Domain Extender for Collision Resistant Hash Functions: Improving Upon Merkle-Damgård Iteration

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

Abstract

We study the problem of securely extending the domain of a collision resistant compression function. A new construction based on directed acyclic graphs is described. This generalizes the usual iterated hashing constructions. Our main contribution is to introduce a new technique for hashing arbitrary length strings. Combined with DAG based hashing, this technique gives a new hashing algorithm. The amount of padding and the number of invocations of the compression function required by the new algorithm is smaller than the general Merkle-Damgård algorithm. Lastly, we describe the design of a new parallel hash algorithm.

Keywords : hash function, compression function, composition principle, collision resistance, 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 including 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. Most practical hash functions are designed from scratch. 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 principle*.

Composition principles based on iterated applications of the compression function are known and these are called variants of the Merkle-Damgård algorithm [2, 4]. The most general of these algorithms can hash arbitrarily long messages and assumes the compression function to be only collision resistant. Other variants can hash messages of a maximum possible length or assumes the compression function to be both collision resistant and one-way. See Section 3 for a detailed discussion of several variants of the Merkle-Damgård algorithm.

Our Contributions: In this paper, we are concerned with the problem of constructing a hash function which can hash arbitrarily long messages and which can be proved to be collision resistant under the assumption that the compression function is collision resistant. To justify the non-triviality of the problem we describe a construction which can be proved to be secure if the compression function is both collision resistant and one-way while it is insecure if the compression function is only collision resistant.

The first step in our construction is to consider a very general class of domain extending algorithms. The structure of any algorithm in the class that we consider can be described using a directed acyclic graph (DAG). In Section 5, we provide a construction of a secure domain extending algorithm using an arbitrary DAG. The Merkle-Damgård algorithm uses a dipath and is a special case of DAG based algorithms.

Our main contribution (in Section 6) is to provide a solution to the problem of hashing arbitrary length strings for DAG based algorithms. Our algorithm improves upon the (general) Merkle-Damgård algorithm both in terms of padding length and number of invocations. Our construction can be proved to be collision resistant under the assumption that the compression function is *only* collision resistant.

In Section 8, we provide some concrete examples of hashing structures and show that these can be combined nicely to design a parallel hash function. We note, however, that we do not provide a detailed specification of an actual hash function. Such a specification will necessarily involve many practical and implementation issues which are not really within the scope of the current work.

A theoretical justification of our work is provided by the fact that our results improve upon a fifteen year old classical work. Since our work improves upon the Merkle-Damgård algorithm, a natural question is whether further improvements are possible. This naturally leads to the problem of obtaining non-trivial lower bounds (and optimal algorithms) on padding lengths and number of invocations. These problems can provide motivation for future research.

2 Preliminaries

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. The reverse of the string x will be denoted by x^r. By an (n, m) function we will mean a function which maps $\{0,1\}^n$ to $\{0,1\}^m$. All logarithms in the paper are in base two.

For $n > m$, let h be an (n, m) function. Two n-bit strings x and x' in X are said to collide for h, if $x \neq x'$ but $h(x) = h(x')$. A hash function $h : X \to Y$ is said to be collision resistant if it is computationally intractable to find collisions for h. A formal definition of this concept requires the consideration of a family of functions (see [2, 5]).

In this paper, we are interested in "securely" extending the domain of a hash function. More precisely, given an (n, m) function $h : \{0,1\}^n \to \{0,1\}^m$, with $n > m+1$, we construct a function $h^\infty : \cup_{i\geq 1}\{0,1\}^i \to \{0,1\}^m$, such that one can prove the following: Given any collision for h^∞, it is possible to obtain a collision for h. The last statement is formalized in terms of a Turing reduction between two suitably defined problems (see below). The advantage of this method is that we only prove a reduction and at no point are we required to use a formal definition of collision resistance. This approach has been previously used in the study of hash functions [6].

We now turn to the task of defining our approach to reducibilities between different problems related to the property of collision resistance. Consider the following problem as defined in [6].

Problem	:	Collision $\mathrm{Col}(n, m)$
Instance	:	An (n, m) hash function h.
Find	:	$x, x' \in \{0,1\}^n$ such that $x \neq x'$ and $h(x) = h(x')$.

