Collision-Resistant Hashing: Towards Making UOWHFs Practical

Mihir Bellare[1] and Phillip Rogaway[2]

[1] Dept. of Computer Science & Engineering, University of California at San Diego, 9500 Gilman Drive, La Jolla, California 92093, USA. E-Mail: mihir@cs.ucsd.edu. URL: http://www-cse.ucsd.edu/users/mihir.

[2] Dept. of Computer Science, Engineering II Bldg., University of California at Davis, Davis, CA 95616, USA. E-mail: rogaway@cs.ucdavis.edu. URL: http://wwwcsif.cs.ucdavis.edu/~rogaway.

Abstract. Recent attacks on the cryptographic hash functions MD4 and MD5 make it clear that (strong) collision-resistance is a hard-to-achieve goal. We look towards a weaker notion, the *universal one-way hash functions* (UOWHFs) of Naor and Yung, and investigate their practical potential. The goal is to build UOWHFs not based on number theoretic assumptions, but from the primitives underlying current cryptographic hash functions like MD5 and SHA-1. Pursuing this goal leads us to new questions. The main one is how to extend a compression function to a full-fledged hash function in this new setting. We show that the classic Merkle-Damgård method used in the standard setting fails for these weaker kinds of hash functions, and we present some new methods that work. Our main construction is the "XOR tree." We also consider the problem of input length-variability and present a general solution.

1 Introduction

A cryptographic hash function is a map F which takes a string of arbitrary length and maps it to a string of some fixed-length k. The property usually desired of these functions is *collision-resistance*: it should be "hard" to find distinct strings x and y such that $F(x) = F(y)$.

Cryptographic hash functions are much used, most importantly for digital signatures, and cheap constructions are highly desirable. But in recent years we have seen a spate of attacks bringing down our most popular constructions, MD4 and MD5 [8–11]. The conclusion is that the design of collision-resistant hash functions may be harder than we had thought.

What can we do? One approach is to design new hash functions. This is being done, with SHA-1 [17] and RIPEMD-160 [12] being new designs which are more conservative then their predecessors. In this paper we suggest a complementary approach: weaken the goal, and then make do with hash functions meeting this goal. Ask less of a hash function and it is less likely to disappoint!

Luckily, a suitable weaker notion already exists: *universal one-way hash functions* (UOWHF), as defined by Naor and Yung [16]. But existing constructions, based on general or algebraic assumptions [16,22,13], are not too efficient. We take a different approach. We integrate the notion with current hashing technology, looking to build UOWHFs out of MD5 and SHA-1 type primitives.

The main technical issue we investigate is how to extend the classic Merkle-Damgård paradigm [15,7] to the UOWHF setting. In other words, how to build "full-fledged" UOWHFs out of UOW compression functions. We address practical issues like key sizes and input-length variability. Our main construction, the "XOR tree," also turns out to have applications to reducing key sizes for existing subset-sum based constructions. To make for results more directly meaningful to practice we treat security "concretely," as opposed to asymptotically.

Unfortunately, the name UOWHFs does not reflect the property of the notion, which is a weak form of collision-resistance. We will call our non-asymptotic version *target collision-resistance* (TCR). We refer to the customary notion of collision resistance as *any collision-resistance* (ACR).

1.1 Background

A function $F : D \to \{0,1\}^k$ on some domain D is a *compression function* if D is the set of strings of some small length (eg., $D = \{0,1\}^{640}$ for the compression function of MD5). It is a (full-fledged) hash function if $D = \{0,1\}^*$, or at least $D = \{0,1\}^{\leq \mu}$ for some large number μ. In either case, a *collision* for F is a pair $x, y \in D$ such that $x \neq y$ but $F(x) = F(y)$. Informally, F is said to be *any collision-resistant* (ACR) if it is computationally hard to find a collision.

THE MD METHOD. The Merkle-Damgård construction [15,7] turns an ACR compression function F: $\{0,1\}^{k+b} \to \{0,1\}^k$ into an ACR hash function MDF. Fix $C_0 \in \{0,1\}^k$. Given $M = M[1] \ldots M[n]$, for $M[i] \in \{0,1\}^b$, compute $C_i = F(C_{i-1} \| M[i])$ and set $MDF(M) = C_n$. The property of this method is that if it is hard to find collisions in F, then it is hard to find collisions in MDF.

All of the popular hash functions (MD4, MD5, SHA-1, and RIPEMD-160) use the MD construction. Thus the crucial component of each algorithm is the underlying compression function, and we want it to be ACR. But the compression function of MD4 is not: following den Boer and Bosselaers [8], collisions were found by Dobbertin [10]. Then collisions were found for the compression function of MD5 [9,11]. These attacks are enough to give up on MD4 and MD5 from the point of view of ACR. No collisions have been found for the compression functions of SHA-1 and RIPEMD-160, and these may well be stronger.

KEYING. In the above popular hash functions there is no explicit key. But Damgård [6,7] defines ACR via keyed functions, and it is in this setting that he proves the MD construction correct [7]. Keying hash functions seems essential for a meaningful formalization of security. Thus, in truth, a hash function F does not have the signature above: it must have two arguments, one for the key K and one for the message M. To use F one selects a random key K, publishes it, and from then on one hashes according to F_K. In effect, the key amounts to the description of a particular hash function.

1.2 Target Collision-Resistance

With an ACR hash function F the key K is published and the adversary wins if she manages to find *any* collision x, y for the F_K. The points x and y may depend arbitrarily on K; any pair of distinct points will do. In the notion of Naor and Yung [16] the adversary no longer wins by finding just any collision. The adversary must choose one point, say x, in a way which does *not* depend on K, and then, later, given K, find another point y, this time allowed to depend on K, such that x, y is a collision for F_K. Thus, although it might be easy to find a collision x, y in F_K by making both x, y depend on K, the adversary may be unable to find collisions if she must "commit" to x before seeing K. We call this weakened notion of security *target collision-resistance* (TCR). (In the terminology of [16] it is universal one-wayness.)

