mathcal{K}_i = \mathcal{P}_i^{(1)}$. The associated hash function $h_{\mathcal{P}_q^{(r)}}$ is an $(rq(n-m) + qm, qm)$ function.

Example 4 (contracting binary tree): For $t \geq 1$, let T_t be the binary tree with t levels and $2^t - 1$ nodes defined by $T_t = (\{1, \ldots, 2^t - 1\}, \{(i, \lfloor i/2 \rfloor) : 2 \leq i \leq 2^t - 1\})$. We define an assignment α which assigns m to each arc of T_t. Then the fan-in of any non exposed node is $2m$ and since α is proper we must have $n \geq 2m$. We denote the corresponding structure by $\mathcal{T}_t = (n, m, 2^{t-1}, 1, T_t, \alpha)$. The depth of $\mathcal{T}_t$ is t. The associated hash function $h_{\mathcal{T}_t}$ is a $((2^t - 1)(n-m) + m, m)$ hash function.

Example 5 (expanding binary tree): For $t \geq 1$, let I_t be the inverted binary tree of t levels: $I_t = (\{1, \ldots, 2^t - 1\}, \{(i, 2i), (i, 2i+1) : 1 \leq i \leq 2^{t-1} - 1\})$. The assignment α assigns $(m/2)$ to each arc of I_t. We denote the corresponding structure by $\mathcal{I}_t = (n, m, 1, 2^{t-1}, I_t, \alpha)$. The depth of $\mathcal{I}_t$ is t. The associated hash function $h_{\mathcal{I}_t}$ is a $((2^t - 1)(n-m) + 2^{t-1} m, 2^{t-1} m)$ function.

Example 6 (parallel structure): For $t \geq 1$ and $r \geq 0$, define $\mathcal{S}_t^{(r)} = \mathcal{I}_t \circ \mathcal{P}_{2^{t-1}}^{(r)} \circ \mathcal{T}_t$. The numbers of exposed and output nodes of $\mathcal{S}_t^{(r)}$ are both one and its depth is $2t + r$. The associated hash

function $h_{\mathcal{S}_t^{(r)}}$ is a $((2^t(r+2)-2)(n-m)+m, m)$ function.

Example 7 (incremental parallel structure): For $r \geq 0$, $t \geq 1$ and $1 \leq s \leq 2^{t-1}$ define the structure

$$\mathcal{S}_t^{(r,s)} = (\mathcal{I}_t \circ \mathcal{P}_{2^{t-1}}^{(r)}) \circ (\mathcal{K}_s \bullet \mathcal{T}_t). \tag{4}$$

The numbers of exposed and output nodes of $\mathcal{S}_t^{(r,s)}$ are both one and its depth is $2t+r+1$. The hash function $h_{\mathcal{S}_t^{(r,s)}}$ associated to the structure $\mathcal{S}_t^{(r,s)}$ is an $((2^{t+1}+r2^{t-1}+s-2)(n-m)+m, m)$ function.

4.1 A Parallelizable Hash Algorithm

We build on Example 7 above to obtain a parallel algorithm for extending the domain of a collision resistant hash function. The algorithm will use 2^{t-1} processors and its structure will be $\mathcal{S}_t^{(r,s)}$ for some r and s which are determined by the length of the message x in the following manner. Define $\lambda(t) = (2^{t+1}-2)(n-m)+m$ and $\delta(t) = 2^{t-1}(n-m)$. Let x be the string which is to be hashed.

1. Write $|x| - \lambda(t) = r\delta(t) + \gamma_1$, where γ_1 is a unique integer from the set $\{1, \ldots, \lambda(t)\}$.
2. Write $\gamma_1 = \gamma_2(n-m) + \gamma_3$, where γ_3 is a unique integer from the set $\{1, \ldots, n-m\}$.
3. Set $x = x||0^{n-m-\gamma_3}$. Note that the amount of padding is at most $(n-m-1)$.

Note that $1 \leq \gamma_2 < 2^t$. The above steps define the parameters r, γ_1, γ_2 and γ_3. We further define $s = \gamma_2 + 1$. For $t \geq 1$, we define the function $g_t^* : \cup_{i \geq \lambda(t)} \{0,1\}^i \to \{0,1\}^m$ by

$$g_t^*(x) = h_{\mathcal{S}_t^{(r,s)}}(x||0^{n-m-\gamma_3}) \tag{5}$$

where $h_{\mathcal{S}_t^{(r,s)}}$ is the hash function associated with the structure $\mathcal{S}_t^{(r,s)}$. Note that the amount of padding done to the string x is at most $n-m-1$.

Remark : One constraint for using the structure $\mathcal{T}_t$ is that the (n,m) compression function h must satisfy $n \geq 2m$. The structure $\mathcal{S}_t^{(r,s)}$ contains the structure $\mathcal{T}_t$ and hence this constraint also holds for $\mathcal{S}_t^{(r,s)}$. However, from a practical point of view this is not really a constraint, since all known practical compression functions satisfy this condition.