By an (ϵ, q) (probabilistic) algorithm for Collision we mean an algorithm which invokes the hash function h at most q times and solves $\mathrm{Col}(n, m)$ with probability of success at least ϵ.

The domain of h is the set of all n-bit strings. We would like to extend the domain to the set of all nonempty binary strings, i.e., to construct a function $h^\infty : \cup_{i \geq 1}\{0,1\}^i \rightarrow \{0,1\}^m$. We would like to relate the difficulty of finding collisions for h^∞ to that of finding collisions for h. Thus, we consider the following problem.

Problem	:	Arbitrary length collision $\mathrm{ALC}(n, m, L)$
Instance	:	An (n, m) hash function h and an integer $L \geq 1$.
Find	:	$x, x' \in \cup_{i=1}^{L}\{0,1\}^i$ such that $x \neq x'$ and $h^\infty(x) = h^\infty(x')$.

By an (ϵ, q, L) (probabilistic) algorithm $\mathcal{A}$ for Arbitrary length collision we will mean an algorithm that makes at most q invocations of the function h and solves $\mathrm{ALC}(n, m, L)$ with probability of success at least ϵ.

Later we show Turing reductions from Collision to Arbitrary Length Collision. Informally, this means that given oracle access to an algorithm for solving $\mathrm{ALC}(n, m, L)$ for h^∞ it is possible to construct an algorithm to solve $\mathrm{Col}(n, m)$ for h. These will show that our constructions preserve the intractibility of finding collisions.

Pre-image resistance: This is an important property for cryptographic hash functions. Informally, this means that given $y \in \{0,1\}^m$, it is computationally infeasible to find an x, such that $f(x) = y$. Pre-image resistance (or one-wayness) is a crucially important property on its own. On the other hand, this property is sometimes used to prove security of domain extending techniques for collision resistant hash functions. Suppose the domain of an (n, m) hash function h is extended to obtain the hash function $H()$. For certain constructions [2], one can show that h^∞ is collision resistant if h is *both* collision resistant and one-way. We would like to emphasize that this is not the approach we will take in this paper. In our constructions, we will assume h to be *only* collision resistant.

3 Iterated Hashing

In this section, we briefly review iterative techniques for extending the domain of a collision resistant compression function. These techniques are attributed to [4, 2] and are commonly called the Merkle-Damgård constructions.

Let h be an (n, m) compression function and IV be an m-bit string. Each of the domain extending methods described below use IV and h to construct a new function which can hash "long" strings to obtain m-bit digest. The IV can be chosen randomly, but once chosen it cannot be changed and becomes part of the specification for the extended domain hash function.

3.1 Construction I: Basic Iteration

We define a hash function $H^{(\mathrm{I})}$ whose domain consists of all binary strings whose length is a multiple of $(n-m)$. Let x be a message whose length is $i(n-m)$ for some $i \geq 1$. We write $x = x_1||\cdots||x_i$, where each x_j is a string of length $(n-m)$. Define $z_1 = h(\mathsf{IV}||x_1)$ and for $j > 1$, define $z_j = h(z_{j-1}||x_j)$. The digest of x under $H^{(\mathrm{I})}$ is defined to be z_i, i.e., $H^{(\mathrm{I})} = z_i$.

The function $H^{(\mathrm{I})}$ can be proved to be collision resistant. Briefly, the argument proceeds as follows. Suppose x and x' are two strings such that $x \neq x'$ and $H^{(\mathrm{I})}(x) = H^{(\mathrm{I})}(x')$. If we have $|x| = |x'|$, then an easy backward induction shows that there must be a collision for the function h. On the other hand, if $|x| \neq |x'|$, then it can be argued that the collision for $H^{(\mathrm{I})}$ either leads to a collision for h or a pre-image of IV under h. Thus, if we assume that h is *both* collision resistant *and* pre-image resistant, then $H^{(\mathrm{I})}$ is collision resistant.

3.2 Construction II: General Construction

Our description of the general version (which appears in [2]) is from [7] for the case $n - m > 1$. (The case $n - m = 1$ is a little more complicated. We do not mention it here since we will not consider such values of n and m for our constructions.)