Naor and Yung [16] formalize this via the standard "polynomial-time adversaries achieve negligible success probability" approach of asymptotic cryptography. In order to get results which are more directly meaningful for practice, our formalization is non-asymptotic. See Section 2.

NO BIRTHDAYS! Besides being a weaker notion (and hence easier to achieve) we wish to stress one important practical advantage of TCR over ACR: because x must be specified before K is known, birthday attacks to find collisions are not possible. This means the hash length k can be small, like 64 or 80 bits, as compared to 128 or 160 bits for an ACR hash function. This is important to us for several reasons and we will appeal to it later.

GOOD ENOUGH FOR SIGNING. In weakening the security requirement on hash functions we might risk reducing their utility. But TCR is strong enough for the major applications, *if appropriately used.* In particular, it is possible to use TCR hash functions for hashing a message before signing. Due to page limits, the constructions are omitted here: they can be found in [4]. The idea is to pick a new key K for each message M and then sign the pair $(K, F_K(M))$, where F is TCR. This works best for short keys. When they are long some extra tricks can be used, as described in [4], but we are better off with small keys. Thus there is a strong motivation for keeping keys short.

1.3 Making TCR Functions out of Standard Hash Functions

The most direct way to make a TCR hash function is to appropriately key an existing hash function such as MD5 or SHA-1. We caution that one must be careful in how this keying is done. If not, making a TCR assumption about the keyed function may really be no weaker than making an ACR assumption about the original hash function. See Section 4.

1.4 Extending TCR Compression Functions to TCR Hash Functions

Instead of trying to key an entire hash function, as above, a good strategy might be to implement a TCR compression function (perhaps by keying an existing compression function) and then transform it into a (full-fledged) TCR hash function using some simple construction. The question we investigate is how to do this transformation. This turns out to be quite interesting.

THE MD METHOD DOES NOT WORK FOR TCR. Suppose we are given a TCR compression function H in which each κ-bit key specifies a map H_K: $\{0,1\}^{k+b} \rightarrow \{0,1\}^k$. We want to build a TCR hash function H' in which each key K specifies a map H'_K on arbitrary strings. The obvious thought is to apply the MD method to H_K. However, we show in Section 5.1 that this does not work. We give an example of a compression functions which is secure in the TCR sense but for which the resulting hash function is not.

LINEAR ITERATION: BASIC AND XOR. The most direct extension we found to the MD construction is use a different key at each stage. This works, and its exact security is analyzed in Section 5.2. But the method needs a long key.

We provide a variant of the above scheme which uses only one key for the compression function, but also uses a number of auxiliary keys, which are XORed in at the various stages. This can slightly reduce key sizes, and it also has some advantages from a key-scheduling point of view (eg., it may be slow to "set up" the key of a compression function, so it's best if this not be changed too often).

THE BASIC TREE SCHEME. To get major reductions in key size we turn to trees.[1] Wegman and Carter [25] give a tree-based construction of universal hash functions that reduces key sizes, and Naor and Yung have already pointed out that key lengths for UOWHFs can be reduced by the same method [16, Section 2.3]. We recall this basic tree construction in Section 5.4 and provide a concrete analysis of its security. Then we look at key sizes. Suppose we start with a compression function with key length κ, mapping $2k$ bits to k bits, and we want to hash nk bits to k bits. The basic tree construction yields a hash function with a key size of $\kappa \lg n$ bits.[2] Key lengths have been reduced, but one can reduce them more.

THE XOR TREE SCHEME. Our main construction is the XOR tree scheme. Here, the hash function uses only one key for the compression function and some auxiliary keys. If we start with a compression function with key length κ, mapping $2k$ bits to k bits, and we want to hash nk bits to k bits, the XOR tree construction yields a hash function with a key size of $\kappa + 2k \lg n$ bits.

Recall that k is short, like 64 bits, since we do not need to worry about birthday attacks for TCR functions. On the other hand, κ can be quite large (and in many constructions, it is). So $\kappa + 2k \lg n$ may be much less than $\kappa \lg n$.

[1] It may be worth remarking that the obvious idea for reducing key size is to let the key be a seed to a pseudorandom number generator and specify longer keys by stretching the seed to any desired length. The problem is that our keys are public (they are available to the adversary) and pseudorandom generators are of no apparent use in such a context.

[2] This corresponds to a binary tree construction. In Section 5.4 we consider the more general case of starting with a compression function of mk bits to k bits and building an m-ary tree.

1.5 Other Results

REDUCING KEY SIZES FOR OTHER CONSTRUCTIONS. Our main motivation has been building TCR hash functions from primitives underlying popular cryptographic hash functions. But XOR trees can also be used to reduce key sizes for TCR hash functions built from combinatorial or algebraic primitives. For example, the subset sum based construction of [13] uses a key of size sl bits to hash s bits to l bits, where l is a security parameter which controls subset sum instance sizes. (Think of l as a few hundred.) So the size of the key is even longer than the size of the data. The basic (binary) tree scheme can be applied to reduce this: starting with a compression function taking $2l$ bits to l bits (it has key length $\kappa = 2l^2$) the key size of the resulting hash function is $\kappa \lg n = 2l^2 \lg(n)$, where n is the number of (l-bit) blocks hashed. With our (binary) XOR tree scheme, the key size of the resulting function is $\kappa + 2l \lg(n) = 2l(l + \lg(n))$. The latter can be quite a bit smaller. For example if $l = 300$ and $n = 10$ KBytes, the key length for the basic scheme is about 15 times larger than that for the XOR scheme.

