An extended abstract of this paper appears in *Advances in Cryptology – Crypto 97 Proceedings*, Lecture Notes in Computer Science Vol. ??, B. Kaliski ed., Springer-Verlag, 1997. This is the full version.

Collision-Resistant Hashing: Towards Making UOWHFs Practical

Mihir Bellare[*] Phillip Rogaway[†]

July 10, 1997

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.

---

[*] 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://d--g-ut-p--tp--m-a--pvg.-o--i-eefood.rest/users/mihir`. Su part by NSF CAREER Award CCR-9624439 and a 1996 Packard Foundation Fellowship in Science and Engineering.

[†] 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`. Supported in part by NSF CAREER Award CCR-9624560.

Contents

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 c. The property usually desired of these functions is *collision-resistance*: it should be "hard" to find distinct strings M and M' such that $f(M) = f(M)$.

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 [10, 11, 12, 13] bring down our most popular constructions, MD4 and MD5 [25, 26]. 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 [22] and RIPEMD-160 [14] 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 weakened 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 [21]. But existing constructions, based on general or algebraic assumptions [21, 28, 17], 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 [20, 9] to the UOWHF setting. In other words, how to build "extended" 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 some existing constructions of UOWHFs. 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

Let $\Sigma = \{0,1\}$ be the binary alphabet. Informally, a function $f : \mathit{Msgs} \to \Sigma^c$ on some domain Msgs is a "compression function" if Msgs is the set of strings of some small length (eg., $\mathit{Msgs} = \Sigma^{640}$ for the compression function of MD5). It is an "extended hash function" if $\mathit{Msgs} = \Sigma^*$ (or at least some big subset of Σ^*). Either way, a *collision* for f is a pair $M, M' \in \mathit{Msgs}$ such that $M \neq M'$ but $f(M) = f(M')$. Still 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 [20, 9] takes a function $f\colon\ \Sigma^{c+m} \to \Sigma^c$ and extends it to a function $MDf\colon\ \Sigma^* \to \Sigma^c$. Assume for simplicity that $M = M_1 \cdots M_n$ is a sequence of exactly n blocks, each block of m bits. Fix $C_0 \in \Sigma^c$. Then compute $C_i = F(C_{i-1} \| M_i)$ and set $MDf(M) = C_n$. Roughly said, 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.

Most 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 [10], collisions were found by Dobbertin [12]. Later, collisions were found for the compression function of MD5, again by den Boer and Bosselaers [11], and in a stronger form by Dobbertin [13]. 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 popular hash functions mentioned above there is no explicit key. But Damgård [8, 9] defines ACR via keyed functions, and it is in this setting that he proves the MD construction correct [9]. Keying hash functions seems essential for a meaningful formalization of security.

When one treats things carefully, then, a hash function F should not have the signature described above: it must take *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 you hash according to F_K. In essence, the key for F specifies the particular function $f = F_K$ which is used to hash strings.

1.2 Target Collision-Resistance

With an ACR hash function F the key K is announced and the adversary wins if she manages to find *any* collision M, M' for $f = F_K$. The points M and M' may depend arbitrarily on K; any pair of distinct points will do. In the notion of Naor and Yung [21] the adversary no longer wins by finding just any collision. The adversary must choose one point, say M, in a way which does *not* depend on K, and then, later, given K, the adversary must find a second point M' (this time allowed to depend on K) such that M, M' is a collision for F_K. While it might be easy to find a collision M, M' in F_K by making both M, M' depend on K, the adversary may be unable to find collisions if she is forced to "commit" to one point of the collision before seeing K. We call this weakened notion of security *target collision-resistance* (TCR). (In the terminology of [21] it is universal one-wayness.)

Naor and Yung [21] 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 c 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. See Section 7. 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 Section 7, 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 convenient way to make a TCR hash function is to directly 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.

Method	**Key length**	**See**
Basic Linear Hash $H \mapsto LH$	Lk/m	Section 5.2 and Figure 3
XOR Linear Hash $H \mapsto XLH$	$k + Lc/m$	Section 5.3 and Figure 4
Basic Tree Hash $H \mapsto TH$	$k \log_d(L/c)$	Section 5.4 and Figure 5
XOR Tree Hash $H \mapsto XTH$	$k + dc\log_d(L/c)$	Section 5.5 and Figure 6

Figure 1: The schemes of this paper which turn a TCR compression function H into an extended TCR hash function. Here k is the key length of H. For the first two schemes $H_K : \Sigma^{c+m} \to \Sigma^c$ and for the next two schemes $H_K : \Sigma^{dc} \to \Sigma^c$, for some $d \geq 2$ and any k-bit key K. In all cases, L is the length of the message to be hashed (measured in bits).

1.4 Extending TCR Compression Functions to TCR Hash Functions

Instead of keying the entire hash function at once a good strategy might be to key just the compression function. Then one could hope to transform this TCR compression function into an extended 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 k-bit key specifies a map H_K: $\Sigma^{m+c} \to \Sigma^c$. 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.

Let us clarify one point. Doesn't the function resulting from the MD construction meet even the stronger notion of ACR? The problem is that we are starting from a weaker compression function: our compression function is only TCR. We find this is not enough to imply that the hash function meets the weaker TCR notion.

LINEAR HASH: BASIC AND XOR. To preserve TCR, the most direct extension we found to the MD construction is to 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.[1]

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 HASH. To get major reductions in key size we turn to trees. Wegman and Carter [31] 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

---

[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.

same method [21, 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 H with key length k mapping dc bits to c bits, and we want to hash a L bit message down to m bits. The basic tree construction yields a hash function with a key size of $k \log_d(L/c)$ bits. Key lengths have been reduced, but one can reduce them more.

The XOR tree hash. 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 k mapping dc bits to c bits, and we want to hash L bits to c bits, the XOR tree construction yields a hash function with a key size of $k + dc \log_d(L/c)$ bits.

Here c is short, like 64 bits, since we do not need to worry about birthday attacks for TCR functions. On the other hand, k can be quite large (and in many constructions, it is). So the key length for the XOR tree hash will usually be much better than the key length for the basic tree hash.

Summary. For a summary of the constructions and their key lengths, see Figure 1.

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 [17] uses a key of size Ls bits to hash L bits to s bits, where s is a security parameter which controls subset sum instance sizes. (Think of s 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 $2s$ bits to s bits (it has key length $k = 2s^2$) the key size of the resulting hash function is $k \lg(L/s) = 2s^2 \lg(L/s)$. With our (binary) XOR tree scheme, the key size of the resulting function is $k + 2s \lg(L/s) = 2s(s + \lg(L/s))$. The latter can be quite a bit smaller. For example for $s = 300$ and a message of length $L = 10$ KBytes, the key length for the basic tree scheme is about 182 KBytes while that for the XOR trees scheme is about 23 KBytes, so that the gain is a factor of about 8.

Domains and collision lengths. Strings to be hashed may be of (virtually) any length at all. Nonetheless, it is often convenient to think of messages as having lengths which are multiples of some fixed block size, like 512. This restriction can be removed using simple padding techniques. (For example, append to each message a “1” bit and then the minimal number of “0” bits so that the padded message is in the domain of the hash function. This method, and many others, provably preserve TCR, and ACR, too.) For details, see Section 6.

Our proofs of security will rule out adversaries who can find collisions for equal-length strings M, M'. In practice, collisions between strings of unequal length have to be prevented, too. To handle this we again provide a general construction. But this time the standard padding techniques do not necessarily work. We give a method that does. It turns a hash function secure against equal-length strings $M, M' \in \mathit{Msgs}$ into a hash function secure against collisions of arbitrary strings $\mathit{shortMsg}, \mathit{longMsg} \in \mathit{Msgs}$. The method requires just one extra application of the compression function. See Section 6.

It is the above two results which effectively justify our restricting our attention to hash functions that resist equal-length collisions for some set of convenient input lengths.

1.6 Related Work

We have already described the most closely related work, which is due to Naor and Yung [21], Merkle [20], and Damgård [8, 9].

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

A good deal of work has gone into keying hash functions for message authentication [1, 29, 18, 23]. In particular, HMAC is a popular solution to this problem [1, 19]. The difference is that in the message authentication setting, parties share a secret key, which is used to key the hash function; in our setting, there are no secret keys, and the hash function is to be keyed with a key that, although chosen at random, is eventually available to the adversary.

Bellare, Canetti and Krawczyk [2] considered keyed compression functions as pseudorandom functions, and showed that applying the MD construction then yields a pseudorandom function. Again, the difference is that the notion of pseudorandomness relies on the secrecy of the key.

A weaker-than-standard notion of hashing is considered in [1]. However their notion is based on a hidden key and hash functions meeting their notion, although useful in the message authentication setting, don't suffice for digital signatures, where the computation of the hash function must be public to enable signature verification.

A preliminary version of this paper appeared as [4]. This is the full version.

2 Notions of Hashing

Hash functions like MD5 or SHA-1 have no explicit key. But no notion of collision-freeness has been offered for such a keyless setting. To get a sense why this is so, suppose f is a function $f : \Sigma^* \to \Sigma^c$, for some integer c. We would like to say it is collision-free if there is no efficient program that can find collisions in f. But in fact, no matter what is f, there *is* such a program. Clearly there exists a pair M, M' which is a collision for f, and hence there exists a program which very quickly finds collisions, namely the program that has the description of M, M' embedded in its code, and just outputs M, M'. While, in practice, it may be "difficult" to explicitly find this program, a formalization in terms of the existence of collision-finding programs is ruled out. It seems the natural way to get a meaningful notion of security is to talk about families of functions.

FAMILIES OF HASH FUNCTIONS. In a family of hash functions F each key K specifies a particular hash function $f = F_K$ in the family. Each such function maps Msgs to Σ^c where $\mathit{Msgs} \subseteq \Sigma^*$ is some set of messages associated to the family, and c is the *hash length* (output length) associated to the family. The key K will be taken from some key space Σ^k, and k will be called the *key length.* If $\mathit{Msgs} = \Sigma^\ell$ for some ℓ then ℓ is called the *input length.*

Formally, a family F of (keyed) hash functions is a map $F\colon \Sigma^k \times \mathit{Msgs} \to \Sigma^c$. We define $F_K\colon \mathit{Msgs} \to \Sigma^c$ by $F_K(M) = F(K, M)$ for each $K \in \Sigma^k$ and each $M \in \mathit{Msgs}$. We use either the notation $F_K(M)$ or $F(K, M)$, as convenient.

The hash family F is a *compression function* if the domain is $\mathit{Msgs} = \Sigma^\ell$ for some small constant ℓ (eg., $\ell = 512$). It is an *extended hash function* if Msgs is contains long strings.

We say that $f : \mathit{Msgs} \to \Sigma^*$ is *length consistent* if $|f(M)| = |f(M')|$ whenever $|M| = |M'|$. A family of hash functions F is length consistent if $|F_K(M)| = |F_{K'}(M')|$ whenever $|M| = |M'|$ and $|K| = |K'|$.

COLLISIONS. Recall a *collision* for a function f defined on a domain Msgs is a pair of strings $M, M' \in \mathit{Msgs}$ such that $M \neq M'$ but $f(M) = f(M')$. In our setting the function of interest will be $f = F_K$ 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. We will define both below. First some technicalities.

PROGRAMS AND TIMING. We fix some RAM (random access machine) model of computation, including pointers, as in any algorithms text (eg. [6]), and we measure execution time of a program with respect to that model. An *adversary* is a program for our model, written in some fixed programming language. Any program is allowed randomness: the programming language supports a $\Theta(\log n)$-time operation $\textsc{FlipCoin}(n)$ which returns a random number between 1 and n. 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.* (This prevents, for example, the possibility of declaring very efficient a program that stores in its code a table giving collisions for lots of different key values.) If $F : \Sigma^k \times \mathit{Msgs} \to \Sigma^c$ is a family of hash functions we let T_F indicate the worst-case time to compute $F_K(M)$, in the underlying model of computation, when $K \in \Sigma^k$ and $M \in \mathit{Msgs}$. This may be infinite or enormous if *Msgs* is. To handle that possibility we let $T_{F,\ell}$ denote the worst-case time to compute $F_K(M)$ when $K \in \Sigma^k$ and $M \in \mathit{Msgs} \cap \Sigma^{\leq \ell}$.

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 an M, M' for f. In the keyed setting, it can be formalized like this (cf. [8, 9]). An adversary CF, called a *collision-finder,* is given K chosen at random from Σ^k and is said to *succeed* if it outputs a collision M, M' for F_K. We measure the quality of a hash function by seeing how successful an adversary can be when compared against the adversary's resource expenditure. Formally, a collision-finder CF is said to (t, μ, ϵ)*-break* the family of hash functions $F : \Sigma^k \times \mathit{Msgs} \to \Sigma^k$ if the running time of the adversary is at most t, strings M, M' that CF outputs have length at most μ, and the probability that CF, on input K, outputs a collision M, M' for F_K is at least ϵ. Here the probability is take over K (a random point in Σ^k) and CF's random coins.