Let $H^{(\mathrm{II})}$ be the extended domain hash function which is to be defined. Let x be a message to be hashed and we have to define the digest $H^{(\mathrm{II})}(x)$. Write $x = x_1||x_2||\ldots||x_k$, where $|x_1| = |x_2| = \cdots = |x_{k-1}| = n - m - 1$ and $|x_k| = n - m - 1 - d$ with $0 \leq d \leq n - m - 2$. For $1 \leq i \leq k-1$, let $y_i = x_i$; $y_k = x_k||0^d$ and y_{k+1} is the $(n-m-1)$-bit binary representation of d. Define $z_1 = h(\mathsf{IV}||0||y_1)$ and for $1 \leq i \leq k$, define $z_{i+1} = h(z_i||1||y_{i+1})$. The digest of x under $H^{(\mathrm{II})}$ is z_{k+1}, i.e., $H^{(\mathrm{II})}(x) = z_{k+1}$.

Note that the domain consists of all possible binary strings, i.e., there is no length restriction on the input message x. It can be shown that $H^{(\mathrm{II})}$ is collision resistant assuming h to be *only* collision resistant. (See [7] for a proof.)

3.3 Construction III: SHA Family Construction

The specification of the SHA family of constructions uses a variant of the iterative hashing technique. We denote this variant by $H^{(\mathrm{III})}$.

Let x be the message to be hashed. First we form the string: $\mathsf{pad}(x) = x||1||0^k||\mathsf{bin}_c(|x|)$, where c is a constant such that $c < n-m$, $\mathsf{bin}_c(|x|)$ is the c-bit binary representation of x and k is the least non-negative integer such that $|x| + 1 + k \equiv (n - m - c) \bmod (n - m)$, or equivalently $x + c + 1 + k \equiv 0 \bmod (n - m)$. The length of $\mathsf{pad}(x)$ is equal to $l(n - m)$ for some $l \geq 1$. (For SHA-256, $n = 768$, $m = 256$ and $c = 64$.) The message digest is defined to be $H^{(\mathrm{III})}(x) = H^{(\mathrm{I})}(\mathsf{pad}(x))$.

This construction can only handle messages of lengths less than 2^c. Putting $c = 64$ (as in SHA-256) is usually sufficient for all practical purposes. The maximum amount of padding is $n-m$ which is a constant, i.e., independent of the message length.

3.4 Construction IV: Another Length Bounded Construction

We define a function $H^{(\mathrm{IV})}$ which like $H^{(\mathrm{III})}$ can also hash all binary strings of a maximum possible length.

Let the message be x. Append the minimum number of zeros to x so as to make the length a multiple of $(n-m)$. Now divide x into l blocks $x_0, \ldots, x_{l-1}$ of lengths $(n-m)$ bits each. Define $y_0 = h(\mathsf{IV}||x_0)$ and for $1 \leq i \leq l-1$, define $y_i = h(y_{i-1}||x_i)$. Finally define $z = h(y_{l-1}||w)$, where w is the $(n-m)$-bit binary

Table 1: Comparison of features of different constructions for a message x.

Cons.	domain sz.	length res.	padding	# invoc.	assumption on $h()$
I	infinite	$\|x\| = i(n-m)$, $i \geq 1$	none	$\frac{\|x\|}{n-m}$	c.r. and one-way
II	infinite	none	$2n-m-2$ $+\left\lceil \frac{\|x\|}{n-m-1} \right\rceil$	$1+\left\lceil \frac{\|x\|}{n-m-1} \right\rceil$	c.r.
III	2^c, $c < n-m$	$\|x\| < 2^c$, $c < n-m$	m	$a+\left\lceil \frac{\|x\|}{n-m} \right\rceil$, $a \in \{0,1\}$	c.r.
IV	$< 2^{n-m}$	$\|x\| < 2^{n-m}$	$2n-m-1$	$1+\left\lceil \frac{\|x\|}{n-m} \right\rceil$	c.r.

representation of $|x|$, i.e. $w = \mathsf{bin}_{n-m}(|x|)$. The digest of x is z. Clearly, this algorithm can be applied only when the length of x is less than 2^{n-m}. Again, this construction can be proved to be collision resistant assuming h to be *only* collision resistant.