INPUT-LENGTH VARIABILITY. The proofs for our constructions rule out adversaries who can find collisions x, y for equal-length strings. In practice, collisions between strings of unequal length must be prevented. To handle this we provide a general mechanism for turning hash functions secure against equal-length collisions into hash functions secure also against collisions of possibly unequal length. This needs just one extra application of the compression function. See Section 6. Given this, we can concentrate on designing functions that resist equal-length collisions.

1.6 Related Work

The general approach to concrete, quantitative security that we are following began with [3].

Keying hash functions has arisen in other contexts. A good deal of work has gone into keying hash functions for message authentication [1,23,14,18]. But that is a very different problem from what we look at here; there, the key is secret, and one is trying to achieve a particular goal of private key cryptography. In another direction for keying a hash function Bellare, Canetti and Krawczyk [2] considered keyed compression functions as pseudorandom functions, and showed that applying the MD construction then yields a pseudorandom function.

A weaker-than-standard notion of hashing is considered in [1], but that notion is based on a hidden key and those hash functions don't suffice for signatures.

2 Notions of Hashing

When one looks at hash functions like MD5 or SHA-1 there is no explicit key. However, no notion of collision-freeness can be properly formalized in such a setting. The first step is thus to talk of families of functions.

FAMILIES OF HASH FUNCTIONS. In a family of hash functions F, each key K specifies a particular hash function F_K in the family. Each such function maps D to $\{0,1\}^k$ where D is some domain associated to the family, and k is the *hash length*, or *output length*, also depending on the family. The key K will be drawn at random from some key space $\{0,1\}^\kappa$, and κ will be called the *key length*. If $D = \{0,1\}^l$ for some l then l is called the *input length*.

Formally, a family F of (keyed) hash functions is a map F: $\{0,1\}^\kappa \times D \to \{0,1\}^k$. We define F_K: $D \to \{0,1\}^k$ by $F_K(x) = F(K,x)$ for each $K \in \{0,1\}^\kappa$ and each $x \in D$. We use either the notation $F_K(x)$ or $F(K,x)$, as convenient.

The hash family F is a *compression function* if the domain is $D = \{0,1\}^m$ for some (small) constant m (eg., $m = 512$). It is an *extended hash function* if $D = \{0,1\}^M$ or $\{0,1\}^{\leq M}$ for some (large) constant M (eg., $M = 2^{64} - 1$).

We say F is *length consistent* if for every $K \in \{0,1\}^\kappa$ and every $x, y \in D$ it is the case that if $|x| = |y|$ then $|F(K,x)| = |F(K,y)|$.

COLLISIONS. A *collision* for a function f defined on a domain D is a pair of strings $x, y \in D$ such that $x \neq y$ but $f(x) = f(y)$. In our setting the function of interest is $f = F_K$ for a randomly chosen key K. Security of a hash family talks about the difficulty of finding collisions in F_K. There are two notions of security. The stronger one we call here *any collision-resistance* (ACR). The weaker one, which is due to Naor and Yung [16], we call *target collision-resistance* (TCR). We now define both notions. First some technicalities.

PROGRAMS AND TIMING. A model of computation is fixed, and the execution time of a program is discussed and analyzed as in any algorithms text, eg. [5]. An *adversary* is a program for our model, written in some fixed programming language. Any program is allowed randomness: the programming language supports the operation FLIPCOIN() which returns a random bit. By convention, when we speak of the running time of an adversary we mean the actual execution time in the fixed model of computation plus the length of the description of the program. For F a family of hash functions we let T_F denote the worst-case time to compute F in the underlying model of computation. Namely, given $K \in \{0,1\}^\kappa$ and $x \in D$, this is the time to output $F(K,x)$.

2.1 Any Collision-Resistance — ACR

The "standard" notion of collision resistance for a function f is that given f it is hard to find a collision x, y for f. In the keyed setting, it can be formalized like this (eg. [6,7]). The adversary C, called a *collision-finder*, is given K chosen at random from $\{0,1\}^\kappa$, and is said to *succeed* if it outputs a collision x, y for F_K. We measure the quality of the hash family by the success probability of the adversary as a function of the time it invests. Formally, a collision-finder C is said to (t,ϵ)-break the family of hash functions $F : \{0,1\}^\kappa \times D \to \{0,1\}^k$ if the running time of the adversary is at most t and the probability that C, on input K, outputs a collision x, y for F_K is at least ϵ. Here the probability is take over K (a random point in $\{0,1\}^\kappa$) and C's random coins. Informally we say F is "any collision-resistant" (ACR) if for every collision-finder who (t, ϵ)-breaks F, the ratio t/ϵ is large.

Note that the adversary is given the (random) point K (the key is "announced") and only then is the adversary asked to find a collisions for F_K. So the adversary may employ a strategy in which the collision which is found depends on K. This makes the notion very strong.

2.2 Target Collision-Resistance — TCR

In the notion of [16] the adversary does not get credit for finding any old collision. The adversary must still find a collision x, y, but now x is not allowed to depend on the key: the adversary must choose it before the key K is known. Only after "committing" to x does the adversary get K. Then she must find y.

Formally, the adversary $C = (C_1, C_2)$ (called a target collision finder) consists of two algorithms, C_1 and C_2. First, C_1 is run, to produce x and possibly some extra "state information" σ that C_1 wants to pass to C_2. We call x the *target message.* Now, a random key K is chosen and C_2 is run. Algorithm C_2 is given K, x, σ and must find y different from x such that $F_K(x) = F_K(y)$.

The formalization of [16] was asymptotic. Here we provide a concrete one, and call this version of the notion target collision-resistance (TCR).