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.

Often we don't care about the length of the collisions that an adversary may find. In this case we omit μ from the notation above.

Informally, we say that F is "any collision-resistant" (ACR) if for every collision-finder who (t, ϵ)-breaks F, the ratio t/ϵ is large.

2.2 Target Collision-Resistance — TCR

In the notion of [21] the adversary does not get credit for finding any old collision. The adversary must still find a collision M, M', but now M is not allowed to depend on the key: the adversary must choose it before the key K is known. Only after "committing" to M does the adversary get K. Then it must find M'.

Formally, the adversary $\mathsf{CF} = (\text{CF-I}, \text{CF-II})$ (called a *target collision finder*) consists of two algorithms, CF-I and CF-II. First, CF-I is run, to produce M and possibly some extra "state information," *State*, that CF-I wants to pass to CF-II. We call M the *target message*. Now, a random key K is chosen and CF-II is run. Algorithm CF-II is given K, M, State and must find M' different from M such that $F_K(M) = F_K(M')$. We call M' the *sibling message.* The sibling message can depend on the key, but the target message can not.

The formalization of [21] 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 $\mathsf{CF} = (\text{CF-I}, \text{CF-II})$ is called an *equal-length target collision finder* if the messages M, M' which CF-I outputs always satisfy $|M| = |M'|$. It is called a *variable-length target collision finder* when no such restriction is made on the relative lengths of M, M'.

Let $\mathsf{CF} = (\text{CF-I}, \text{CF-II})$ be a target-collision finder. We say that it (t, μ, ϵ)*-breaks* F if its running time is at most t, the strings M, M' output by CF are of length at most μ, and CF finds a collision with probability at least ϵ. The running time is the sum of the running times for CF-I and CF-II, and the probability is over the coins of CF and the choice of K. We say that F is (t, μ, ϵ)*-resistant to equal-length target collisions* if there is no equal-length target collision finder which (t, μ, ϵ)-breaks F. We say that F is (t, μ, ϵ)*-resistant to variable-length target collisions* if there is no variable-length target collision finder which (t, μ, ϵ)-breaks F. If we say that F is (t, μ, ϵ)-TCR, or (t, μ, ϵ)-resistant to target collisions, we mean it is (t, μ, ϵ)-resistant to variable-length target collisions.

We will sometimes write $\mathit{ProbSuccess}(\mathsf{CF}, F)$ to denote the probability that CF finds a collision in F.

Often we don't care about the length of the collisions that an adversary may find. In this case we omit μ from the notations above.

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, ϵ)-breaks F, the ratio t/ϵ is large.

Remark 2.1 Notice that we do not restrict the adversary to any particular attack strategy. If a family of hash functions meets this notion of security, then it is secure against *all* attacks that can be run given the prescribed resources. This is the advantage of the "provable security" approach.

Remark 2.2 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.

Remark 2.3 Consider the following alternative syntax for a target collision-finder: it is an algorithm B together with a string M. The associated notion for target collision-resistance is that (B, M) is *successful* in attacking F if, on input K, algorithm B outputs an M' such that $M' \neq M$ and $F_K(M) = F_K(M')$. This definition is in some ways simpler than our "find-target/find-sibling" notion, since there is only one algorithm involved and, consequently, we have no state information to communicate from one algorithm to another.

The two notions are actually equivalent. Certainly the alternative notion is not stronger: given (B, M), we could have constructed $\mathsf{CF} = (\text{CF-I}, \text{CF-II})$ where CF-I outputs M, λ (where λ is the empty string) and CF-II behaves like B. To see that the alternative notion is not weaker, start with a collision finder $\mathsf{CF} = (\text{CF-I}, \text{CF-II})$. Consider the random coins that cause CF-I to maximize the probability that CF will succeed. For these random coins there is a resulting (M, State) which CF-I will output. Construct a (M, B) using this message M and letting B be the algorithm which behaves like CF-II started in the state indicated by State. The success probability of B will be at least the success probability of CF.

Why did we select our "find-target/find-sibling" formalization instead of the alternative one? First, the find-target/find-sibling formalization more directly mirrors our intuition, reflecting the observation that, in the real world, there *is* computation associated to finding the target message. Second, if one considers a parameterized collection of hash families, a different family for each key length k, then our find-target/find-sibling notion immediately generalizes to give a proper, uniform notion for security for such objects. This is not true for the alternative notion.

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 [21] 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 $H_1\colon\ \Sigma^{k_1}\times\Sigma^{\ell_1}\to\Sigma^{\ell_2}$ and $H_2\colon\ \Sigma^{k_2}\times\Sigma^{\ell_2}\to\Sigma^{c}$ be families of hash functions. The *composition* $H_2\circ H_1\colon\ \Sigma^{k_1+k_2}\times\Sigma^{\ell_1}\to\Sigma^{c}$ is the family defined by

$$(H_2\circ H_1)(K_1K_2,\ M)\ =\ H_2(K_2,\ H_1(K_1,M))\ ,$$

for all $K_1\in\Sigma^{k_1}$, $K_2\in\Sigma^{k_2}$, and $M\in\Sigma^{\ell_1}$. From the proof of Naor and Yung's composition lemma [21] we extract the concrete security parameters to get the following. For completeness a proof is provided in Appendix A.

Lemma 3.1 (TCR composition lemma) *Let $H_1\colon\ \Sigma^{k_1}\times\Sigma^{\ell_1}\to\Sigma^{\ell_2}$ and $H_2\colon\ \Sigma^{k_2}\times\Sigma^{\ell_2}\to\Sigma^{c}$ be families of hash functions. Assume the first is (t_1,ϵ_1)-secure against target collisions and the second is (t_2,μ_2,ϵ_2)-secure against target collisions. Then the composition $H=H_2\circ H_1$ is (t,μ,ϵ)-secure against target collisions, where*

$$\begin{aligned} t &= \min(t_1-\Theta(k_2),\ t_2-T_{H_1,\mu_2}-\Theta(k_1)) \\ \mu &= \mu_2 \\ \epsilon &= \epsilon_1+\epsilon_2\ . \end{aligned}$$

In this paper we also 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 Appendix A for the proof.

Lemma 3.2 (TCR composition lemma for equal-length collisions) *Let $H_1\colon\ \Sigma^{k_1}\times\Sigma^{\ell_1}\to\Sigma^{\ell_2}$ and $H_2\colon\ \Sigma^{k_2}\times\Sigma^{\ell_2}\to\Sigma^{c}$ 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,μ_2,ϵ_2)-resistant to equal-length target collisions. Then the composition $H=H_2\circ H_1$ is (t,μ,ϵ)-resistant to equal-length target collisions, where*

$$\begin{aligned} t &= \min(t_1-\Theta(k_2),\ t_2-2T_{H_1,\mu_2}-\Theta(k_1)) \\ \mu &= \mu_2 \\ \epsilon &= \epsilon_1+\epsilon_2\ . \end{aligned}$$

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.

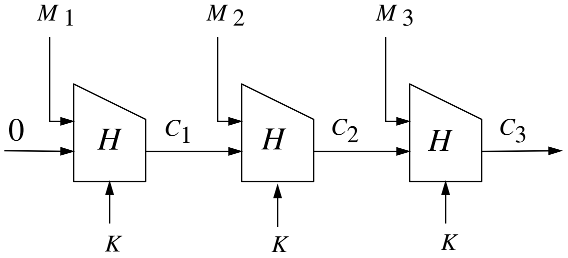

Figure 2: Construction *MDH* — The Merkle-Damgård construction with a common key K. It is possible for H to be TCR and yet *MDH* might fail to be.

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 $\text{MD5}^{**}_K(M) = \text{MD5}(K\|M\|K)$. It seems likely that any extension of Dobbertin's attack [13] which finds collisions in MD5 would also defeat MD5**. Letting md5 denote the compression function of MD5, note that if for any $C \in \Sigma^{128}$ you can find distinct $M, M' \in \Sigma^{512}$ such that $\text{md5}(C\|M) = \text{md5}(C\|M')$, 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 bits in with 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 keyed compression function can then be extended to an extended 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 each of which has some fixed length of m bits. For notational simplicity, let $\Sigma_m = \Sigma^m$ be the space of possible message blocks. A message is then regarded as $M = M_1 \cdots M_n$ where $M_i \in \Sigma_m$ for each $i = 1, \ldots, n$. The number of m-bit blocks in such a message M is denoted by $n = |M|_m$. Typically N will stand for some maximum number of allowed blocks, so that $n \leq N$.

We are given a TCR compression function H. We wish to build an extended function H'. We begin by looking at the method used in the ACR setting.

5.1 The MD Construction Doesn't Propagate TCR

Suppose we start with a compression function $H\colon\ \Sigma^k \times \Sigma^{c+m} \to \Sigma^c$ and we want to hash a message $M_1 \cdots M_n \in \Sigma_m^n$. The MD method gives a keyed family of functions $MDH^n\colon\ \Sigma^k \times \Sigma_m^n \to \Sigma^c$ as follows. First fix some c-bit initial vector IV, say IV $= 0^c$. We then define MDH according to:

```
Algorithm $MDH^n(K,\ M)$
$C_0 \leftarrow$ IV
for $i = 1, \ldots, n$ do
    $C_i \leftarrow H(K,\ C_{i-1} \| M_i)$
return $C_n$
```

For a picture, see Figure 2.

Damgård [9] 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 above 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). Calling this F, we construct H so that H is still TCR, but MDH^n is not TCR, for some integer n. The proposition below gives the exact bounds with which H, on the one hand, inherits the TCRness of F, and, on the other hand, MDH^n can be broken.

Proposition 5.1 *Suppose there exists a compression function* F: $\Sigma^k \times \Sigma^{c+m'} \to \Sigma^c$ *with* $m' > k$ *such that* F *is* (t', ϵ')*-resistant to target collisions. Then there exists a compression function* H *such that*

(1) H *is* (t, ϵ)*-resistant to target collisions for* $t = t' - \Theta(k + m')$ *and* $\epsilon' = \epsilon + 2^{-k+1}$

(2) *There is a collision-finder that* (t, ϵ)*-breaks* MDH^2, *where* $t = \Theta(m')$ *and* $\epsilon = 1 - 2^{-k}$.

Proof: We set $m = m' - k$, which is positive by assumption. We will construct H: $\Sigma^k \times \Sigma^{(c+k)+m} \to \Sigma^{c+k}$ such that H is TCR but MDH^2 is not. The construction of H is like this. For $K \in \Sigma^k$, $x \in \Sigma^c$, $y \in \Sigma^k$ and $z \in \Sigma^m$, let

$$H(K, x \| y \| z) \;=\; H_K(x \| y \| z) \;=\; \begin{cases} F_K(x \| y \| z) \| K & \text{if } y \neq K \\ 1^c \| 1^k & \text{if } y = K. \end{cases}$$

First we claim H is TCR secure. Second we claim that MDH^2 is not. Lets check the latter first.

Let IV $=$ IV$_1 \|$ IV$_2$ be the $(c + k)$-bit initial vector. (IV$_1$ is the first c bits and IV$_2$ the rest. This is chosen independently of H and our attack works regardless of its value.) Here is the attack. In the first stage our collision finder must output a two block string M. It outputs $M = 0^m \| 0^m$. (Recall the block length of H is m.) Now, in the second stage, the collision finder receives K. It ignores K and outputs $M' = 1^m \| 0^m$. Since K is chosen at random, it is different from IV$_2$ with high probability (at least $1 - 2^{-k}$), and under this assumption one can check that

$$\begin{aligned} MDH_K(0^m \| 0^m) &= H_K(H_K(\mathrm{IV} \| 0^m) \| 0^m) = H_K(F_K(\mathrm{IV} \| 0^m) \| K \| 0^m) = 1^c \| 1^k \\ MDH_K(1^m \| 0^m) &= H_K(H_K(\mathrm{IV} \| 1^m) \| 0^m) = H_K(F_K(\mathrm{IV} \| 1^m) \| K \| 0^m) = 1^c \| 1^k\ . \end{aligned}$$

So M, M' is a collision for MDH^2, meaning the latter is not TCR.

Now we need to check that H, however, was TCR. We claim this is true because by assumption F is TCR. The intuition is that as long as $y \neq K$, the first block of the output of H is just the output of F and so one can't find collisions here. But since the target message must be specified before seeing K, the adversary has only a 2^{-k} chance of having $y = K$ in the target message $x \| y \| z$.