We now define $H_t^* : \cup_{i \geq 1} \{0,1\}^i \to \{0,1\}^m$ in the following manner. First, for the sake of convenience, we need to define a function $g_0 : \cup_{i=n+1}^{2n-m-1} \{0,1\}^i \to \{0,1\}^m$ in the following manner. Let x be a string with $n < |x| < 2n-m$ and $w = x||0^{2n-m-|x|}$. Write $w = w_1||w_2$ where $|w_1| = n$ and $|w_2| = n-m$. We define $g_0(x) = h(h(w_1)||w_2)$.

$$\left.\begin{array}{rcll} H_t^*(x) & = & h(x||0^{n-|x|}) & \text{if } |x| \leq n; \\ & = & g_0(x) & \text{if } n < |x| < 2n-m; \\ & = & g_i^*(x) & \text{if } \lambda(i) \leq |x| < \lambda(i+1) \text{ and } 1 \leq i < t; \\ & = & g_t^*(x) & \text{if } |x| \geq \lambda(t). \end{array}\right\} \tag{6}$$

The desired function H_t^∞ is obtained from the function H_t^* using the construction of Section 3. In this construction also we use an initialization vector (IV) of length m.

Computation of $\mathbf{H_t^\infty(x)}$

1. Let $X_0 = x$ and for $i \geq 1$, $X_i = \chi(X_{i-1}) = \chi^i(X_0)$.
2. Let j be the least positive integer such that $|X_j| \leq n - m$.
3. Set $Y_0 = H_t^*(\mathsf{IV}||0||X_0)$.
4. For $i = 1$ to $j - 1$ set $Y_i = H_t^*(Y_{i-1}||1||X_i)$.
5. $Y_j = H_t^*(Y_{j-1}||X_j^r)$.
6. Output Y_j.

The collision resistance of H_t^∞ is proved in a manner similar to that of Theorem 6 and this gives us the following result.

Theorem 9 *Let H_t^∞ be the hash function defined as above. Then a collision for H_t^∞ yields a collision for h.*

5 Conclusion

We have considered the problem of securely extending the domain of a collision resistant hash function using an arbitrary DAG. A new technique for handling arbitrary length string have been introduced for which the amount of padding is asymptotically lesser than the amount of padding required for the Merkle-Damgård construction. Our third contribution has been to provide some concrete DAG based examples and provide a foundation for the design of an efficient parallel hash algorithm. We believe that it is possible to design a practical hash function based on this principle. In this paper, we have entirely concentrated on the property of collision resistance. In fact, all the domain extending techniques considered here also preserve the property of preimage resistance.

References

[1] M. Bellare and P. Rogaway. Collision-resistant hashing: towards making UOWHFs practical. *Lecture Notes in Computer Science*, Proceedings of CRYPTO 1997, pp 470-484.

[2] M. Bellare and D. Micciancio. A New Paradigm for Collision-Free Hashing: Incrementality at Reduced Cost. *Lecture Notes in Computer Science*, (Advances in Cryptology - EUROCRYPT 1997), pages 163-192.

[3] J. Black, P. Rogaway and T. Shrimpton. Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV. *Advances in Cryptology*, CRYPTO 2002: 320-335.

[4] A. Bosselaers, R. Govaerts and J. Vandewalle, SHA: A Design for Parallel Architectures? *Lecture Notes in Computer Science*, (Advances in Cryptology - Eurocrypt'97), pages 348-362.

[5] R. Canetti. Towards Realizing Random Oracles: Hash Functions that Hide All Partial Information. *Advances in Cryptology*, CRYPTO'97, LNCS 1294, pp 455-469.

[6] D. Chaum, E. van Heijst and B. Pfitzmann. Cryptographically strong undeniable signatures, unconditionally secure for the signer. *Lecture Notes in Computer Science*, 576 (1992), 470-484, (Advances in Cryptology - CRYPTO'91).

[7] I. B. Damgård. A design principle for hash functions. *Lecture Notes in Computer Science*, 435 (1990), 416-427 (Advances in Cryptology - CRYPTO'89).

[8] R. C. Merkle. One way hash functions and DES. *Lecture Notes in Computer Science*, 435 (1990), 428-226 (Advances in Cryptology - CRYPTO'89).

[9] P. Pal and P. Sarkar. PARSHA-256: A Parallelizable Hash Function and a Multithreaded Implementation, *Proceedings of FSE*, 2003.

[10] B. Preneel. The state of cryptographic hash functions. *Lecture Notes in Computer Science*, 1561 (1999), 158-182 (Lectures on Data Security: Modern Cryptology in Theory and Practice).

[11] C. Schnorr and S. Vaudenay. Parallel FFT-Hashing. *Lecture Notes in Computer Science*, Fast Software Encryption, LNCS 809, pages 149-156, 1994.

[12] P. Sarkar and P. J. Schellenberg. A Parallel Algorithm for Extending Cryptographic Hash Functions, *Proceedings of Indocrypt* 2001, LNCS pp 40-49.

[13] D. R. Stinson. Some observations on the theory of cryptographic hash functions. IACR preprint server, http://eprint.iacr.org/2001/020/.

[14] D. R. Stinson. *Cryptography: Theory and Practice*, CRC Press, 1995.