We begin with some special cases. A target collision finder $C = (C_1, C_2)$ is called an equal-length target collision finder if the messages x, y which C_1 outputs always satisfy $|x| = |y|$. It is called a variable-length target collision finder when no restriction is made on the relative lengths of x, y.

Let $C = (C_1, C_2)$ be a target-collision finder. We say it (t, ϵ)-breaks F if its running time is at most t and it finds a collision with probability at least ϵ. The running time is the sum of the running times of the two algorithms and the probability is over the coins of C_1 and C_2 and the choice of K. We say that F is (t, ϵ)-resistant to equal-length target collisions if there is no equal-length target collision finder which (t, ϵ)-breaks F. F is (t, ϵ)-resistant to variable-length target collisions if there is no variable-length target collision finder which (t, ϵ)-breaks F. If we just say F is (t, ϵ)-TCR, or (t, ϵ)-resistant to target collisions, we mean it is (t, ϵ)-resistant to variable-length target collisions.

Resistance to equal-length target collisions is a weaker notion than resistance to variable-length target collisions: in the former, the adversary is only being given credit if it finds collisions where the messages are of the same length. In practice, we want resistance to variable-length target resistance. However, it turns out the convenient design approach is to focus on resistance to equal-length target collisions and then achieve resistance to variable-length target collisions via a general transformations we present in Section 6.

Informally, we say F is "target collision-resistant" (TCR) (or, resp., TCR to equal-length collisions) if it for every (resp., equal-length) target collision-finder who (t, e)-breaks F, the ratio t/ϵ is large.

3 Composition Lemmas

It is useful to hash a long string in stages, first cutting down its length via one hash function, then applying another to this output to cut it down further. Naor and Yung [16] considered this kind of composition in the context of TCR hash functions. We first state a concrete version of their lemma and then extend it to an equal-length collision analogue which is in fact what we will use.

Let H_1: $\{0,1\}^{\kappa_1} \times \{0,1\}^{l_1} \to \{0,1\}^{l_2}$ and H_2: $\{0,1\}^{\kappa_2} \times \{0,1\}^{l_2} \to \{0,1\}^k$ be families of hash functions. The *composition* $H_2 \circ H_1$: $\{0,1\}^{\kappa_1+\kappa_2} \times \{0,1\}^{l_1} \to \{0,1\}^k$ is the family defined by $(H_2 \circ H_1)(K_2K_1, M) = H_2(K_2, H_1(K_1, M))$ for all $K_1 \in \{0,1\}^{\kappa_1}$, $K_2 \in \{0,1\}^{\kappa_2}$, and $M \in \{0,1\}^{l_1}$. From the proof of Naor and Yung's composition lemma [16] we extract the concrete security parameters to get the following.

Lemma 1. (TCR composition lemma) *Let H_1: $\{0,1\}^{\kappa_1} \times \{0,1\}^{l_1} \to \{0,1\}^{l_2}$ and H_2: $\{0,1\}^{\kappa_2} \times \{0,1\}^{l_2} \to \{0,1\}^k$ be families of hash functions. Assume the first is (t_1, ϵ_1)-secure against target collisions and the second is (t_2, ϵ_2)-secure against target collisions. Then the composition $H_2 \circ H_1$ is (t, ϵ)-secure against target collisions, where $t = \Theta(\min(t_1 - k_2, t_2 - 2T_{H_1} - k_1))$ and $\epsilon = \epsilon_1 + \epsilon_2$.*

In this paper we need such a lemma for the case of equal-length TCR. This requires an extra condition on the first family of hash functions, namely that it be length consistent. See [4] for a proof.

Lemma 2. (TCR composition lemma for equal-length collisions) *Let H_1: $\{0,1\}^{\kappa_1} \times \{0,1\}^{l_1} \to \{0,1\}^{l_2}$ and H_2: $\{0,1\}^{\kappa_2} \times \{0,1\}^{l_2} \to \{0,1\}^k$ be families of hash functions. Assume the first is length consistent and (t_1, ϵ_1)-resistant to equal-length target collisions. Assume the second is (t_2, ϵ_2)-resistant to equal-length target collisions. Then the composition $H_2 \circ H_1$ is (t, ϵ)-resistant to equal-length target collisions, where $t = \Theta(\min(t_1 - k_2, t_2 - 2T_{H_1} - k_1))$ and $\epsilon = \epsilon_1 + \epsilon_2$.*

4 TCR Hash Functions from Standard Hash Functions

The most direct way to construct a TCR hash function is to key a function like MD5 or SHA-1. We point out the importance of doing this keying with care.

Suppose, for example, that one keys MD5 through its 128-bit initial chaining value, IV. Denote the resulting hash function family by MD5*. Then breaking MD5* (in the sense of violating TCR) amounts to finding collisions in an algorithm which is identical to MD5 except that it begins with a random, known IV (as opposed to the published one). It seems unlikely that this task would be harder than finding collisions in MD5 itself. It could even be easier!

Alternatively, suppose one tries to use the well-known "envelope" method, setting $\mathrm{MD5}^{**}_K(M) = \mathrm{MD5}(K\|M\|K)$. It seems likely that any extension of Dobbertin's attack [11] which finds collisions in MD5 would also defeat MD5**. Letting md5 denote the compression function of MD5, note that if for any c you can find m, m' such that $\mathrm{md5}(c\|m) = \mathrm{md5}(c\|m')$, then you have broken MD5**.

A safer approach might be to incorporate key bits throughout the message being hashed. For example, with $|K| = 128$ one might intertwine 128 bits of key and the next 384 bits of message into every 512-bit block. (For example, every fourth byte might consist of key.) Now the cryptanalyst's job amounts to finding a collision M, M' in MD5 where we have pre-specified a large number of (random) values to be sprinkled in particular places throughout M and M'. This would seem to be very hard.