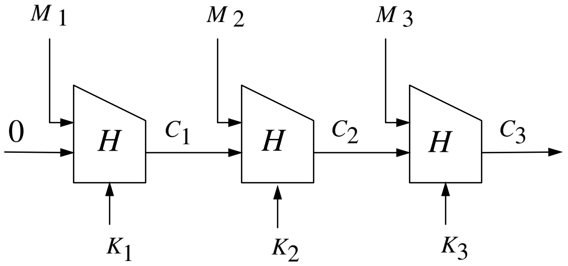

Figure 3: Construction LH — The basic linear scheme. If H is TCR then LH is TCR, too.

Formally we claim that if F was (t', ϵ')-resistant to target collisions then H is (t, ϵ)-resistant to target collisions for $t = t' - \Theta(k + c + m)$ and $\epsilon = \epsilon' + 2^{-k+1}$. To see this, suppose $\mathsf{CF} = (\text{CF-I}, \text{CF-II})$ is a target collision finder which (t, ϵ)-breaks H. We construct a collision finder $\mathsf{CF}' = (\text{CF-I}', \text{CF-II}')$ which (t', ϵ')-breaks F. CF-I$'$ runs CF-I to get a target message $M_1 = x_1 \parallel y_1 \parallel z_1$, and outputs the same target message. Now CF-II$'$ receives a random k-bit key K, and is given M_1. It wants to find $M_2 = x_2 \parallel y_2 \parallel z_2$ so that M_1, M_2 is a collision for F_K. If $K = y_1$ it aborts, but this happens only with probability 2^{-k}. Now it gives K to CF-II, along with M_1. CF-II outputs a message $M_2 = x_2 \parallel y_2 \parallel z_2$. Suppose M_1, M_2 is a collision for H_K. We now consider two cases, that $y_2 = K$ and $y_2 \neq K$. In the former case the probability that $H_K(M_1) = H_K(M_2)$ is at most 2^{-k}, because it can only happen if $K = 1^k$, and K was chosen at random. (The last block of $H_K(M_2)$ is 1^k and the last block of $H_K(M_1)$ is K.) In the latter case, having $H_K(M_1) = H_K(M_2)$ means we must have $F_K(M_1) = F_K(M_2)$, so that M_1, M_2 is a collision for F_K. Thus the collision finder CF' runs in time $t + O(k + c + m)$ and $\mathit{ProbSuccess}(\mathsf{CF}', F) \geq 1 - 2^{-k+1}$. The result follows. ▌

One might criticize the example above for being somewhat "artificial." But recall the goal is to find general constructions that work for *any* compression function. What the above shows is that this hope is lost for the standard MD construction.

Remark 5.2 There is a possible source of confusion on the subject of how a concrete hash function like MD5 can be seen as an application of the MD construction. In functions like MD5 there is no explicit key, so the relationship is not so obvious. There is, however, a compression function, call it md5, and one often thinks of this compression function as taking two arguments: a 128-bit chaining value, C, and the 512-bit message block, M. From that one might assume that md5(C, M) is the concrete realization of a family of family of hash functions $H_C(M)$. But what happens in MD5 would then be completely different from what Damgård defined— and rightly so, since it is easy to see that chaining $H_C(M)$ as it is done in MD5 would not preserve collision resistance in the sense of ACR or TCR. Instead, the proper viewpoint for seeing MD5 as an instance of the MD-construction is to think of the input message to md5 as the entire 640-bits, so that $H_K(C \| M) = \text{md5}(C \| M)$. Thus md5 corresponds to H_K. What, then, is K? In essence, it is unpredictable choices that were involved in deciding on the md5 algorithm itself— Rivest choose the key K and that key *is* md5. Only under this viewpoint are the MD4-family of hash functions instances of the Merkle/Damgård construction.

5.2 The Basic Linear Hash

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

Let H: $\Sigma^k \times \Sigma^{c+m} \to \Sigma^c$ be the given TCR compression function. To hash $M = M_1 \ldots M_n \in \Sigma_m^n$ we use n keys, $K_1, \ldots, K_n$, one key for each application of the underlying compression function. Namely, fixing some IV, for concreteness $\mathrm{IV} = 0^c$, we have:

```
**Algorithm** $LH(K_1 \ldots K_n,\ M)$
$C_0 \leftarrow \mathrm{IV}$
**for** $i = 1, \ldots, n$ **do**
    $C_i \leftarrow H(K_i, C_{i-1} \,\|\, M_i)$
**return** $C_n$
```

This is depicted in Figure 3. The family of hash functions LH^N: $\Sigma^{Nk} \times \Sigma_m^{\leq N} \to \Sigma^c$ is defined by letting $LH^N(K_1 \ldots K_N, M) = LH(K_1 \ldots K_n, M)$ where $n = |M|_m$, for every $K_1, \ldots, K_N \in \Sigma^k$ and every $M \in \Sigma_m^{\leq N}$. For notational convenience we define $LH^0(\lambda, \lambda) = 0^c$ and notice that for all $n \geq 1$,

$$LH(K_1 \cdots K_n,\ M_1 \ldots M_n) \quad = \quad H_{K_n}(LH(K_1 \cdots K_{n-1},\ M_1 \cdots M_{n-1}) \,\|\, M_n)\ .$$

The following theorem says that if the compression function H is resistant to target collisions then so is the extended hash function LH^N.

Theorem 5.3 *Suppose* H: $\Sigma^k \times \Sigma^{c+m} \to \Sigma^c$ *is* (t', ϵ')*-resistant to target collisions. Suppose* $N \geq 1$. *Then* LH^N: $\Sigma^{Nk} \times \Sigma_m^{\leq N} \to \Sigma^c$ *is* (t, ϵ)*-resistant to equal-length target collisions, where* $\epsilon = N\epsilon'$ *and* $t = t' - \Theta(N) \cdot (T_H + m + k + c)$.

Proof: We begin with the following observation. If $M, M' \in \Sigma_m^n$ is a collision for $LH^N(K_1 \ldots K_N, \cdot)$ —meaning $M \neq M'$ but $LH^N(K_1 \ldots K_N, M) = LH^N(K_1 \ldots K_N, M')$— then there exists a $j \in \{1, \ldots, n\}$ such that the following hold:

$$\left\{ \begin{array}{rcl} LH_{K_1 \ldots K_j}(M_1 \ldots M_j) & = & LH_{K_1 \ldots K_j}(M'_1 \ldots M'_j) \\ LH_{K_1 \ldots K_{j-1}}(M_1 \ldots M_{j-1}) \,\|\, M_j & \neq & LH_{K_1 \ldots K_{j-1}}(M'_1 \ldots M'_{j-1}) \,\|\, M'_j\ . \end{array} \right. \tag{1}$$

This is not hard to see, by "tracing back" the collision. We propose to exploit this to find collisions in H.

For the proof, suppose $\mathsf{CF} = (\text{CF-I}, \text{CF-II})$ is a equal-length target collision finder which (t, ϵ)-breaks LH^N. We construct a target collision finder $\mathsf{CF}' = (\text{CF-I}', \text{CF-II}')$ which (t', ϵ')-breaks H. The definition of CF' is as follows:

```
**Algorithm** CF-I′
$(M, State) \leftarrow$ CF-I and $n \leftarrow |M|_m$
$i \xleftarrow{R} \{1, \ldots, n\}$
$K_1, \ldots, K_{i-1} \xleftarrow{R} \Sigma^k$
$x \leftarrow LH(K_1 \ldots K_{i-1}, M_1 \ldots M_{i-1}) \,\|\, M_i$
**return** $(x, (i, K_1, \ldots, K_{i-1}, M, State))$
```

```
**Algorithm** CF-II′$(K, x, (i, K_1, \ldots, K_{i-1}, M, State))$
$K_i \leftarrow K$
$K_{i+1}, \ldots, K_N \xleftarrow{R} \Sigma^k$
$M' \leftarrow$ CF-II$(K_1 \ldots K_N, M, State)$
$x' \leftarrow LH(K_1 \ldots K_{i-1}, M'_1 \ldots M'_{i-1}) \,\|\, M'_i$.
**return** $x'$
```

We must now bound the probability that x, x' is a collision for $H(K, \cdot)$ in the experiment describing the attack of CF' on H. Notice that the distribution on the keys $K_1, \ldots, K_N$ is uniform (remember $K = K_i$ too is chosen at random in the experiment) and so CF finds a collision with probability $\mathit{ProbSuccess}(\mathsf{CF}, LH^N) > \epsilon$. The distribution of the keys is also independent of i, and the latter was chosen at random, so if M, M' is a collision for $LH^N(K_1 \ldots K_N, \cdot)$ then we have $i = j$ (where j is the value of Equation (1)) with probability $1/n \geq 1/N$. So $\epsilon' > \epsilon/N$.

The running time of CF' is that of CF plus the overhead. This overhead is $\Theta(N)[m + T_H + k + c]$. The choice of t in the theorem statement makes all this at most t', from which we conclude the result. ▌

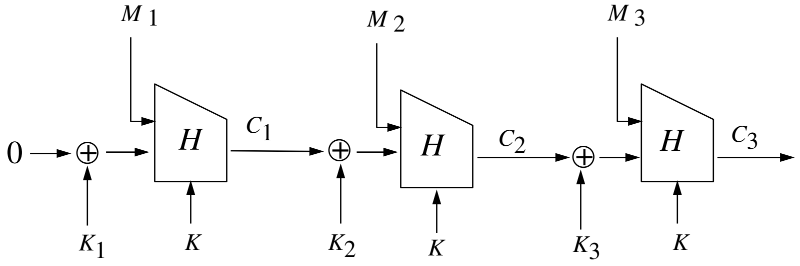

Figure 4: Construction *XLH* — The XOR linear scheme. Compared to *LH*, the key size may be reduced. But it is still long.

Remark 5.4 We emphasize that, with the above theorem, when the key K is given to the adversary following the adversary's identifying the target collision $M = M_1 \cdots M_n$, it is the *entire* key $K = K_1 \cdots K_N$ which is given to the adversary, and not just the prefix of it $K_1 \cdots K_n$. This makes the result stronger. Subsequent theorems will be the same.

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 iteration, but an auxiliary "mask" key K_i, depending on the iteration number i, is XORed to the chaining variable in the i-th iteration. 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."

More precisely, to hash $M = M_1 \ldots M_n \in \Sigma_m^n$ ($n \le N$) we use one key $K \in \Sigma^k$ for the compression function and auxiliary keys $K_1, \ldots, K_n \in \Sigma^c$, as follows. As usual IV $= 0^c$.

```
Algorithm $XLH(KK_1 \ldots K_n,\ M)$
$C_0 \leftarrow \mathrm{IV}$
for $i = 1, \ldots, n$ do
    $D_{i-1} \leftarrow K_i \oplus C_{i-1}$
    $C_i \leftarrow H(K_i, D_{i-1} \,\|\, M_i)$
return $C_n$
```

This is depicted in Figure 4. The family of hash functions $XLH^N\colon\ \Sigma^{k+Nc} \times \Sigma_m^{\le N} \to \Sigma^c$ is defined by letting $XLH^N(KK_1 \ldots K_N, M) = XLH(KK_1 \ldots K_n, M)$ where $n = |M|_m$, for every $K \in \Sigma^k$, every $K_1, \ldots, K_N \in \Sigma^c$, and every $M \in \Sigma_m^{\le N}$. For notational convenience we define $XLH(K, \lambda) = 0^c$ and notice that for all $n \ge 1$

$$XLH(KK_1 \ldots K_n, M_1 \ldots M_n) \quad = \quad H_K(\,(\,K_n \oplus XLH(KK_1 \ldots K_{n-1}, M_1 \ldots M_{n-1})\,)\,\|\, M_n)\ .$$

The following theorem says that if the compression function H is resistant to target collisions then so is the extended hash function XLH^N.