3.5 Role of IV

Each of the constructions described above use an m-bit string as an IV. The IV is essential in Construction I, since in this construction we require h to be such that it is infeasible to find a pre-image of IV under h. On the other hand, for Constructions II to IV, we can replace IV by the initial m bits of the message without affecting the collision resistance of the extended domain hash function. If we do this, then in certain cases, we can hash an extra m bits without increasing the number of invocations of h. In general, this is not a significant gain, though it may become significant if we repeatedly hash short messages such as digital certificates.

3.6 Discussion

In Table 1, we compare the properties of the different constructions. For each construction, we provide the size of the extended domain; the restriction on the lengths of messages to be hashed; the maximum amount of padding; the maximum number of invocations of $h()$ that are made while extending the domain; and the security assumption made on $h()$. (In our count of the number of padded bits, we also include the IV.) The first construction is proved to be collision resistant under the assumption that $h()$ is *both* collision resistant *and* one-way, while the other three constructions can be proved to be collision resistant under the assumption that $h()$ is *only* collision resistant. Construction II can handle arbitrary length strings, while the Constructions III and IV can handle bounded length strings. On the other hand, Constructions III and IV are more efficient than Construction II.

Question: The theoretical question that now arises is whether it is possible to obtain a construction which can handle arbitrary length strings, whose collision resistance is based *only* on the collision resistance of h and which is more efficient than Construction II?

4 Difficulty of Domain Extension

We would like to provide some evidence that it is non-trivial to obtain an answer to the question raised in Section 3.6. It is often believed that "padding with the length at the end is sufficient to ensure collision resistance". Investigating such a claim in full generality is difficult. Instead, we consider a "natural" extension of Construction III (the SHA family construction) to arbitrary length strings and show the following two facts.

- It is correct if we assume h to be both collision resistant and one-way on IV but
- It is incorrect when we assume h to be only collision resistant.

For an integer i, let $\mathsf{bin}(i)$ denote the minimum length binary representation of i and for a binary string x, let $\chi(x)$ denote the minimum length binary representation of the length of x, i.e., $\chi(x) = \mathsf{bin}(|x|)$.

Construction V: We want to define a function $H^{(\mathrm{V})}$ which can handle arbitrary length strings. As in Construction III, define

$$\begin{aligned} \mathsf{pad}(x) &= x||1||0^k||\mathsf{bin}(|x|) \\ &= x||1||0^k||\chi(x) \end{aligned}$$

where k is the minimum non-negative integer which satisfies the equation $|x|+|\chi(x)|+1+k \equiv 0 \bmod (n-m)$. This ensures that the length of $\mathsf{pad}(x)$ is equal to $l(n-m)$ for some $l \geq 1$ and hence we can apply the iterative technique as in Construction III to compute the message digest. (The exact Construction III is obtained by substituting $\mathsf{bin}_c(|x|)$ for $\mathsf{bin}(x)$.)

The digest of x under $H^{(\mathrm{V})}$ is defined to be $H^{(\mathrm{I})}(\mathsf{pad}(x))$, i.e., $H^{(\mathrm{V})}(x) = H^{(\mathrm{I})}(\mathsf{pad}(x))$. Since we do not put any bound on the length of $\mathsf{bin}(|x|)$, this construction can handle arbitrary length strings. Let us now consider the correctness of Construction V.

Condition 1: Suppose h is *both* collision resistant *and* it is infeasible to find a pre-image of IV. Then, using an argument as in the case of Construction I, it is possible to show by a backward induction that a collision for $H^{(\mathrm{V})}$ either provides a collision for h or a pre-image of IV under h.

Condition 2: Suppose that we want to assume h to be *only* collision resistant. We show that assuming h to be only collision resistant is not sufficient to show the correctness of Construction V. Let us consider the meaning of this statement in more details. Suppose that there is some element in the range of h which has a unique pre-image. Then the ability to find this pre-image (or even knowing it a priori) does not violate the collision resistance of h. On the other hand, the knowledge of this pre-image can make it possible to construct a collision for $H^{(\mathrm{V})}$. This is the approach that we take below.