Note that the approach above (shuffling key and message bits) is equally at home in defining a TCR compression function based on the compression function underlying a map like MD5 or SHA-1. The resulting compression keyed compression function can then be extended to a full-fledged keyed hash function using the constructs of this paper. Doing this one will gain in provable-security but lose out in increased key length.

5 TCR Hashing based on TCR Compression Functions

Throughout this section messages will be viewed as sequences of blocks of some length l, with l depending on the context. For notational simplicity, let $\Sigma_l = \{0,1\}^l$.

We are given a TCR compression function H. We wish to extend it to a full-fledged hash function H'. We begin with a method which does *not* work.

5.1 The MD Construction Doesn't Propagate TCR

Suppose H: $\{0,1\}^\kappa \times \{0,1\}^{k+b} \rightarrow \{0,1\}^k$ and we want to hash messages of nb-bits. The MD method gives a keyed family of functions MDH^n: $\{0,1\}^\kappa \times \{0,1\}^{nb} \rightarrow \{0,1\}^k$. To define it, fix some k-bit initial vector, say $\mathrm{IV} = 0^k$. Now view the message $M = M[1] \ldots M[n]$ as divided into n blocks, each of b bits. Let $MDH^n(K, M) = C_n$ where $C_0 = \mathrm{IV}$ and, for $i \geq 1$, $C_i = H_K(C_{i-1} \parallel M[i])$.

Damgård [7] shows that if H is ACR then so is MDH^n. It would be nice if this worked for TCR too. But it does not. The reason is a little subtle. If H is TCR it still might be easy to find collisions in H_K if we knew K in advance (meaning we were allowed to see K before specifying any point for the collision). However, a few MD iterations of H on a fixed point can effectively surface the key K, causing subsequent iterations to misbehave. This intuition can be formalized by giving an example of a compression function H which is TCR but for which MDH^n is not. To give such an example we must first assume that some TCR compression function exists (else the question is moot).

Proposition 3. *Suppose there exists a* TCR *compression function* F: $\{0,1\}^\kappa \times \{0,1\}^{k+b} \rightarrow \{0,1\}^k$. *Then there exists a* TCR *compression function* H *and an integer* $n \leq \max(2, \lceil (k + \kappa + b)/b \rceil)$ *such that* MDH^n *is* not TCR.

The statement of the above proposition is "informal" insofar as we have pushed the numerical bounds into the proof.

5.2 The Basic Linear Hash

Though the MD construction doesn't propagate TCR, a natural approach is to iterate just in MDH^n but with a different key at each round. We will show that this does preserve TCR.

We let H: $\{0,1\}^\kappa \times \{0,1\}^{k+b} \rightarrow \{0,1\}^k$ be the given TCR compression function. The message $M = M[1] \ldots M[s]$ to be hashed is viewed as divided into $s \leq n$ blocks, each of size b.

In the basic linear scheme a hash function is specified by n different keys, $K_1, \ldots, K_n \in \{0,1\}^\kappa$, one key for each application of the underlying compression function. We hash as in the MD method but with a different key at each stage. Again we set the IV to a constant, say 0^k. Formally, $LH^n(K_1 \cdots K_n, M) = C_s$ where $C_0 = 0^k$ and for $i = 1, \ldots, s$: $C_i = H(K_i, C_{i-1} \parallel M[i])$. The family of hash functions is LH^n: $\{0,1\}^{n\kappa} \times \Sigma_b^{\leq n} \rightarrow \{0,1\}^k$. The following theorem says that if the compression function H is resistant to target collisions then so is the extended hash function LH^n. The proof is in [4].

Theorem 4. *Suppose H: $\{0,1\}^\kappa \times \{0,1\}^{k+b} \rightarrow \{0,1\}^k$ is (t', ϵ')-resistant to target collisions. Suppose $n \geq 1$. Then LH^n: $\{0,1\}^{n\kappa} \times \Sigma_b^{\leq n} \rightarrow \{0,1\}^k$ is (t, ϵ)-resistant to equal-length target collisions, where $\epsilon = n\epsilon'$ and $t = t' - \Theta(n \cdot [T_H + \kappa])$.*

5.3 The XOR Linear Hash

We present a variant of the above in which the compression function uses the same key K in each stage, but an auxiliary "mask" key K_i, depending on the stage number i, is XORed to the chaining variable in the i-th stage. One advantage is that the key size is reduced compared to the basic scheme for some choices of the parameters. Another advantage is in key scheduling. If the compression function is being computed in hardware it may be preferable to fix the key for the compression function. In software too there can be a penalty for key "setup."

We now describe the scheme properly. Recall the message is $M = M[1] \ldots M[s]$ where $s \leq n$. The function is specified by $n+1$ different keys $K, K_1, \ldots, K_n$ as indicated above. Let $XLH^n(KK_1 \ldots K_n,\ M) = C_s$ where $C_0 = 0^k$ and for $i = 1, \ldots, s$ we set

$$C'_{i-1} = K_i \oplus C_{i-1} \quad \text{and} \quad C_i = H(K,\ C'_{i-1} \parallel M[i]) .$$

The corresponding family is XLH^n: $\{0,1\}^{\kappa+nk} \times \Sigma_b^{\leq n} \rightarrow \{0,1\}^k$. The proof of the following can be found in [4].