Theorem 5.5 *Suppose* $H\colon\ \Sigma^k \times \Sigma^{c+m} \to \Sigma^c$ *is* (t', ϵ')*-resistant to target collisions. Suppose* $N \ge 1$*. Then* $XLH^N\colon\ \Sigma^{k+Nc} \times \Sigma_m^{\le N} \to \Sigma^c$ *is* (t, ϵ)*-resistant to equal-length target collisions, where* $\epsilon = N\epsilon'$ *and* $t = t' - \Theta(N) \cdot (T_H + m + k + c)$*.*

Proof: We follow and modify the proof of Theorem 5.3. The starting observation is that if $M, M' \in \Sigma_m^n$ is a collision for $XLH^N(KK_1 \ldots K_N, \cdot)$ —meaning $M \ne M'$ but $XLH^N(KK_1 \ldots K_N, M) =$

$XLH^N(KK_1 \ldots K_N, M')$— then there exists a $j \in \{1, \ldots, n\}$ such that the following hold:

$$\left\{ \begin{array}{rcl} XLH_{KK_1 \ldots K_j}(M_1 \ldots M_j) & = & XLH_{KK_1 \ldots K_j}(M'_1 \ldots M'_j) \\ XLH_{KK_1 \ldots K_{j-1}}(M_1 \ldots M_{j-1}) \parallel M_j & \neq & XLH_{KK_1 \ldots K_{j-1}}(M'_1 \ldots M'_{j-1}) \parallel M'_j \, . \end{array} \right. \quad (2)$$

Again this is not hard to see, by "tracing back" the collision, and also cancelling the value of K_j which is XORed to he output of the $(j-1)$-th stage for both messages. Now given equal-length target collision finder $\mathsf{CF} = (\text{CF-I}, \text{CF-II})$ which (t, ϵ)-breaks XLH^N we construct target collision finder $\mathsf{CF}' = (\text{CF-I}', \text{CF-II}')$ as follows:

```
Algorithm CF-I'
$(M, State) \leftarrow$ CF-I and $n \leftarrow |M|_m$
$i \xleftarrow{R} \{1, \ldots, n\}$
$D \xleftarrow{R} \Sigma^c$
$x \leftarrow D \parallel M_i$
return $(x, (i, D, M, State))$
```

```
Algorithm CF-II'$(K, x, (i, D, M, State))$
$K_1, \ldots, K_{i-1}, K_{i+1}, \ldots, K_N \xleftarrow{R} \Sigma^c$
$C \leftarrow XLH(KK_1 \ldots K_{i-1}, M_1 \ldots M_{i-1})$
$K_i \leftarrow D \oplus C$
$M' \leftarrow$ CF-II$(KK_1 \ldots K_N, M, State)$
$D' \leftarrow K_i \oplus XLH(KK_1 \ldots K_{i-1}, M'_1 \ldots M'_{i-1})$
$x' \leftarrow D' \parallel M'_i$
return $x'$
```

The idea is that CF' wants K to play the role of the primary key for XLH^N. It is also hoping that $i = j$. It wants that the collision for H_K be x, x' where $x = D_{i-1} \parallel M_i$ and $x' = D'_{i-1} \parallel M'_i$ where D_{i-1}, D'_{i-1}, are, respectively, the values of the masked chaining variables for M, M', after the $(i-1)$-st stage in XLH^N, namely $D_{i-1} = K_i \oplus XLH(KK_1 \ldots K_{i-1}, M_1 \ldots M_{i-1})$ and $D'_{i-1} = K_i \oplus XLH(KK_1 \ldots K_{i-1}, M'_1 \ldots M'_{i-1})$. However, before it knows K, it has no way of knowing D_i, because the latter is a function of K, so how can it output a target message? The trick is to set $x = D \parallel M_i$ for some random D. Later, after knowing K, CF-II will pick K_i so that this value of D is correct, ie. indeed $D = D_{i-1}$ for the chosen keys. This is done by choosing $K_i = D \oplus XLH(KK_1 \ldots K_{i-1}, M_1 \ldots M_{i-1})$. Notice that this K_i chosen by CF-II is random and independent of all other keys because C was random. So the distribution on the key for XLH^N that is provided to CF-II is correct.

Given this the probability that x, y is a collision for $H(K, \cdot)$ can be computed as in the proof of Theorem 5.3, based on Equation (2), and the bound on the running time can be made similarly. ▮

5.4 The Basic Tree Hash

A tree can be used to reduce the key size. We are slightly more general than [21], considering d-ary trees for $d \geq 2$, and also allowing the message to be hashed to have a number of blocks less than the maximum, as opposed to mandating that all messages have the maximum number of blocks.

We start with a compression function $H\colon\ \Sigma^k \times \Sigma^{dc} \to \Sigma^c$. We first describe a primitive we will use.

PARALLEL HASH. We are given a message M with length a multiple of dc, and view it as $M = M_1 \ldots M_n$ where $M_i \in \Sigma_{dc}$. We hash each block using the compression function and concatenate the results. More precisely,

```
Algorithm $PH(K, M)$
$n \leftarrow |M|_{dc}$
for $i = 1, \ldots, n$ do
```

```
    $N_i \leftarrow H(K, M_i)$
return $N_1 \| \cdots \| N_n$
```

For any $N \geq 1$ the above definition gives rise to a family of hash functions $PH^N\colon\ \Sigma^k \times (\Sigma_c \cup \Sigma_{dc}^{\leq N}) \rightarrow \Sigma_c^{\leq N}$ defined as follows:

$$PH^{dN}(K, M) = \begin{cases} PH(K, M) & \text{if } |M| \text{ is a multiple of } dc \\ M & \text{if } |M| = c \end{cases}$$

Notice that only one key is used. Notice too that PH^N agrees with H when the input is of dc bits.

Lemma 5.6 *Suppose* $H\colon\ \Sigma^k \times \Sigma^{dc} \rightarrow \Sigma^c$ *is* (t', ϵ')*-resistant to target collisions. Suppose* $N \geq 1$*. Then* $PH^{dN}\colon\ \Sigma^k \times (\Sigma_c \cup \Sigma_{dc}^{\leq N}) \rightarrow \Sigma_c^{\leq N}$ *is* (t, ϵ)*-resistant to equal-length target collisions, where* $\epsilon = N\epsilon'$ *and* $t = t' - \Theta(Ndc)$*.*

Proof: We extend and "concretize" the proof sketch in [21, Section 2.3].

Suppose $\mathsf{CF} = (\text{CF-I}, \text{CF-II})$ is an equal length target collision finder which (t, ϵ)-breaks PH^N. We construct an equal length target collision finder $\mathsf{CF}' = (\text{CF-I}', \text{CF-II}')$ which (t', ϵ')-breaks H, as follows:

```
Algorithm CF-I'
$(M, State) \leftarrow$ CF-I
if $|M| = c$ then $x \leftarrow M$ and $i \leftarrow 0$
else
    $n \leftarrow |M|_{dc}$
    $i \xleftarrow{R} \{1, \ldots, n\}$
    $x \leftarrow M_i$
return $(x, (i, M, State))$
```

```
Algorithm CF-II'$(K, x, (i, M, State))$
$M' \leftarrow$ CF-II$(K, M, State)$
if $|M'| = c$ then $x' \leftarrow M'$
else
    $x' \leftarrow M'_i$
return $x'$
```

We must now bound the probability that x, x' is a collision for $H(K, \cdot)$. Suppose M, M' is a collision for $PH^N(K, \cdot)$. We know that $|M| = |M'|$. Thus there are two cases: either $M, M' \in \Sigma_c$ or $M, M' \in \Sigma_{dc}^n$ for some $n \leq N$.

Notice that in the first case, it is impossible for M, M' to be a collision for $PH^N(K, \cdot)$ because $PH^N(K, M) = M$ and $PH^N(K, M') = M'$ and the only way we could have $PH^N(K, M) = PH^N(K, M')$ is when $M = M'$, which is outlawed for collisions. So we can assume we are in the second case.

This means $M, M' \in \Sigma_{dc}^n$ for some $n \leq N$. That M, M' is a collision for $PH^N(K, \cdot)$ means that $H(K, M_j) = H(K, M'_j)$ for all $j = 1, \ldots, n$. But since $M \neq M'$ there is some j such that $M_j \neq M'_j$. With probability $1/n \geq 1/N$ we have $i = j$. So the probability that x, x' is a collision for $H(K, \cdot)$ is at least ϵ/N.

Finally, the running time of CF' is that of CF plus an overhead that amounts to $\Theta(Ndc)$. The result follows. ▌

BASIC TREE HASH. Assume that we wish to hash a message $M = M_1 \ldots M_n$ down to c bits, where each block $M_i \in \Sigma_c$ consists of c bits, and the number of blocks is $n = d^l$ for some $l \geq 1$. We can do the hashing by applying the parallel hash l times. A different key is used for each application of the parallel hash. That is:

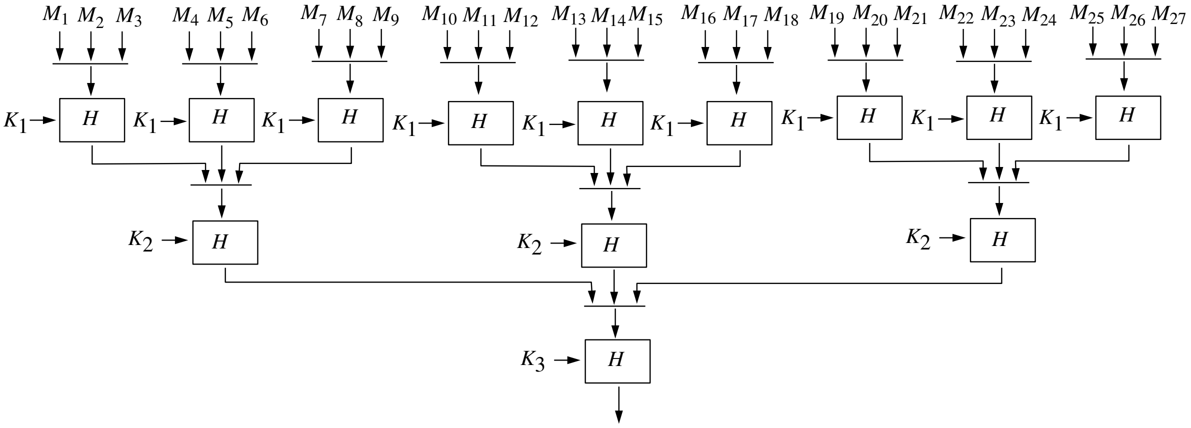

Figure 5: Construction TH — The basic tree scheme, illustrated for the case of $d = 3$, $l = 3$.

```
Algorithm $TH(K_1 \cdots K_l,\ M)$
$Level[0] \leftarrow M$
for $j = 1, \ldots, l$ do
    $Level[j] \leftarrow PH^{d^{l-j}}(K_j,\ Level[j-1])$
return $Level[l]$
```

Visualize this as building a d-ary tree of depth l. The leaves correspond to the message blocks and the root corresponds to the final hash value. Group the nodes at level 0 (the leaves) into runs of size d (that is, each run consists of d blocks, each block being c bits long) and hash each group via $H(K_1, \cdot)$. (This process is represented succinctly in the algorithm as an application of the parallel hash.) This yields d^{l-1} values, each a c-bit block, which form the nodes at level 1 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 $j - 1$ of the tree. At level $l - 1$ we have d nodes, which are hashed under $H(K_l, \cdot)$ to yield the root, which, at level l, is the final hash value. See Figure 5.

As usual, we extend the hash function to allow inputs of various lengths. Assume that all messages we will hash have a number of c-bit blocks which is at most $N = d^\ell$, for some $\ell \geq 1$. For simplicity, further assume that any message M to be hashed has a number of blocks which is a power of d. Then we can define $TH^N\colon\ \Sigma^{\ell k} \times \bigcup_{l=0}^{\ell} \Sigma_c^{d^l} \to \Sigma^c$ by setting $TH^N(K_1 \cdots K_l, M) = TH(K_1 \cdots K_l, M)$, where $l = \log_d(|M|_c)$.

Notice that one key is used for every hash of a given level of the tree, but the key changes across levels. The key length of TH^N is thus $k \cdot \ell = k \cdot \log_d N = k \cdot \log_d(L/c)$ where $L = c \cdot d^\ell$ is the maximum message length.

Notice that this hash family can be viewed as a composition of the parallel hash families, namely

$$TH^{d^\ell} \;=\; PH^{d^0} \circ PH^{d^1} \circ PH^{d^2} \ldots \circ PH^{d^{\ell-1}}\ . \qquad (3)$$

We can now assess the security by applying the composition lemma and the analysis of the security of the parallel hash.

Theorem 5.7 *Suppose* $H\colon\ \Sigma^k \times \Sigma^{dc} \to \Sigma^c$ *is* (t', ϵ')*-resistant to target collisions. Suppose* $N = d^\ell$ *where* $\ell \geq 1$*. Then* $TH^N\colon\ \Sigma^{\ell k} \times \bigcup_{l=0}^{\ell} \Sigma_c^{d^l} \to \Sigma^c$ *is* (t, ϵ)*-resistant to equal-length target collisions, where* $\epsilon = (N-1)\epsilon'/(d-1)$ *and* $t = t' - \Theta(N) \cdot (T_H + k + c)$*.*

Proof: For each $l = 0, \ldots, \ell - 1$, Lemma 5.6 says that PH^{d^l} is (t_l, ϵ_l)-resistant to equal-length collisions, where

$$\begin{aligned} t_l &= t' - \Theta(cd^{l+1}) && (4) \\ \epsilon_l &= d^i \epsilon' \; . && (5) \end{aligned}$$

Note that each PH^{d^l} is length consistent. Now look at Equation (3) and apply Lemma 3.2 ℓ times.