Our first task is to choose a suitable collision resistant h. For this, we must assume that some function $h'()$ with suitable parameters is collision resistant, as otherwise the question is moot. (See [1] for a similar situation in regard to universal one-way hash functions.)

Suppose $h'()$ is an (n, m') collision resistant function, with $m' = m - 1$ and $n - m = 2^\tau \geq 16$. Further, let IV and σ be arbitrary m-bit and $(n-m)$-bit strings respectively. Using h', we define an (n, m) function h for which it is infeasible to find collisions and for which $\mathsf{IV}||\sigma$ is the only pre-image of IV. Write $\mathsf{IV} = \mathsf{IV}'||b$,

where $|\mathsf{IV}'| = m-1$ and b is a bit. For any n-bit string x, define

$$\left.\begin{array}{rcll} h(x) & = & \mathsf{IV} & \text{if } x = \mathsf{IV}||\sigma; \\ & = & \mathsf{IV}'||(1-b) & \text{if } h'(x) = \mathsf{IV}' \text{ and } x \neq \mathsf{IV}||\sigma; \\ & = & h'(x)||0 & \text{if } h'(x) \neq \mathsf{IV}'. \end{array}\right\} \quad (1)$$

Clearly, IV has the unique pre-image $\mathsf{IV}||\sigma$ under h. On the other hand, any collision for h yields a collision for h'. Hence, h is collision resistant if h' is collision resistant. (Note that h is not surjective, but that is not relevant to the assumption that h is collision resistant.) Note that if we use Construction I to extend the domain of h, then we get a function $H^{(\mathrm{I})}$ with the following property: $H^{(\mathrm{I})}(\sigma^i) = \mathsf{IV}$ for all $i \geq 1$, where σ^i denotes i many repetitions of σ.

The conversion from h' to h works for any IV and σ. Choose IV to be an arbitrary m-bit string; and $\sigma = y'||1||0^{\tau-1}||1$, where y' is an arbitrary string of length $(n-m-1-\tau)$. Then we can define the function h as above. (The justification for choosing σ as above will become clear later.)

Consider the function $H^{(\mathrm{V})}$. This function is defined for any h and IV and hence also for h and IV defined as above. We show that for such h and IV it is possible to exhibit a collision for $H^{(\mathrm{V})}$.

We define two strings x and x' in the following manner. String x is a "short" string, while string x' is a "long" string. Define $x = 0^{n-m-1-\tau}$ and then $\chi(x) = \lceil \log(n-m-1-\tau) \rceil = \tau$ and hence

$$\mathsf{pad}(x) \quad = \quad 0^{n-m-1-\tau}||1||\chi(x).$$

Note that in this case $k = 0$ and $|\mathsf{pad}(x)| = n-m$.

We now define the string x'. First we set the length of x' by defining $\chi(x') = 1||\mathsf{pad}(x)$ and hence $|\chi(x')| = n-m+1$. This sets the length of x' to be $2^{n-m} + (n-m) + |x|$. At this point, we know $\mathsf{pad}(x') = x'||1||0^{\tau-1}||\chi(x')$. This sets the length of $\mathsf{pad}(x')$ to be $2^{n-m} + 3(n-m)$. We write $x' = z'||y'$ where $|z'| = 2^{n-m}+n-m$. (Note $|y'| = |x|$, and we could, if we like, choose $y' = x$.) Recall $\sigma = y'||1||0^{\tau-1}||1$ and is of length $(n-m)$. We define z' to be i many repetitions of σ, i.e., $z' = \sigma^i$, where $i = 1 + 2^{n-m-\tau}$. Thus, we can write

$$\mathsf{pad}(x') = \sigma^{2+2^{n-m-\tau}}||\mathsf{pad}(x)$$

i.e., $2 + 2^{n-m-\tau}$ repetitions of σ followed by $\mathsf{pad}(x)$. Now,

$$\begin{array}{rcl} H^{(\mathrm{V})}(x') & = & H^{(\mathrm{I})}(\mathsf{pad}(x')) \\ & = & H^{(\mathrm{I})}(\sigma^{2+2^{n-m-\tau}}||\mathsf{pad}(x)) \\ & = & h(H^{(\mathrm{I})}(\sigma^{2+2^{n-m-\tau}}), \mathsf{pad}(x)) \\ & = & h(\mathsf{IV}, \mathsf{pad}(x)) \\ & = & H^{(\mathrm{I})}(\mathsf{pad}(x)) \\ & = & H^{(\mathrm{V})}(x). \end{array}$$

Clearly, $x \neq x'$ and hence we obtain a collision for $H^{(\mathrm{V})}$. Thus, $H^{(\mathrm{V})}$ is not collision resistant, even though h is collision resistant. In fact, in the proof we have used the fact that IV has a unique and known pre-image under h.