Theorem 5. *Suppose H: $\{0,1\}^\kappa \times \{0,1\}^{k+b} \rightarrow \{0,1\}^k$ is (t', ϵ')-resistant to target collisions. Suppose $n \geq 1$. Then XLH^n: $\{0,1\}^{\kappa+nk} \times \Sigma_b^{\leq n} \rightarrow \{0,1\}^k$ is (t, ϵ)-resistant to equal-length target collisions, where $\epsilon = n\epsilon'$ and $t = t' - \Theta(n \cdot [T_H + \kappa])$.*

5.4 The Basic Tree Scheme

A tree can be used to reduce the key size. A tree scheme described by Naor and Yung [16] uses $\lg n$ keys (for the compression function) to hash an n-block message (using a binary tree). Later we will do better, but first let us provide a description and concrete security analysis of the basic scheme.

We are slightly more general. First, we consider m-ary trees for some $m \geq 2$. This means we start with a compression function H: $\{0,1\}^\kappa \times \{0,1\}^{mk} \to \{0,1\}^k$ and want to use it to hash messages of length much more than mk. Second, we do not wish to fix the number of blocks that messages will be.

PARALLEL HASH. We first describe a primitive we will use. We are given a message M which is either a single k-bit block, or else consists of s blocks, each mk bits. In the former case we leave the message unchanged. In the latter case view the message as $M = M[1] \ldots M[s]$ and hash each mk-bit block into a k-bit block via the compression function. We call this procedure "parallel hash."

Formally, for any integer $l \geq 1$ we define PH^{lm}: $\{0,1\}^\kappa \times (\Sigma_k \cup \Sigma_{mk}^{\leq l}) \to \Sigma_k^{\leq l}$ like this. If $M \in \Sigma_k$ set $PH^{lm}(K, M) = M$. Else write $M = M[1] \ldots M[s] \in \Sigma_{mk}^s$ $(s \leq l)$ and set

$$PH^{lm}(K,M) \;=\; H(K,M[1]) \,\|\, H(K,M[2]) \,\|\, \cdots \,\|\, H(K,M[s]) \,.$$

Notice that only one key is used. Notice that $PH^m = H$ is the original compression function augmented to be the identity on messages of size k.

Lemma 6. *Suppose H: $\{0,1\}^\kappa \times \{0,1\}^{mk} \to \{0,1\}^k$ is (t', ϵ')-resistant to target collisions. Suppose $l \geq 1$ and let $n = lm$. Then PH^n: $\{0,1\}^\kappa \times \Sigma_k \cup \Sigma_{mk}^{\leq l} \to \Sigma_k^{\leq l}$ is (t, ϵ)-resistant to equal-length target collisions, where $\epsilon = l\epsilon'$ and $t = t' - \Theta(nk)$.*

Proof. The proof is a slight extension and "concretization" of the proof sketch in [16, Section 2.3], and for completeness is given in [4]. ▮

BASIC TREE HASH. We assume that the messages we will hash have length bounded by km^d, for some constants m and d. To simplify our exposition we further insist that any message M to be hashed has exactly m^i k-bit blocks, for some $i \in \{0, 1, \ldots, d\}$. The hash function uses d keys $K_1, \ldots, K_d$, and the hash value is denoted $TH(K_1 \ldots K_d, M)$. It is computed on $M = M[1] \cdots M[m^i]$ by building an m-ary tree of depth i. The leaves correspond to the message blocks and the root corresponds to the final hash value. Group the nodes at level 1 (the leaves) into runs of size m and hash each group via $H(K_1, \cdot)$. This yields m^{i-1} values, which form the nodes at level 2 of the tree. Now continue the process. At each level we use a different key. Thus $H(K_j, \cdot)$ is the function used to hash the nodes at level j of the tree. At level i we have m nodes, which are hashed under $H(K_i, \cdot)$ to yield the root, which, at level $i+1$, is the final hash value.

Formally, the hash function can be defined by compositions of the parallel hashes we described above. Namely

$$TH^{m^d} \;=\; PH^m \circ PH^{m^2} \circ \ldots \circ PH^{m^{d-1}} \circ PH^{m^d} \,. \tag{1}$$

By our notion of composition each parallel hash uses a different key. Thus one key is used for every hash of a given level, but the key changes across levels. So there are d keys in all. We can assess the security by applying the composition lemma and the analysis of the security of the parallel hash.

Theorem 7. *Suppose H: $\{0,1\}^\kappa \times \{0,1\}^{mk} \to \{0,1\}^k$ is (t', ϵ')-resistant to target collisions. Suppose $n = m^d$ where $d \geq 1$. Then TH^n: $\{0,1\}^{d\kappa} \times \bigcup_{i=0}^{d} \Sigma_k^{m^i} \to \{0,1\}^k$ is (t, ϵ)-resistant to equal-length target collisions, where $\epsilon = n\epsilon'/(m-1)$ and $t = t' - \Theta(n \cdot [T_H + \kappa])$.*

Proof. For each $i = 1, \ldots, d$, Lemma 6 says that PH^{m^i} is (t_i, ϵ_i)-secure against equal-length collisions, where $t_i = t - \Theta(m^i)$ and $\epsilon_i = m^{i-1}\epsilon'$. Note that PH^{m^i} is length consistent for any $i = 1, \ldots, d$. Now look at Equation (1) and apply Lemma 2 d times. This gives us $\epsilon = (m^0 + \ldots + m^{d-1})\epsilon' = (m^d - 1)\epsilon'/(m-1) \geq n\epsilon'/(m-1)$. The time estimates can be checked similarly. Details omitted. ▌

5.5 The XOR Tree Hash

In the basic tree hash we key the compression function anew at each level of the tree. Thus the key length is $\kappa \cdot \log_m(n)$, which can be large, because κ may be large. In the XOR variant there is one key K defining $H(K, \cdot)$ and this is used at all levels. However, there are auxiliary keys $K_1, \ldots, K_d$, one per level. These are not keys for the compression function: they are just XORed to the data at each stage. As described in Section 5.2, the motivation is that we can get shorter keys, and also better key scheduling.