Let's analyze this inductively. Namely say $PH^{d^0} \circ \ldots \circ PH^{d^i}$ is $(\bar{\epsilon}_i, \bar{t}_l)$-resistant to equal-length collisions, $i = 0, \ldots, \ell - 1$. We know $\bar{\epsilon}_0 = \epsilon_0 = \epsilon'$ and $\bar{t}_0 = t_0 = t' - \Theta(dc)$. Now for $l \geq 1$ we view $PH^{d^0} \circ \ldots \circ PH^{d^i}$ as

$$\underbrace{(PH^{d^0} \circ PH^{d^1} \circ \ldots \circ PH^{d^{l-1}})}_{H_2} \circ \underbrace{PH^{d^l}}_{H_1} \; .$$

By Lemma 3.2 we have

$$\begin{aligned} \bar{t}_l &= \min(t_l - k, \bar{t}_{l-1} - 2T_{PH^{d^l}} - k) && (6) \\ \bar{\epsilon}_l &= \epsilon_l + \bar{\epsilon}_{l-1} \; . && (7) \end{aligned}$$

Lets simplify these in turn, beginning with the probability.

We are interested in $\epsilon = \bar{\epsilon}_{\ell-1}$. Applying Equation (7) and Equation (5) we have

$$\epsilon \;=\; \epsilon_0 + \ldots + \epsilon_{\ell-1} \;=\; (d^0 + \ldots + d^{\ell-1})\epsilon' \;=\; \frac{d^\ell - 1}{d-1} \cdot \epsilon' \;=\; \frac{N-1}{d-1} \cdot \epsilon' \; .$$

Now we want to compute $t = \bar{t}_{\ell-1}$. We start from Equation (6) and try to get a simpler expression for $\bar{t}_i$. It must be that $\bar{t}_{l-1} \leq t'$. Using this and Equation (4) we have

$$\begin{aligned} \bar{t}_l &= \min(t_l - k, \bar{t}_{l-1} - 2T_{PH^{d^l}} - k) \\ &\geq \bar{t}_{l-1} - \Theta(d^{l+1}c) - 2T_{PH^{d^l}} - k \\ &= \bar{t}_{l-1} - \Theta(d^{l+1}c) - 2d^i T_H - k \; . \end{aligned}$$

This means

$$\begin{aligned} \bar{t}_{\ell-1} &\geq \bar{t}_0 - \sum_{l=1}^{\ell-1} (\Theta(d^{l+1}c) + 2d^l T_H + k) \\ &\geq t' - \Theta(dc) - \sum_{l=1}^{\ell-1} (\Theta(d^{l+1}c) + 2d^l T_H + k) \\ &\geq t' - \Theta(d^\ell) \cdot (c + k) - 2d^\ell T_H \; . \end{aligned}$$

Thus we can set t as in the theorem statement, and the result follows. ▮

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 (to hash an nc-bit message) is $k \cdot \log_d(n)$, which can be large, because k 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.

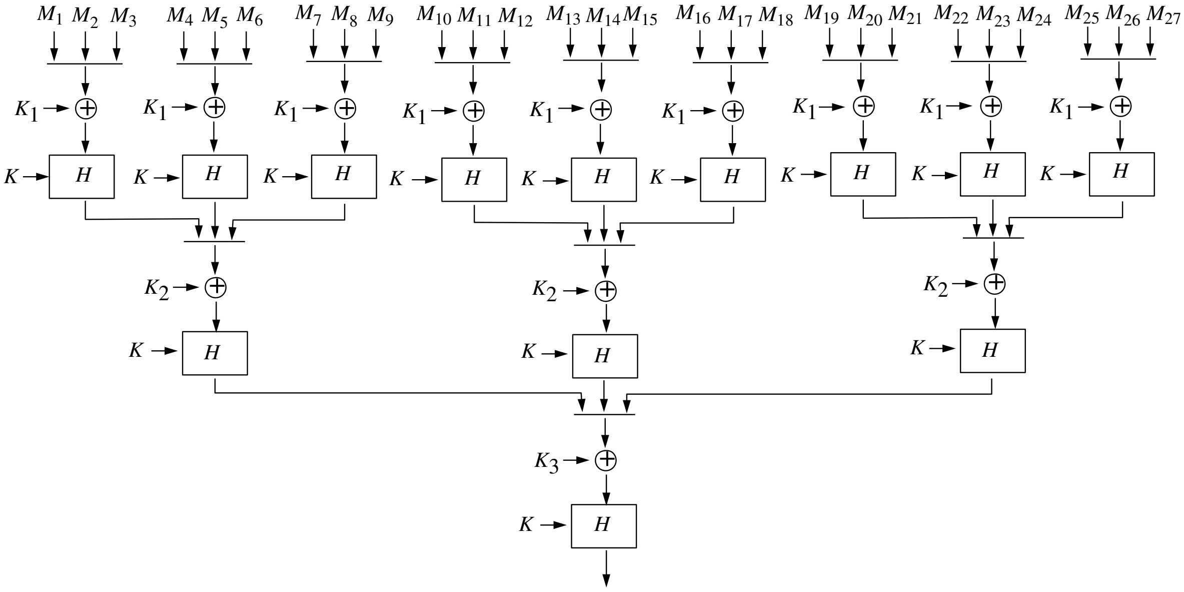

Figure 6: Construction XTH — The XOR tree scheme, illustrated for $m = 3$ an $d = 3$. Notice that H is always keyed with K, while an auxiliary key, differing for each level, is XORed just before the application of H_K.

Specifically, to hash a message $M \in \Sigma_c^{d^l}$ $(l \leq \ell)$ we use one key $K \in \Sigma^k$ for the compression function (this is called the primary key) and auxiliary keys $K_1, \ldots, K_l \in \Sigma^{dc}$. Before describing the algorithm we need some notation. Namely for a string X and integer $j \geq 1$ let

$$X^{(j)} = \underbrace{X \| \cdots \| X}_{j}$$

denote the string formed by concatenating j copies of X.

We hash a message $M = M_1 \cdots M_n$, where $n = d^l$ and $|M_j| = c$, as follows:

```
**Algorithm** $XTH(K\,K_1 \ldots K_l,\ M)$
$Level[0] \leftarrow M$
**for** $j = 1, \ldots, l$ **do**
    $Level^*[j-1] \leftarrow Level[j-1] \oplus K_j^{(d^{l-j})}$
    $Level[j] \leftarrow PH^{d^{l-j}}(K,\ Level^*[j-1])$
**return** $Level[l]$
```

In other words, a d^l block message $M = Level[0]$ is hashed in l stages, resulting in strings $Level[0] \xrightarrow{1} Level[1] \xrightarrow{2} \cdots \xrightarrow{l} Level[l]$. The last of these is the hash of M. Each stage cuts the message size by a factor of d. Stage j begins by XORing to $Level[j-1]$ a sufficient number of copies of K_j. Then it applies the parallel hash to cut the length. Note that all applications of the parallel hash are under the *same* key K.

For any $d \geq 2$ and $\ell \geq 1$ we define a family of hash functions $XTH^N\colon\ \Sigma^{k+\ell dc} \times \bigcup_{l=0}^{\ell} \Sigma_c^{d^l} \to \Sigma^c$ by $XTH^{d^\ell}(KK_1 \ldots K_\ell, M) = XTH(KK_1 \ldots K_l, M)$ for any $K \in \Sigma^k$, any $K_1, \ldots, K_\ell \in \Sigma^{dc}$, any $l \in \{0, \ldots, \ell\}$ and any $M \in \Sigma_c^{d^l}$.

Once again one can again the construction as a d-ary tree of depth l. The leaves correspond to the message blocks and the root corresponds to the final hash value. Group the nodes at level 0 (the leaves) into runs of size d, XOR each group with the auxiliary key K_1, and then hash each group via $H(K,\cdot)$. This yields a d^{l-1} values, each a c-bit block, which form the nodes at level 1 of the tree. Now continue the process. At each level we use a different auxiliary key but the same hash function. At level $l-1$ we have d nodes, which are XORed with K_l and hashed with $H(K,\cdot)$ to yield the root, which, at level l, is the final hash value. See Figure 6.

Note key length to hash a nc bit message is $k + dc \cdot \log_d(n)$. For example, when $d = 2$, $c = 64$, the resulting key length of $k + 128 \lg n$ 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 [17].

We now proceed with the analysis. Suppose $\alpha \in \{0, \ldots, l-1\}$ is a level of the tree. There are $d^{l-\alpha}$ nodes at this level, divided into $d^{l-\alpha-1}$ groups of d. For $M \in \Sigma_c^{d^l}$ we will use the notation

$$M[\alpha,\beta] \;=\; M_{(\beta-1)d^\alpha+1} \ldots M_{\beta d^\alpha}$$

to describe the part of the message M which hashes to the β-th group of nodes at level α, where $\beta \in \{1, \ldots, d^{l-\alpha-1}\}$. This means that $XTH^{d^\alpha}(K\,K_1 \ldots K_\alpha, M[\alpha,\beta])$ is the β-th node at level α of the tree.

For $l \geq 1$ and $M \in \Sigma_c^{d^l}$ it is convenient to define

```
**Algorithm** $XTHI(KK_1 \ldots K_{l-1}, M)$
$Level[0] \leftarrow M$
**for** $j = 1, \ldots, l-1$ **do**
    $Level^*[j-1] \leftarrow M[j-1] \oplus K_j^{(d^{l-j})}$
    $Level[j] \leftarrow PH^{d^{l-j+1}}(K, Level^*[j-1])$
**return** $Level[l-1]$
```

In other words, do all but the last stage of the XOR tree hash. This means the output $Level[l-1]$ is a member of Σ_c^d. Now $XTH^{d^l}(KK_1 \ldots K_l, M) = PH^d(Level[l-1] \oplus K_l^{(d)})$. But this last parallel hash is just the compression function $H(K,\cdot)$, so that we have the relation

$$XTH(KK_1 \ldots K_l, M) \;\;=\;\; H(\,K\,,\, XTHI(KK_1 \ldots K_{l-1}, M) \oplus K_l^{(d)}\,)\,. \tag{8}$$

We will use this later.

We can no longer appeal to the composition lemma in proving security, because the different parallel hashes use a common key K. Instead we give a direct proof of security.

Theorem 5.8 *Suppose* H: $\Sigma^k \times \Sigma^{dc} \to \Sigma^c$ *is* (t', ϵ')*-resistant to target collisions. Suppose* $N = d^\ell$ *where* $\ell \geq 1$. *Then* XTH^N: $\Sigma^{k+\ell dc} \times \bigcup_{l=0}^{\ell} \Sigma_c^{d^l} \to \Sigma^c$ *is* (t,ϵ)*-resistant to equal-length target collisions, where* $\epsilon = (N-1)\epsilon'/(d-1)$ *and* $t = t' - \Theta(N) \cdot (T_H + dc + k)$.

Proof: Suppose $M, M' \in \Sigma_c^{d^l}$ is a collision for $XTH^{d^l}(KK_1 \cdots K_\ell, \cdot)$. We observe that there is then a level $\alpha \in \{1, \ldots, l\}$ of the tree, and a $\beta \in \{1, \ldots, d^{l-\alpha-1}\}$, for which

$$\left\{ \begin{array}{rcl} XTH^{d^l}(K_1 \cdots K_\alpha, M[\alpha,\beta]) & = & XTH^{d^l}(K_1 \cdots K_\alpha, M'[\alpha,\beta]) \\ XTHI(K_1 \cdots K_{\alpha-1}, M[\alpha,\beta]) & \neq & XTHI(K_1 \cdots K_{\alpha-1}, M'[\alpha,\beta])\,. \end{array} \right. \tag{9}$$

This can be seen by reverse induction on the tree level, beginning with the fact that $XTH^{d^l}(KK_1 \cdots K_\ell, M) = XTH^{d^l}(K_1 \cdots K_\ell, M')$. In combination with Equation (8) this tells us how to find collisions for $H(K,\cdot)$ given collisions in $XTH^{d^l}(K\,K_1 \cdots K_\ell, \cdot)$. We will exploit this below.