In view of this, we consider the problem of extending the domain of a collision resistant hash function to be a non-trivial problem.

5 DAG Hashing

So far, we have considered iterated hashing. Our main task will be to provide a new construction for securely extending the domain of a collision resistant hash function. We actually do this for a general class

of hashing algorithms whose structure can be described using a directed acyclic graph (DAG).

A DAG D is defined as $D = (V, A)$ 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. For each output node v of D, set $\mathsf{level}(v) = l(D) - 1$; drop all the output nodes from D to get a new DAG D_1. For each output node of v of D_1, set $\mathsf{level}(v) = l(D) - 2$; again drop all the output nodes from D_1 to get a new DAG D_2. Continue this process until all nodes of D have been assigned level numbers. 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.

5.1 Construction

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 as is made precise below. The substring of the message which is provided as input to v is denoted by $x(v)$ and is of length $|x(v)| = n - \mu(v)$. 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 $\mathbf{h_{\mathcal{S}}(x)}$

1. For $i = 0$ to $l(D) - 1$ do
2. For $v_j \in S_i$
3. set $y_j = h(z_j)$.
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.

5.2 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 $h : \{0,1\}^n \rightarrow \{0,1\}^m$ be a compression function. Then $h_{\mathcal{S}} : \{0,1\}^N \rightarrow \{0,1\}^M$ 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 length of the output of each node is m bits, hence the length of the output of $h_{\mathcal{S}}$ is sm bits.

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. ■

The next result shows 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 distinct 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 an appropriate 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. ■

6 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 [2] (see Construction II in Section 3.2) 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 [2] 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. 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 $\mathcal{S}_k = (n, m, D_k, \alpha_k)$, where α_k is a proper assignment on D_k. Given a compression function $h : \{0,1\}^n \rightarrow \{0,1\}^m$, and a family of structures $\mathcal{F}$, we define a family of hash functions $\{h_k\}_{k \geq 1}$, where $h_k = h_{\mathcal{S}_k}$. From Proposition 2, we have

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

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 $h^* : \cup_{i\geq 1}\{0,1\}^i \rightarrow \{0,1\}^m$ in the following manner.

$$\left.\begin{array}{rcll} h^*(x) & = & h_1(x||0^{n-|x|}) & \text{if } 1 \leq |x| \leq n; \\ & = & h_{k+1}(x||0^{(k+1)(n-m)+m-|x|}) & \text{if } k(n-m)+m < |x| \leq (k+1)(n-m)+m. \end{array}\right\} \quad (2)$$

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 $h^\infty(x) : \cup_{i\geq 1}\{0,1\}^i \rightarrow \{0,1\}^m$ 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, recall that $\chi(x)$ denotes 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 $\chi()$. Recall that the reverse of a binary string y is denoted by y^r.

Proposition 4 *Let x be a binary string. Then*

1. *The first bit of $y = \chi(x)$ is 1 and hence the last bit of y^r is also 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 $\chi^j(x) = 10$.*

Remark : For the construction of h^∞ given below 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 practical compression functions satisfy this condition.

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 : The value of j in the above algorithm will be more than one only if the length of the message is greater than 2^{n-m}. For practical compression functions (such as SHA, RIPEMD, etc.) the value of $(n-m)$ is at least 128. Thus, for all practical compression functions and practical sized messages the value of j will be equal to one.

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

Theorem 5 *If there is an (ϵ, q, L)-algorithm to solve $\mathrm{ALC}(n, m, L)$ for h^∞, then there is an $(\epsilon, q+2\eta)$-algorithm to solve $\mathrm{Col}(n, m)$ for h, where η is the number of invocations of h made by h^∞ in hashing a message of length L.*

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 for $0 \leq i \leq j$, $|Y_i| = m$.