We now describe the hash function. As before, assume that the messages we will hash have m^i blocks, each k-bits, for some $i \in \{0, \ldots, d\}$. The hash function is described by a "primary key" $K \in \{0,1\}^\kappa$ and d "auxiliary keys" $K_1, \ldots, K_d \in \{0,1\}^{mk}$. The hash value is denoted $XTH(KK_1 \ldots K_d, M)$. It is computed by building an m-ary tree of depth i whose root is the hash value. Level 1 consists of the leaves, which are the m^d individual blocks of the message. Group them into runs of m blocks each. Before hashing each sequence of blocks, however, XOR key K_1 to each group. Now hash each masked group via $H(K, \cdot)$. This yields m^{d-1} values, which form the nodes at level 2 of the tree. Now continue the process. At each level we use a different key for the masking, but the same key K for the hashing.

Here is a more formal description. Let $M = M[1] \ldots M[s]$ be the message, consisting of k-bit blocks, and suppose $s = m^i$. We define $XTH(KK_1 \cdots K_d, M)$ recursively. First assume $i = 0$ so that $M = M[1]$ consists of 1 block. We set $XTH(K, M) = M[1]$. Now suppose the message M is longer, namely $i \geq 1$. Let $N_j = M[(j-1)m^{i-1} + 1] \ldots M[jm^{i-1}]$ for $j = 1, \ldots, m$. Let

$$XTHI(KK_1 \ldots K_{i-1}, M) = \\ XTH(KK_1 \ldots K_{i-1}, N_1) \,\|\, \cdots \,\|\, XTH(KK_1 \cdots K_{i-1}, N_m) .$$

Then let

$$XTH(KK_1 \ldots K_i, M) = H(K, K_i \oplus XTHI(KK_1 \ldots K_{i-1}, M)) .$$

The first function here computes the input for the next compression stage. Before applying the compression function to it, however, we XOR in the auxiliary key for this stage. We also set $XTH(KK_1 \ldots K_d, M) = XTH(KK_1 \ldots K_i, M)$. We let XTH^n: $\{0,1\}^{\kappa+dmk} \times \bigcup_{i=0}^{d} \Sigma_k^{m^i} \to \{0,1\}^k$ denote the XOR tree hash family for messages of at most $n = m^d$ blocks.

The key length is $\kappa + mk \cdot \log_m(n)$. For $m = 2$, $k = 64$, the resulting key length of $\kappa + 128 \lg n$ is significantly smaller than for the basic tree scheme in the case where the key size of the compression function is quite big, as happens for examples in the constructions of [13].

We can no longer appeal to the composition lemma in proving security, because the hashing at the different levels of the tree involve a common key K. Instead we give a direct proof of security, which can be found in [4].

Theorem 8. *Suppose* H: $\{0,1\}^\kappa \times \{0,1\}^{mk} \to \{0,1\}^k$ *is* (t', ϵ')*-resistant to target collisions. Suppose* $n = m^d$ *where* $d \geq 1$. *Then* XTH^n: $\{0,1\}^{\kappa+dmk} \times \bigcup_{i=0}^{d} \Sigma_k^{m^i} \to \{0,1\}^k$ *is* (t, ϵ)*-resistant to equal-length target collisions, where* $\epsilon = n\epsilon'/(m-1)$ *and* $t = t' - \Theta(n \cdot [T_H + mk])$.

6 Length Variability

In Section 5 we proved security against equal-length target collisions. In practice one requires security against variable-length target collisions.

It is often assumed that input-length variability can be handled by padding the final block of a message M to be hashed so that it unambiguously encodes $|M|$. Let $pad(\cdot)$ denote such a padding function (eg., that of [20]). If H is secure against equal-length target collisions is is $H \circ pad$ secure against variable-length target collisions? Not necessarily. And the same applies to ACR. It is easy to construct such examples.

Here, instead, is a general technique to achieve length-variability. It requires one extra application of the compression function. See [4] for a proof.

Theorem 9. *Fix* $m > 0$ *and let* D_1 *be a set of strings each of length less than* 2^m. *Let* H_1: $\{0,1\}^{\kappa_1} \times D_1 \to \{0,1\}^{l_1}$ *and* H_2: $\{0,1\}^{\kappa_2} \times \{0,1\}^{l_1+m} \to \{0,1\}^k$ *be families of hash functions. Assume* H_1 *is* (t_1, ϵ_1)*-secure against equal-length target collisions and* H_2 *is* (t_2, ϵ_2)*-secure against equal-length target collisions. Define* H: $\{0,1\}^{\kappa_1+\kappa_2} \times D_1 \to \{0,1\}^k$ *by*

$$H(K_1K_2, M) \;=\; H_2(K_2, H_1(K_1, M) \,\|\, |M|_m)$$

where $|M|_m$ *is the length of* M *written as a string of exactly* m *bits,* $M \in D_1$, $K_1 \in \{0,1\}^{\kappa_1}$, *and* $K_2 \in \{0,1\}^{\kappa_2}$. *Then* H *is* (t, ϵ)*-secure against variable-length target collisions, where* $t = \min(t_1 - \Theta(\kappa_2),\ t_2 - \Theta(\kappa_1 - 2T_{H_1} - 2l_1 - 2m))$ *and* $\epsilon = \epsilon_1 + \epsilon_2$.

While length-indicating padding doesn't work in general, does it work for the schemes of Section 5? For LH the answer is *no*: starting with an arbitrary TCR compression function H_0 one can construct a TCR compression function H for which $LH \circ pad$ is insecure against variable-length target collisions. For XLH the answer is *yes*: if H is a TCR compression function then $XLH \circ pad$ is guaranteed to be secure against target collisions; one can appropriately modify the proof of Theorem 5 to show this.