Suppose $\mathsf{CF} = (\text{CF-I}, \text{CF-II})$ is a target collision finder which (t,ϵ)-breaks XTH^N. We construct a target collision finder $\mathsf{CF}' = (\text{CF-I}', \text{CF-II}')$ which (t',ϵ')-breaks H, as follows:

```
**Algorithm** CF-I'
$x \overset{R}{\leftarrow} \Sigma^{dc}$
**return** $x$
```

```
**Algorithm** CF-II'$(K, x)$
$(M, \mathit{State}) \overset{R}{\leftarrow}$ CF-I and $l \leftarrow \log_d(|M|_c)$
$r \overset{R}{\leftarrow} \{1, \ldots, l\}$ and $j \overset{R}{\leftarrow} \{1, \ldots, d^{l-r}\}$
$K_1, \ldots, K_{r-1}, K_{r+1}, \ldots, K_\ell \overset{R}{\leftarrow} \Sigma^{dc}$
$z \leftarrow XTHI(KK_1 \ldots K_{r-1}, M[r, j])$
$K_r \leftarrow z \oplus x$
$M' \leftarrow$ CF-II$(K\ K_1 \cdots K_\ell, M, \mathit{State})$
$x' \leftarrow K_r \oplus XTHI(K_1 \cdots K_{r-1}, M'[r, j])$
**return** $x'$
```

The target message finding algorithm CF-I is very simple: it just outputs some random string x of length dc. The sibling finder CF-II begins by letting M be the target message output by CF-I. It then picks a tree level $r \in \{1, \ldots, l\}$ at random. Recall that at level r we have d^{l-r} nodes. We group them into groups of size d, so that we view them as forming a set of d^{l-r-1} strings, each dc bits long. After XORing each of these with K_r we get the inputs to the compression function for this stage. Our goal is to make x one of these inputs, namely x should be K_r XORed with one of the groups at level r. The key idea is that K_r will be chosen as a function of x to make this happen. How? CF-II' picks $j \in \{1, \ldots, d^{l-r}\}$ at random and sets z, K_r as indicated in the code. Notice that since x was chosen randomly and independently of anything else, the keys $K_1, \ldots, K_\ell$ are all random and independent of each other. Now CF-II' gives key $K\ K_1 \cdots K_\ell$ to CF-II, along with *State* as state information. CF-II outputs a message M'. (We know that $|M| = |M'|$. Also, if we are lucky, M, M' is a collision for $XTH^{d^l}(K\ K_1 \cdots K_\ell, \cdot)$, and we proceed under this assumption.) CF-II' computes, for this message, the value at the same node as before, namely x', and outputs this.

We must now bound the probability that x, y is a collision for $H(K, \cdot)$. We use Equation (9). The number of possibilities for (α, β) is at most $d^0 + \cdots + d^{\ell-1} = (d^\ell - 1)/(d-1)$. Since r, j were chosen at random we have probability at least $(d-1)/(d^\ell - 1)$ that $(r, j) = (\alpha, \beta)$. So the probability that x, y is a collision for $H(K, \cdot)$ is $\epsilon' \geq \epsilon(d-1)/(d^\ell - 1) \geq \epsilon(d-1)/(n-1)$. The time bounds can be verified by looking at the pseudocode. ▌

6 Message Lengths

The constructions and results in Section 5 make two restrictions we will now indicate how to remove. First, we proved security against equal-length target collisions. In practice one requires security against variable-length target collisions. Second, we assumed message lengths are multiples of some fixed number, like a block size, or even a power of some fixed number, like in the tree schemes. In reality any length should be allowed.

We begin by showing how to extend a TCR hash function secure against equal-length collisions into a TCR hash function secure against variable-length collisions. Then we will see how to handle strings of any length.

6.1 Length Variability

Suppose we have a hash function secure against equal-length collisions. We want to address input-length variability, meaning make it secure against variable-length 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 $|M|$. For example, say the block length is 512. One might append a "1" to the message, and then the minimal number of zeros so that the length becomes 64 bits shy of a multiple of 512 bits, and then append $|M|_{64}$ — the length of M, encoded as a 64-bit binary number (assuming $|M| < 2^{64}$). (This is the padding method used in [25] and many other hash function.) Let $Pad(\cdot)$ denote such a padding function. 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 input-length variability. Namely, we first hash the message using one key. Then we concatenate the message length to the result, and hash again, using a second key. The second hashing typically requires just one extra application of the compression function, since we are hashing a small, fixed length message. If the hash functions used are secure against equal-length target collisions, the result is secure against variable-length target collisions.

Theorem 6.1 *Fix* $m > 0$ *and let* $Msgs_1$ *be a set of strings each of length less than* 2^m. *Let* $H_1\colon \Sigma^{k_1} \times Msgs_1 \to \Sigma^{l_1}$ *and* $H_2\colon \Sigma^{k_2} \times \Sigma^{l_1+m} \to \Sigma^c$ *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* $H\colon \Sigma^{k_1+k_2} \times Msgs_1 \to \Sigma^c$ *by*

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

where $\langle |M| \rangle_m$ *is the length of* M *written as a string of exactly* m *bits,* $M \in Msgs_1$, $K_1 \in \Sigma^{k_1}$, *and* $K_2 \in \Sigma^{k_2}$. *Then* H *is* (t, ϵ)*-secure against variable-length target collisions, where* $t = \min(t_1 - k_2,\ t_2 - k_1 - 2T_{H_1} - 2l_1 - 2)$ *and* $\epsilon = \epsilon_1 + \epsilon_2$.

Proof: Let $\mathsf{CF} = (\text{CF-I}, \text{CF-II})$ be a target collision finder for H which runs in time t. Consider the experiment describing CF's attack on H, namely

$$(M, State) \leftarrow \text{CF-I}\,;\; K_1 \overset{R}{\leftarrow} \Sigma^{k_1}\,;\; K_2 \overset{R}{\leftarrow} \Sigma^{k_2}\,;\; M' \leftarrow \text{CF-II}(K_1K_2, M, State)\;. \tag{10}$$

Let $x = H_1(K_1, M)$ and $x' = H_1(K_1, M')$. Now let E_1 be following event: CF is successful and $x = x'$ and $|M| = |M'|$. Let E_2 be the following event: CF is successful, and either $x \neq x'$ or $|M| \neq |M'|$. Let $p_1 = \Pr[E_1]$ and $p_2 = \Pr[E_2]$, the probabilities being under the experiment of Equation (10). Notice that E_1, E_2 are disjoint events with union the event that CF is successful, so we have $ProbSuccess(\mathsf{CF}, H) = p_1 + p_2$. Thus it suffices to upper bound p_1, p_2.

We do this by defining a target collision finder $\mathsf{CF}_1 = (\text{CF-I}_1, \text{CF-II}_1)$ for H_1 and a target collision finder $\mathsf{CF}_2 = (\text{CF-I}_2, \text{CF-II}_2)$ for H_2 so that $ProbSuccess(\mathsf{CF}_1, H_1) = p_1$ and $ProbSuccess(\mathsf{CF}_2, H_2) = p_2$. We make sure that the running time of CF_1 is at most t_1 and that of CF_2 is at most t_2. Out assumptions about the security of H_1, H_2 then imply that $p_1 \leq \epsilon_1$ and $p_2 \leq \epsilon_2$, so that $ProbSuccess(\mathsf{CF}, H) \leq \epsilon_1 + \epsilon_2$.

It remains to define the two algorithms. They are:

```
Algorithm CF-I$_1$
$(M, State) \overset{R}{\leftarrow}$ CF-I
return $(M, State)$
```

```
Algorithm CF-II$_1(K_1, M, State)$
$K_2 \overset{R}{\leftarrow} \Sigma^{k_2}$
$M' \overset{R}{\leftarrow}$ CF-II$(K_1K_2, M, State)$
return $M'$
```

```
Algorithm CF-I$_2$
$(M, State) \overset{R}{\leftarrow}$ CF-I and $K_1 \overset{R}{\leftarrow} \Sigma^{k_1}$
$x \leftarrow H_1(K_1, M)$
$y \leftarrow x \,\|\, |M|_m$
return $(y, (M, State, K_1))$
```

```
Algorithm CF-II$_2(K_2, y, (M, State, K_1))$
$M' \overset{R}{\leftarrow}$ CF-II$(K_1K_2, M, State)$
$x' \leftarrow H_1(K_1, M')$
$y' \leftarrow x' \,\|\, |M'|_m$
return $y'$
```

For the analysis, first consider the experiment describing CF_1's attack on H_1. Let $x = H_1(K_1, M)$ and $x' = H_1(K_1, M')$. CF_1 is successful in breaking H_1 when $|M| = |M'|$ but $M \neq M'$ and $x = x'$. Notice $x = x'$ and $|M| = |M'|$ implies $H_2(K_2, x \,\|\, |M|_m) = H_2(K_2, x' \,\|\, |M'|_m)$ where K_2 is the key chosen by CF-II$_1$. This means that if M, M' is an equal length collision for H_1 then it will also be an equal length collision for H. This means M, M' is an equal length collision for $H_1(K_1, \cdot)$ with exactly the probability that event E_1 occurs in the experiment of Equation (12). It follows that $\mathit{ProbSuccess}(\mathsf{CF}_1, H_1) = p_1$.

Now consider the experiment describing CF_2's attack on H_2. By definition CF_2 is successful in breaking H_2 when $|y| = |y'|$ but $y \neq y'$ and $H_2(K_2, y) = H_2(K_2, y')$. But we always have $|y| = |y'|$ because the domain of H_2 only contains strings of a fixed length, namely $l_1 + m$. Since $y = x \,\|\, |M|_m$ and $y' = x' \,\|\, |M'|_m$ we have $y \neq y'$ if either $x \neq x'$ or $|M| \neq |M'|$. This means y, y' is a collision for $H_2(K_2, \cdot)$ with exactly the probability that event E_2 occurs in the experiment of Equation (10). It follows that $\mathit{ProbSuccess}(\mathsf{CF}_2, H_2) = p_2$.

It remains to bound the running times. That of CF_1 is $t + k_2$ and this is at most t_1 for the choice of t in the lemma statement. That of CF_2 is $t + k_2 + 2T_{H_1} + 2m$ which by the choice of t_2 in the lemma statement is at most t_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 $LH \circ \mathit{pad}$ is insecure against variable-length target collisions. For XLH the answer is *yes*: if H is a TCR compression function then $XLH \circ \mathit{pad}$ is guaranteed to be secure against target collisions; one can appropriately modify the proof of Theorem 5.5 to show this. We did not investigate the analogous questions for TH and XTH.

6.2 Padding

Combining the methods of Sections 5 and 6.1 we have constructions for TCR hash functions which are secure against variable-length collisions on a domain that has "gaps"— our domains only include strings that have length a multiple of some block length, or even, in the case of the tree schemes, a power of some integer d. To wrap things up we must eliminate the restriction that lengths are of some particular values. Simple padding schemes work fine. This is shown by the following result.

Theorem 6.2 *Let* $\ell_1 < \cdots < \ell_{\max}$ *be numbers,* $\mathit{Msgs} = \bigcup_{i=1}^{\max} \Sigma^{\ell_i}$, $\mathrm{MAX} < \ell_{\max}$, *and* $\mathit{Msgs}_* = \Sigma^{\leq \mathrm{MAX}}$. *Let* $\mathit{Pad} : \mathit{Msgs}_* \to \mathit{Msgs}$ *be a length consistent injective function with inverse,* Unpad*, computable in time* T_{Unpad}. *Suppose* $H : \Sigma^k \times \mathit{Msgs} \to \Sigma^c$ *and define* $H_* : \Sigma^k \times \mathit{Msgs}_* \to \Sigma^c$ *by* $H_*(M) = H(\mathit{Pad}(M))$. *Suppose* H *is* (t, ϵ)*-resistant to equal-length target collisions. Then* H_* *is* (t_*, ϵ_*)*-resistant to equal-length target collisions, where* $t_* = t - 2T_{\mathrm{Unpad}}$ *and* $\epsilon_* = \epsilon$.

Proof: Let $\mathsf{CF}_* = (\text{CF-I}_*, \text{CF-II}_*)$ be a target collision finder for H_* which runs in time t_*. We define a target collision finder $\mathsf{CF} = (\text{CF-I}, \text{CF-II})$ for H as follows:

```
Algorithm CF-I
$(M_*, State) \xleftarrow{R}$ CF-I$_*$
$M \leftarrow Unpad(M_*)$
return $(M, State)$
```

```
Algorithm CF-II$(K, M, State)$
$M'_* \xleftarrow{R}$ CF-II$(K, M, State)$
$M' \leftarrow Unpad(M'_*)$
return $M'$
```

Because Pad is injective and length-consistent each collision (M_*, M'_*) found by CF_* yields a collision (M, M') found by CF of equal-length strings in the domain of H. Thus $\mathit{ProbSuccess}(\mathsf{CF}, H) =$

$\mathit{ProbSuccess}(\mathsf{CF}_*, H_*)$ and we have justified the claimed value for choice of ϵ_*. It remains to look at the running time of adversary CF. This is just $t + 2T_{\text{Unpad}}$, from which the theorem follows. ▌

7 Signing with a TCR Hash Function

Consider the RSA signature primitive [27], where one signs the number $x \in \mathsf{Z}_N$ by $\mathit{SignRSA}_{d,N}(x) = x^d \bmod N$, for appropriately chosen numbers d, N. The usual practice, dating back to [32], is to compute the signature s of a string M according to $s = \mathit{SignRSA}_{d,N}(h(M))$, where h is some sort of hash function.