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

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

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 the strings X_j^r and $X_{j'}^{\prime r}$ end with a 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. In the second case, we are done and in the first case 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 (possibly) padding 0's to the end of $\mathsf{IV}||0||X_0$ and the string W'_l is formed by (possibly) 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.

Let $\mathcal{A}$ be an (ϵ, q, L)-algorithm to solve ALC(n, m, L) for h^∞. Then $\mathcal{A}$ is successful with probability ϵ and in this case let (x, x') be the output of $\mathcal{A}$. Thus, $|x|, |x'| \leq L$ and so h^∞ invokes h at most η times for hashing either x or x'. The algorithm $\mathcal{B}$ to solve Col(n, m) for h is as follows. $\mathcal{B}$ first executes $\mathcal{A}$. If $\mathcal{A}$ fails, then $\mathcal{B}$ also fails. If $\mathcal{A}$ succeeds and returns (x, x'), then $\mathcal{B}$ invokes h^∞ on both x and x' and "scans backwards" until a collision for h is found. By the above discussion, if (x, x') is a collision for h^∞, then with probability one, the backward scan will produce a collision for h. Thus, the success probability of $\mathcal{B}$ is also ϵ. Further, the number of invocations of h made by $\mathcal{B}$ is found as follows: q times during the execution of $\mathcal{A}$ and at most η times each on x and x', giving a total of at most $q + 2\eta$ invocations. ■

7 Comparison to Iterated Hashing

In this section, we perform a comparison of the new construction to the several variations of the Merkle-Damgård constructions. Before getting into the details, we would like to point a few things.

- Our construction is more general in the sense that it works over an arbitrary DAG, whereas the variations of the Merkle-Damgård algorithm works only with dipaths. Also, we would like to point out that the mechanism in Merkle-Damgård algorithm for handling arbitrary length strings and the associated argument does not carry over to the case of arbitrary DAGs.
- The detailed comparison that we present below is only to Construction II, since this is the algorithm which can hash arbitrary length strings and assumes h to be only collision resistant.
- From a practical point of view, in general, we do not expect our algorithm to replace the Construction III. For most cryptographic purposes, computation of the hash function requires a very small fraction of the total time. Hence, parallel hash computation algorithms (and consequently DAGs) would be required only for special purpose applications. On the other hand, we believe that the issue of obtaining an efficient parallel hash algorithm which can handle arbitrary length strings is of significant theoretical interest.

7.1 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 Construction II. 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 6 *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 Construction II is $2n-m-2+\left\lceil \frac{|x|}{n-m-1} \right\rceil$ (see [7]). 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 bounded above by $O((\log^* |x|)(\log |x|))$. 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.

7.2 Invocation Efficiency

We compare the invocation efficiency of our algorithm to Construction II, i.e., we compare the number of invocations of the compression function h for a message x made by Construction II 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 5 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 $i = j$. Then from Proposition 2, it follows that $|W_i| = (n-m)(k_i - 1) + n$. Now $|W_i| = m+1+|X_i|+|\alpha_i|$ for $0 \leq i \leq j-1$ and $|W_j| = m+|X_j|+|\alpha_j|$, where α_is are the all zero strings which are used as pads to obtain the W_is. Further, $|\alpha_i| \leq n-m-1$ for all $0 \leq i \leq j$. Thus, we obtain $k_i = \left\lceil \frac{|X_i|+1}{(n-m)} \right\rceil$ if $0 \leq i \leq j-1$; and $k_i = \left\lceil \frac{|X_i|}{(n-m)} \right\rceil$ if $i = j$. 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 7 *The total number B of invocations of h made in the computation of h^∞ is equal to*

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

In Construction II, the number of invocations A of the compression function h is equal to $A = 1+\left\lceil \frac{|x|}{n-m-1} \right\rceil$. On the other hand, the number B of invocations of h in our algorithm is given by Proposition 7. 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.