Acknowledgments

The first author is supported in part by a 1996 Packard Foundation Fellowship in Science and Engineering, and NSF CAREER award CCR-9624439. The second author is supported in part by NSF CAREER Award CCR-9624560.

Thanks to the Crypto 97 referees for their comments on this paper.

References

1. M. Bellare, R. Canetti and H. Krawczyk, Keying hash functions for message authentication. *Advances in Cryptology – Crypto* 96 *Proceedings*, Lecture Notes in Computer Science Vol. 1109, N. Koblitz ed., Springer-Verlag, 1996.
2. M. Bellare, R. Canetti and H. Krawczyk, Pseudorandom functions revisited: the cascade construction and its concrete security. *Proceedings of the* 37th *Symposium on Foundations of Computer Science*, IEEE, 1996.
3. M. Bellare, J. Kilian and P. Rogaway, The security of cipher block chaining. *Advances in Cryptology – Crypto* 94 *Proceedings*, Lecture Notes in Computer Science Vol. 839, Y. Desmedt ed., Springer-Verlag, 1994.
4. M. Bellare and P. Rogaway, Collision-Resistant Hashing: Towards Making UOWHFs Practical. Full version of this paper, available via `http://www-cse.ucsd.edu/users/mihir`.
5. T. Cormen, C. Leiserson and R. Rivest, Introduction to Algorithms. McGraw-Hill, 1992.
6. I. Damgård, Collision Free Hash Functions and Public Key Signature Schemes. *Advances in Cryptology – Eurocrypt* 87 *Proceedings*, Lecture Notes in Computer Science Vol. 304, D. Chaum ed., Springer-Verlag, 1987.
7. I. Damgård, A Design Principle for Hash Functions. *Advances in Cryptology – Crypto* 89 *Proceedings*, Lecture Notes in Computer Science Vol. 435, G. Brassard ed., Springer-Verlag, 1989.
8. B. den Boer and A. Bosselaers, An attack on the last two rounds of MD4. *Advances in Cryptology – Crypto* 91 *Proceedings*, Lecture Notes in Computer Science Vol. 576, J. Feigenbaum ed., Springer-Verlag, 1991.
9. B. den Boer and A. Bosselaers, Collisions for the compression function of MD5. *Advances in Cryptology – Eurocrypt* 93 *Proceedings*, Lecture Notes in Computer Science Vol. 765, T. Helleseth ed., Springer-Verlag, 1993.
10. H. Dobbertin, Cryptanalysis of MD4. *Fast Software Encryption—Cambridge Workshop*, Lecture Notes in Computer Science, vol. 1039, D. Gollman, ed., Springer-Verlag, 1996.
11. H. Dobbertin, Cryptanalysis of MD5. Rump Session of Eurocrypt 96, May 1996, `http://www.iacr.org/conferences/ec96/rump/index.html`.

12. H. DOBBERTIN, A. BOSSELAERS AND B. PRENEEL, RIPEMD-160: A strengthened version of RIPEMD, *Fast Software Encryption*, Lecture Notes in Computer Science 1039, D. Gollmann, ed., Springer-Verlag, 1996.
13. R. IMPAGLIAZZO AND M. NAOR, Efficient cryptographic schemes provably as secure as subset sum. *Journal of Cryptology*, Vol. 9, No. 4, Autumn 1996.
14. B. KALISKI AND M. ROBSHAW, Message Authentication with MD5. *RSA Labs' CryptoBytes*, Vol. 1 No. 1, Spring 1995.
15. R. MERKLE, One way hash functions and DES. *Advances in Cryptology – Crypto* 89 *Proceedings*, Lecture Notes in Computer Science Vol. 435, G. Brassard ed., Springer-Verlag, 1989
16. M. NAOR AND M. YUNG, Universal one-way hash functions and their cryptographic applications. *Proceedings of the* 21st *Annual Symposium on Theory of Computing*, ACM, 1989.
17. National Institute of Standards, FIPS 180-1, Secure hash standard. April 1995.
18. B. PRENEEL AND P. VAN OORSCHOT, MD-x MAC and building fast MACs from hash functions. *Advances in Cryptology – Crypto* 95 *Proceedings*, Lecture Notes in Computer Science Vol. 963, D. Coppersmith ed., Springer-Verlag, 1995.
19. RIPE Consortium, Ripe Integrity primitives — Final report of RACE integrity primitives evaluation (R1040). Lecture Notes in Computer Science, vol. 1007, Springer-Verlag, 1995.
20. R. RIVEST, The MD4 message-digest algorithm, *Advances in Cryptology – Crypto* 90 *Proceedings*, Lecture Notes in Computer Science Vol. 537, A. J. Menezes and S. Vanstone ed., Springer-Verlag, 1990, pp. 303–311. Also IETF RFC 1320 (April 1992).
21. R. RIVEST, The MD5 message-digest algorithm. IETF RFC 1321 (April 1992).
22. J. ROMPEL, One-way functions are necessary and sufficient for digital signatures. *Proceedings of the* 22nd *Annual Symposium on Theory of Computing*, ACM, 1990.
23. G. Tsudik, Message authentication with one-way hash functions, *Proceedings of Infocom 92*, IEEE Press, 1992.
24. S. VAUDENAY, On the need for multipermutations: cryptanalysis of MD4 and SAFER. *Fast Software Encryption — Leuven Workshop*, Lecture Notes in Computer Science, vol. 1008, Springer-Verlag, 1995, 286–297.
25. WEGMAN AND CARTER, New hash functions and their use in authentication and set equality, *Journal of Computer and System Sciences*, Vol. 22, 1981, pp. 265–279.