When signing as above there are actually two unrelated reasons for using the hash function h. The first reason is to map the (infinite or enormous) space *MSGS* of strings that we may wish to sign down to the (small) space *Msgs* of strings that our primitive knows how to handle. (For example, one might have $\mathit{Msgs} \subseteq \Sigma^{1024}$ if one is using *SignRSA*.) The second reason for applying h is to help mask algebraic structure in the underlying cryptographic primitive. In particular, *SignRSA* does not, by itself, have the properties one expects of a secure signature scheme, due to its algebraic structure— and yet $\mathit{SignRSA} \circ h$ seems to be a good way to sign when the hash function h is chosen well.

In the current work we are only concerned with reducing lengths, not in covering up algebraic properties of the underlying primitive. Thus we will *assume* that we already have in hand a secure signature scheme. Examples of such schemes are [5, 15, 7]. (The first requires ideal hash functions, aka random oracles, in addition to the assumption that RSA is one-way, while the second and third require only the assumption that RSA is one-way, but are less efficient. There are also, of course, many more schemes, but these are less efficient still.) We imagine that the only problem with *Sign* is its small domain, *Msgs*, and we simply want to enlarge the domain to make a function *Sign* which can sign messages on all of *MSGS*. The domain should be either $\mathit{MSGS} = \Sigma^*$ or $\mathit{MSGS} = \Sigma^{\leq \ell}$ for some enormous number ℓ.

It is a folklore result that if $h : \mathit{MSGS} \to \Sigma^c$ is a randomly selected hash function from an ACR family of hash functions, and if *Sign* is a secure signing function with domain Σ^c, then $\mathit{SIGN} = \mathit{Sign} \circ h$ provides a secure way to sign messages on the domain *MSGS*.

Here we extend the above approach to use TCR hash function. First we will need some basic definitions on signatures and their security.

SYNTAX OF SIGNATURE SCHEMES. A *digital signature scheme*, $(\mathit{Gen}, \mathit{Sign}, \mathit{Verify})$, consists of a *key generation algorithm*, a *signing algorithm*, and a *verifying algorithm*. The first of these algorithms will always be probabilistic; the second algorithm might or might not be; the third algorithm is always deterministic. A digital signature scheme has an associated message space, *Msgs*, where $\mathit{Msgs} \subseteq \Sigma^*$. The key generation algorithm flips coins and outputs a matching public and secret key, $(\mathit{pk}, \mathit{sk}) \overset{R}{\leftarrow} \mathit{Gen}()$. The signing algorithm takes a message $M \in \mathit{Msgs}$ and a secret key sk and it returns a signature $s \overset{R}{\leftarrow} \mathit{Sign}_{\mathit{sk}}(M)$. The verifying algorithm takes a message M, a candidate signature s', and the public key pk, and it returns a bit $\mathit{ok} \leftarrow \mathit{Verify}_{\mathit{pk}}(M, s')$, with 1 signifying "accept" and 0 signifying "reject." We demand that if s was produced via $s \overset{R}{\leftarrow} \mathit{Sign}_{\mathit{sk}}(M)$ then $\mathit{Verify}_{\mathit{pk}}(M, s) = 1$. We let T_{Gen} denote the worst case time for *Gen* to produce a pair $(\mathit{pk}, \mathit{sk})$ and we let $T_{\mathit{Sign}}(m)$ be the worst case time to compute $\mathit{Sign}_{\mathit{sk}}(M)$ for $M \in \mathit{Msgs} \cap \Sigma^{\leq m}$. We write interchangeably $\mathit{Sign}_{\mathit{sk}}(M)$ and $\mathit{Sign}(\mathit{sk}, M)$.

SECURITY OF SIGNATURE SCHEMES. Definitions for the security of signatures in an asymptotic setting were provided by Goldwasser, Micali and Rivest [16]. Concrete security definitions were provided in [5]. We follow the latter.

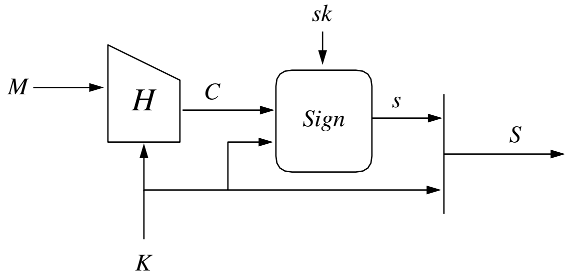

Figure 7: How to extend a signing primitive to a larger domain using a TCR hash function. Key K is chosen anew for each message. The signature of M is $S = K \| s$.

A *forgery finder* FF takes as input a public key pk, and FF tries to forge a signature with respect to pk. To do this it is allowed a chosen message attack. This means that FF can request and obtain signatures of any messages it wants. This is modeled by providing FF with oracle access to the signing algorithm. The forgery finder is deemed *successful* if it outputs a *valid forgery*— a message/signature pair (M, s) such that $\mathit{Verify}_{pk}(M, s) = 1$ and yet M was not a message of which a signature was requested of the signing oracle. The forgery finder FF is said to be a (t, q, μ)-forgery finder if its running time (including the description size of FF, as per our conventions) is at most t, and FF makes at most q queries of its signing oracle, and the length of these queries, as well as the length of the strings (M, s) output by FF, is at most μ. time to answer the signing queries. Such a forgery finder FF is said to (t, q, μ, ϵ)-break the signature scheme if the probability that FF outputs a valid forgery is at least ϵ. The probability is over the random choices of FF as well as the random choices of *Gen* and *Sign*. We say that the signature scheme is (t, q, μ, ϵ)-secure if there is no forgery finder FF which (t, q, μ, ϵ)-breaks it.

SIGNING WITH AN TCR HASH FAMILY — BASIC METHOD. Let $(\mathit{Gen}, \mathit{Sign}, \mathit{Verify})$ be a signature scheme having associated message space *Msgs*. We want to extend this to a signature scheme $(\mathit{GEN}, \mathit{SIGN}, \mathit{VERIFY})$ with an associated (larger) message space *MSGS*. We desire a method with the simplicity of $\mathit{SIGN} = \mathit{Sign} \circ h$, yet we want to avoid the use of an ACR hash family.

Assuming that $\Sigma^c \subseteq \mathit{Msgs}$ for some constant c, one might first try letting H be TCR and using the same scheme sketched above. Namely, fix a random key $K \in \Sigma^k$, let $h = H_K$, and sign M by $s \overset{R}{\leftarrow} \mathit{Sign}_{sk}(h(M))$. The key K is a public constant associated to the signature scheme. This approach works for $H : \Sigma^k \times \mathit{MSGS} \to \Sigma^c$ being ACR but it does not work for H being TCR. The reason is simple: in an adaptive chosen message attack the adversary, knowing K, may be able to find two messages, M and M', which collide under H_K. If so, the adversary asks the signing oracle for the signature of M and from this the adversary immediately knows a valid forgery for M'.

Instead, the signing algorithm can choose K anew for each message. The key K is included with the signature; it is not secret. We have to adjust slightly the domain *Msgs*; now we need that $\Sigma^{k+c} \subseteq \mathit{Msgs}$. Here, formally, is the signature scheme. It is pictured in Figure 7.

```
Algorithm $GEN$
$(pk, sk) \overset{R}{\leftarrow} Gen()$
return $(pk, sk)$
```

```
Algorithm $SIGN_{sk}(M)$
$K \overset{R}{\leftarrow} \Sigma^k$
$s \overset{R}{\leftarrow} Sign(K \| H_K(M))$
return $(K, s)$
```

```
Algorithm $VERIFY_{pk}(M, (K, s))$
$ok \leftarrow Verify_{pk}(H_K(K \| M), s)$
return $ok$
```

We then have the following theorem.

Theorem 7.1 *Let* $(\mathit{Gen}, \mathit{Sign}, \mathit{Verify})$ *be a* $(t_1, q_1, \mu_1, \epsilon_1)$*-secure signature scheme with associated domain* $\mathit{Msgs} \subseteq \Sigma^*$. *Let* $H : \Sigma^k \times \mathit{MSGS} \to \Sigma^c$ *be a family of hash function which is* (t_2, ϵ_2)*-resistant* *to target collisions. Assume* $\Sigma^{k+c} \subseteq \mathit{Msgs}$*. Then the signature scheme* $(\mathit{GEN}, \mathit{SIGN}, \mathit{VERIFY})$ *constructed from* $(\mathit{Gen}, \mathit{Sign}, \mathit{Verify})$ *and* H *is* (t, q, μ, ϵ)*-secure, where* $t = \min\{t_1 - (q+1)T_{H,\mu_1} - qT_{\mathit{Sign}}(k+c) - O(k+c),\ t_2 - (q+1)T_{H,\mu_1} - T_{\mathit{Gen}} - qT_{\mathit{Sign}}(k+c) - O(k+c)\}$, $q = q_1$, $\mu = \mu_1 - c - k$, *and* $\epsilon = \epsilon_1 + q_1\epsilon_2$*.*

Proof: Let FF be a (t, q, μ, ϵ)-forgery finder for $(\mathit{GEN}, \mathit{SIGN}, \mathit{VERIFY})$. We wish to bound ϵ Consider the experiment defining FF's attack, namely

$$(pk, sk) \xleftarrow{R} \mathit{GEN}() \ ; \ (M, (K, s)) \xleftarrow{R} \mathsf{FF}^{\mathit{Sign}(sk,\cdot)}(pk) \ . \tag{11}$$

Suppose FF asks its oracle $M_1, \ldots, M_q$, obtaining responses $(K_1, s_1), \ldots, (K_q, s_q)$, respectively. Let E be the event that $(M, (K, s))$ is a valid forgery. Let E_1 be the event that $(M, (K, s))$ is a valid forgery and $K \notin \{K_1, \ldots, K_1\}$, or else $(M, (K, s))$ is a valid forgery and $K = K_i$ for some $i \in \{1, \ldots, q\}$ and for every $i \in \{1, \ldots, q\}$ for which $K = K_i$ we have that $H_K(M) \neq H_K(M_i)$. Let E_2 be the event that $(M, (K, s))$ is a valid forgery and for some $i \in \{1, \ldots, q\}$ we have that $K = K_i$ and $H_K(M) = H_K(M_i)$. Let $p_1 = \Pr[E_1]$ and let $p_2 = \Pr[E_2]$. Then E is the union of disjoint events E_1 and E_2, and so $\epsilon = p_1 + p_2$. We will thus upper bound ϵ by upper bounding p_1 and p_2.

First let us upper bound p_1. Using FF, which attacks $(\mathit{GEN}, \mathit{SIGN}, \mathit{VERIFY})$, we construct a forgery finder ff, which attacks $(\mathit{Gen}, \mathit{Sign}, \mathit{Verify})$, as follows:

```
Algorithm $\mathsf{ff}^{Sign(sk,\cdot)}(pk)$
Run $\mathsf{FF}(pk)$
When $\mathsf{FF}$ makes its $i$th oracle query, $M_i$,
    $K_i \xleftarrow{R} \Sigma^k$
    $C_i \leftarrow H(K_i, M_i)$
    Use $\mathsf{ff}$'s oracle to obtain $s_i \xleftarrow{R} Sign(sk, K_i \| C_i)$
    Respond to $\mathsf{FF}$'s query with $(K_i, s_i)$
When $\mathsf{FF}$ outputs $(M, (K, s))$, output $(K \| H_K(M), s)$
```

When pk is sampled according to $(pk, sk) \xleftarrow{R} \mathit{Gen}()$ the adversary ff creates for the FF which it runs an environment identical to that corresponding to Equation (11). From this and the definition of event E_1, the probability that ff succeeds in forgery is at least p_1. The time t_1 which ff requires is at most $t_1 = t + (q+1)T_{H,\mu} + O(k+c)$. The number of queries q_1 made by ff is precisely $q_1 = q$. Queries asked by ff are of length $c + k$, while the strings output by ff have length at most $\mu + k$. Consulting the bounds for t, q and μ in the theorem statement we conclude that $p_1 \leq \epsilon_1$.