7.3 Optimal Construction?

Consider the problem of secure domain extension to arbitrary length strings. Both Construction II and our algorithm perform this task. We have shown that our algorithm improves upon the Merkle-Damgård algorithm both in terms of reducing the amount of padding and the number of invocations. This suggests the following two problems.

Lower Bound: Let $\mathcal{A}$ be an algorithm which securely extends the domain of a compression function h to arbitrary length strings. What is the minimum amount of padding and minimum number of invocations of h that $\mathcal{A}$ has to make on an input x of length $|x|$?

Construction: Is there a construction which improves upon our algorithm?

At this point, we do not know the answer to either of these two question. In particular, for the first question, we have not even been able to prove that the amount of padding cannot be constant (i.e. independent of the length of x). On the other hand, for the second question, it might seem that a padding of length proportional to $\log |x|$ might be sufficient. However, actually obtaining such a construction along with a correctness proof does not seem to be easy. We believe that the resolution of these questions can form tasks of future research and the answers will be important for the understanding of collision resistant hash functions.

8 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 [2, 4].

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

$$\mathcal{S}_t^{(r,s)} = (\mathcal{I}_t \circ \mathcal{P}_{2^{t-1}}^{(r)}) \circ (\mathcal{K}_s \bullet \mathcal{T}_t). \tag{3}$$

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.

Remark : We note that the basic idea behind Example 7 is already present in Damgård [2]. However, we provide much more details.

8.1 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 $\{1, \ldots, \delta(t)\}$.
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

$$g_t^*(x) = h_{\mathcal{S}_t^{(r,s)}}(x||0^{n-m-\gamma_3}) \tag{4}$$

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 $H_t^* : \cup_{i \geq 1}\{0,1\}^i \rightarrow \{0,1\}^m$ in the following manner. First, for the sake of convenience, we need to define a function $g_0 : \cup_{i=n+1}^{2n-m-1}\{0,1\}^i \rightarrow \{0,1\}^m$ 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 $g_0(x) = h(h(w_1)||w_2)$.

$$\left.\begin{array}{rcll} H_t^*(x) & = & h(x||0^{n-|x|}) & \text{if } |x| \leq n; \\ & = & g_0(x) & \text{if } n < |x| < 2n-m; \\ & = & g_i^*(x) & \text{if } \lambda(i) \leq |x| < \lambda(i+1) \text{ and } 1 \leq i < t; \\ & = & g_t^*(x) & \text{if } |x| \geq \lambda(t). \end{array}\right\} \quad (5)$$

The desired function H_t^∞ is obtained from the function H_t^* using the construction of Section 6.

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 5 and this gives us the following result.

Theorem 8 *Let H_t^∞ be the hash function defined as above. Then, a collision for H_t^∞ yields a collision for h.*

9 Conclusion

We have considered the problem of securely extending the domain of a collision resistant hash function using an arbitrary DAG. A new efficient construction has been presented. This construction improves upon the general Merkle-Damgård algorithm both in the amount of padded bits and the number of invocations of the compression function. The proof of collision resistance of our construction requires the compression function to only collision resistant (one-wayness is not used in the proof). 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 pre-image resistance.

Acknowledgement: The construction in Section 6 was incorrect in an earlier version of the paper. The error was discovered while discussing the paper with several other people. We would like to thank Rana Barua, Mridul Nandi and Bimal Roy for this.

References

[1] M. Bellare and P. Rogaway, Collision-resistant hashing: towards making UOWHFs practical, in: Proceedings of Crypto 1997, Lecture Notes in Computer Science, volume 1294, Springer, 1997, pp. 470-484.

[2] I. B. Damgård, A design principle for hash functions, in: Proceedings of Crypto 1989, Lecture Notes in Computer Science, volume 435, Springer, 1990, pp. 416-427.

[3] W. Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, volume IT-22, number 6, pages 644–654, year 1976.

[4] R. C. Merkle, One way hash functions and DES, in: Proceedings of Crypto 1989, Lecture Notes in Computer Science, volume 435, Springer, 1990, pp. 428-446.

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

[6] D. R. Stinson, Some observations on the theory of cryptographic hash functions, Designs, Codes and Cryptography, to appear.

[7] D. R. Stinson. *Cryptography: Theory and Practice*, CRC Press, second edition, 2002.