Next we upper bound p_2. To do this we construct a collision finder $\mathsf{CF} = (\text{CF-I}, \text{CF-II})$ for H:

```
**Algorithm** CF-I
$j \xleftarrow{R} \{1, \ldots, q\}$
$(pk, sk) \xleftarrow{R} Gen()$
Run $\mathsf{FF}(pk)$
When $\mathsf{FF}$ makes its $i$th oracle query, $M_i$,
    $K_i \xleftarrow{R} \Sigma^k$
    $C_i \leftarrow H(K_i, M_i)$
    $s_i \xleftarrow{R} Sign(sk, K_i \| C_i)$
    if $i = j$ then
        Let $\mathsf{FF}$-$state$ = the state of $\mathsf{FF}$
        Output $(M_i, \mathsf{FF}\text{-}state)$ and halt
    Respond to $\mathsf{FF}$'s query with $(K_i, s_i)$
```

```
**Algorithm** CF-II$(K, M, \mathsf{FF}\text{-}state)$
Continue running $\mathsf{FF}$, starting in $\mathsf{FF}$-$state$
When $\mathsf{FF}$ makes its $i$th oracle query, $M_i$,
    if $i = j$
        then $K_i \leftarrow K$
        else $K_i \xleftarrow{R} \Sigma^k$
    $C_i \leftarrow H(K_i, M_i)$
    $s_i \xleftarrow{R} Sign(sk, K_i \| C_i)$
    Respond to $\mathsf{FF}$'s query with $(K_i, s_i)$
When $\mathsf{FF}$ outputs $M'$, output $M'$ and halt
```

Collision finder CF creates for FF an environment identical to that corresponding to Equation (11). From this and the definition of event E_2, the probability that ff succeeds in forgery must be at least p_2/q. This is because every time that FF forges with a pair of points $(M, (K, s))$ and $(M', (K, s))$ where $H_K(M) = H_K(M')$ there is a $1/q$ chance that the key K given as input to CF-II was the key with respect to which the forgery was accomplished. The time t_2 which CF requires is at most $t_1 = t + (q+1)T_{H,\mu} + T_{Gen} + qT_{Sign}(k+c) + O(k+c)$. Consulting the bounds for t in the theorem statement we conclude that $p_2/q \leq \epsilon_2$.

Putting our results together we have that $\epsilon \leq p_1 + p_2 \leq \epsilon_1 + q\epsilon_2$ for the given t, q, μ. ▮

HANDLING LONG KEYS. One potential difficulty in using the above approach is that the signature primitive *Sign* might have a domain *Msgs* too small to accommodate (the hash of message M together with) the entire hash key K. This could happen if hash keys are quite long. When using an *ad. hoc.* construction of the sort discussed in Section 4 this will not be a problem, for such cases the key length will be small and independent of the message length. But suppose we are using XOR trees, for example. Then the length of the key grows logarithmically with $m = |M|$. If M is long then $|K|$ might get too big to fit (along with C) in the scope of *Sign*.

To handle this possibility we can hash multiple times. Suppose we start with a long message M, where $m = |M|$. Use a TCR hash family $H_1 : \Sigma^{k_1} \times \Sigma^m \rightarrow \Sigma^c$. If one is using XOR trees, say, then k_1 is $O(\lg m)$. Choose $K_1 \xleftarrow{R} \Sigma^{k_1}$ and compute $C_1 \leftarrow H_{K_1}(M)$. If $k_1 + c$ is too long to for *Sign* then let $H_2 : \Sigma^{k_2} \times \Sigma^{k_1+c} \rightarrow \Sigma^c$ and pick a random key $K_2 \xleftarrow{R} \Sigma^{k_2}$. Assuming now that $\Sigma^{k_2+c} \subseteq Msgs$, the signature of M is defined as $K_1 \| K_2 \| Sign_{sk}(K_2 \| H_2(K_2, H_1(K_1, M)))$. See Figure 8.

What about the efficiency? Things are quite reasonable. We use only one application of *Sign* and some hashing, regardless of message length. The concern may be that we transmit more data since we have to send both the keys K_1, K_2. But K_1 is much shorter than M so the overhead, and K_2 is much shorter than K_1, so if we are already sending M, the overhead in additional bits is not significant.

Of course one may use more than two iterations. In general, we hash as often as necessary to bring the final key size down to a small enough value that the final key can fit in the scope of the given signature function. Using a scheme like XOR trees, the reduction in key sizes proceeds exponentially, so that only $O(\log^* m)$ iterations are needed to hash a string of length m. In practice, this is bounded by a small constant.

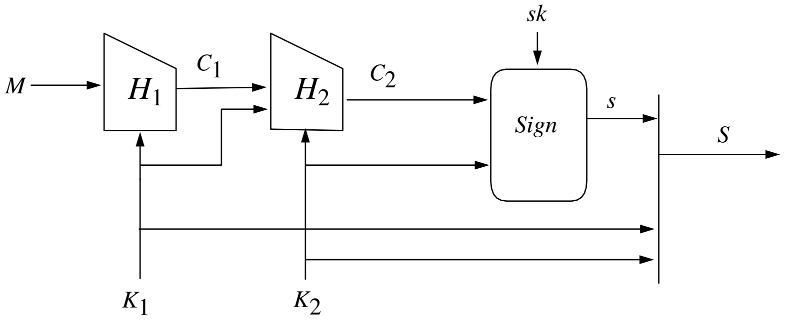

Figure 8: Signing with a TCR hash function using multiple levels of hashing. The technique is useful when key sizes grow with the message lengths, and message may be long.

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. *Advances in Cryptology — Proceedings of CRYPTO '97*, Lecture Notes in Computer Science, Springer-Verlag, 1997. Earlier version of this paper.

[5] M. Bellare and P. Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. *Advances in Cryptology – Eurocrypt* 96 *Proceedings*, Lecture Notes in Computer Science Vol. 1070, U. Maurer ed., Springer-Verlag, 1996.

[6] T. Cormen, C. Leiserson and R. Rivest, Introduction to Algorithms. McGraw-Hill, 1992.

[7] R. Cramer and I. Damgård, New generation of secure and practical RSA based signatures. *Advances in Cryptology – Crypto* 96 *Proceedings*, Lecture Notes in Computer Science Vol. 1109, N. Koblitz ed., Springer-Verlag, 1996.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] H. Dobbertin, Cryptanalysis of MD4. *Fast Software Encryption—Cambridge Workshop*, Lecture Notes in Computer Science, vol. 1039, D. Gollman, ed., Springer-Verlag, 1996.

[13] H. Dobbertin, Cryptanalysis of MD5. Rump Session of Eurocrypt 96, May 1996, `http://d.ngm.9p.pmx6`

[26] R. Rivest, The MD5 message-digest algorithm. IETF RFC 1321 (April 1992).

[27] R. Rivest, A. Shamir and L. Adleman, "A method for obtaining digital signatures and public key cryptosystems," CACM 21 (1978).

[28] J. Rompel, One-way functions are necessary and sufficient for digital signatures. *Proceedings of the* 22nd *Annual Symposium on Theory of Computing*, ACM, 1990.

[29] G. Tsudik, Message authentication with one-way hash functions, *Proceedings of Infocom 92,* IEEE Press, 1992.

[30] 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.

[31] 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.

[32] H. Williams, "A modification of the RSA public key encryption procedure," *IEEE Transactions on Information Theory*, Vol. IT-26, No. 6, November 1980.

A Proofs of the Composition Lemmas

Proof of Lemma 3.1: Let $\mathsf{CF} = (\text{CF-I}, \text{CF-II})$ be a target collision finder for H which runs in time t. Consider the experiment describing CF's attack on H, namely

$$(M, \mathit{State}) \overset{R}{\leftarrow} \text{CF-I}\,;\; K_1 \overset{R}{\leftarrow} \Sigma^{k_1}\,;\; K_2 \overset{R}{\leftarrow} \Sigma^{k_2}\,;\; M' \overset{R}{\leftarrow} \text{CF-II}(K_1K_2, M, \mathit{State})\,. \tag{12}$$

Let $x = H_1(K_1, M)$ and $x' = H_1(K_1, M')$. Now let E_1 be following event: CF is successful and $x = x'$. Let E_2 be the following event: CF is successful, and $x \neq x'$. Let $p_1 = \Pr[E_1]$ and $p_2 = \Pr[E_2]$, the probabilities being under the experiment of Equation (12). Notice that E_1, E_2 are disjoint events with union the event that CF is successful, so we have $\mathit{ProbSuccess}(\mathsf{CF}, H) = p_1 + p_2$. Thus it suffices to upper bound p_1, p_2.

We do this by defining a target collision finder $\mathsf{CF}_1 = (\text{CF-I}_1, \text{CF-II}_1)$ for H_1 and a target collision finder $\mathsf{CF}_2 = (\text{CF-I}_2, \text{CF-II}_2)$ for H_2 so that $\mathit{ProbSuccess}(\mathsf{CF}_1, H_1) = p_1$ and $\mathit{ProbSuccess}(\mathsf{CF}_2, H_2) = p_2$. We make sure that the running time of CF_1 is at most t_1 and that of CF_2 is at most t_2. Out assumptions about the security of H_1, H_2 imply that $p_1 \leq \epsilon_1$ and $p_2 \leq \epsilon_2$, so that $\mathit{ProbSuccess}(\mathsf{CF}, H) \leq \epsilon_1 + \epsilon_2$.

It remains to define the two algorithms. They are:

```
Algorithm $\text{CF-I}_1$
$(M, \mathit{State}) \overset{R}{\leftarrow} \text{CF-I}$
return $(M, \mathit{State})$
```

```
Algorithm $\text{CF-II}_1(K_1, M, \mathit{State})$
$K_2 \overset{R}{\leftarrow} \Sigma^{k_2}$
$M' \overset{R}{\leftarrow} \text{CF-II}(K_1K_2, M, \mathit{State})$
return $M'$
```

```
Algorithm $\text{CF-I}_2$
$(M, \mathit{State}) \overset{R}{\leftarrow} \text{CF-I}$ and $K_1 \overset{R}{\leftarrow} \Sigma^{k_1}$
$x \overset{R}{\leftarrow} H_1(K_1, M)$
return $(x, (M, \mathit{State}, K_1))$
```

```
Algorithm $\text{CF-II}_2(K_2, x, (M, \mathit{State}, K_1))$
$M' \overset{R}{\leftarrow} \text{CF-II}(K_1K_2, M, \mathit{State})$
$x' \overset{R}{\leftarrow} H_1(K_1, M')$
return $x'$
```

The experiment describing CF_1's attack on H_1 is

$$(M, \mathit{State}) \overset{R}{\leftarrow} \text{CF-I}_1 \; ; \; K_1 \overset{R}{\leftarrow} \Sigma^{k_1} \; ; \; M' \overset{R}{\leftarrow} \text{CF-II}_1(K_1, M, \mathit{State}) \; . \tag{13}$$

Let $x = H_1(K_1, M)$ and $x' = H_1(K_1, M')$. By definition CF_1 is successful in breaking H_1 when $M \neq M'$ but $x = x'$. Notice $x = x'$ implies $H_2(K_2, x) = H_2(K_2, x')$ where K_2 is the key chosen by CF-II_1. Furthermore from the definitions of $\text{CF-I}_1, \text{CF-II}_1$ it is easy to see that the experiment of Equation (13) mimics that of Equation (12). This means (M, M') is a collision for $H_1(K_1, \cdot)$ with exactly the probability that event E_1 occurs in the experiment of Equation (12). It follows that $\mathit{ProbSuccess}(\mathsf{CF}_1, H_1) = p_1$.

The experiment describing CF_2's attack on H_2 is

$$(x, (M, \mathit{State}, K_1)) \overset{R}{\leftarrow} \text{CF-I}_2 \; ; \; K_2 \overset{R}{\leftarrow} \Sigma^{k_2} \; ; \; x' \overset{R}{\leftarrow} \text{CF-II}_2(K_2, x, (M, \mathit{State}, K_1)) \; . \tag{14}$$

By definition CF_2 is successful in breaking H_2 when $x \neq x'$ but $H_2(K_2, x) = H_2(K_2, x')$. From the definitions of $\text{CF-I}_2, \text{CF-II}_2$ it is easy to see that the experiment of Equation (14) mimics that of Equation (12). This means (x, x') is a collision for $H_2(K_2, \cdot)$ with exactly the probability that event E_2 occurs in the experiment of Equation (12). It follows that $\mathit{ProbSuccess}(\mathsf{CF}_2, H_2) = p_2$.

It remains to bound the running times and output lengths. The running time of CF_1 is at most $t + O(k_2)$, and this is at most t_1 for the choice of t in the lemma statement. The running time of CF_2 is at most $t + 2T_{H_1}(\mu) + O(k_1)$, which is at most t_2 for the choice of t in the lemma statement. The output length of CF_2 is at most μ_2. The result follows. ▮

Proof of Lemma 3.2: We follow the proof of Lemma 3.1. The constructions are the same. We only need a few additional observations to justify them.

Note that we are guaranteed $|M| = |M'|$ in the collision (M, M') found by CF, because the latter is now by assumption an equal-length collision finder. This means, first, that collisions found by CF_1 are also equal-length ones. It also means that $|x| = |x'|$ (where $x = H_1(K_1, M)$ and $x' = H_1(K_1, M')$) because H_1 is length-consistent. The latter means that the collisions found by CF_2 are also equal-length ones. Put these observations together with the previous proof and we are done. ▮