An abridged version of this paper appears in *Advances in Cryptology – Eurocrypt 97 Proceedings*, Lecture Notes in Computer Science Vol. ??, W. Fumy ed., Springer-Verlag, 1997.

A New Paradigm for Collision-free Hashing: Incrementality at Reduced Cost

MIHIR BELLARE[*]   DANIELE MICCIANCIO[†]

November 1996

Abstract

We present a simple, new paradigm for the design of collision-free hash functions. Any function emanating from this paradigm is *incremental*. (This means that if a message x which I have previously hashed is modified to x' then rather than having to re-compute the hash of x' from scratch, I can quickly "update" the old hash value to the new one, in time proportional to the amount of modification made in x to get x'.) Also any function emanating from this paradigm is parallelizable, useful for hardware implementation.

We derive several specific functions from our paradigm. All use a standard hash function, assumed ideal, and some algebraic operations. The first function, MuHASH, uses one modular multiplication per block of the message, making it reasonably efficient, and significantly faster than previous incremental hash functions. Its security is proven, based on the hardness of the discrete logarithm problem. A second function, AdHASH, is even faster, using additions instead of multiplications, with security proven given either that approximation of the length of shortest lattice vectors is hard or that the weighted subset sum problem is hard. A third function, LtHASH, is a practical variant of recent lattice based functions, with security proven based, again on the hardness of shortest lattice vector approximation.

---

[*] Dept. of Computer Science & Engineering, University of California at San Diego, 9500 Gilman Drive, La Jolla, California 92093, USA. E-Mail: `mihir@watson.ibm.com`. URL: `http://d-n-nut-p--tp--m`

Contents

1 Introduction

A collision-free hash function maps arbitrarily long inputs to outputs of a fixed length, but in such a way that it is computationally infeasible to find a collision, meaning two distinct messages x, y which hash to the same point.[1] These functions were first conceived and designed for the purpose of hashing messages before signing, the point being to apply the (expensive) signature operation only to short data. (Whence the collision-freeness requirement, which is easily seen to be a necessary condition for the security of the signature scheme.) Although this remains the most important usage for these functions, over time many other applications have arisen as well. Collision-free hash functions are now well recognized as one of the important cryptographic primitives, and are in extensive use.

We are interested in finding hash functions that have a particular efficiency feature called "incrementality" which we describe below. Motivated by this we present a new paradigm for the design of collision-free hash functions. We obtain from it some specific incremental hash functions that are significantly faster than previous ones.

It turns out that even putting incrementality aside, functions resulting from our paradigm have attractive features, such as parallelizability.

1.1 Incremental Hashing

THE IDEA. The notion of incrementality was advanced by Bellare, Goldreich and Goldwasser [BGG1]. They point out that when we cryptographically process documents in bulk, these documents may be related to each other, something we could take advantage of to speed up the computation of the cryptographic transformations. Specifically, a message x' which I want to hash may be a simple modification of a message x which I previously hashed. If I have already computed the hash $f(x)$ of x then, rather than re-computing $f(x')$ from scratch, I would like to just quickly "update" the old hash value $f(x)$ to the new value $f(x')$. An incremental hash function is one that permits this.

For example, suppose I want to maintain a hash value of all the files on my hard disk. When one file is modified, I do not want to re-hash the entire disk contents to get the hash value. Instead, I can apply a simple update operation that takes the old hash value and some description of the changes to produce the new hash value, in time proportional to the amount of change.

In summary, what we want is a collision-free hash function f for which the following is true. Let $x = x_1 \ldots x_n$ be some input, viewed as a sequence of blocks, and say block i is modified to x'_i. Let x' be the new message. Then given $f(x), i, x_i, x'_i$ it should be easy to compute $f(x')$.

STANDARD CONSTRUCTIONS FAIL. Incrementality does not seem easy to achieve. Standard methods of hash function construction fail to achieve it because they involve some sort of iteration. This is true for constructions based on block ciphers. (For description of these constructions see for example the survey [PGV].) It is also true for the compression function based constructions that use the Merkle-Damgård meta-method [Me, Da2]. The last includes popular functions like MD5 [Ri], SHA-1 [SHA] and RIPEMD-160 [DBP]. The modular arithmetic based hash functions are in fact also iterative, and so are the bulk of number-theory based ones, eg. [Da1].

A thought that comes to mind is to use a tree structure for hashing, as described in [Me, Da2]. (Adjacent blocks are first hashed together, yielding a text half the length of the original one, and then the process is repeated until a final hash value is obtained.) One is tempted to think this is incremental because if a message block is modified, work proportional only to the tree depth needs to be done to update. The problem is you need to *store the entire tree*, meaning all the intermediate

---

[1] The formal definition in Section 2 speaks of a *family* of functions, but we dispense with the formalities for now.

hash values. What we want is to store only the final hash value and be able to increment given only this.

PAST WORK. To date the only incremental hash function was proposed by [BGG1], based on work of [CHP]. This function is based on discrete exponentiation in a group of prime order. It uses one modular exponentiation per message block to hash the message. This is very expensive, especially compared with standard hash functions. An increment operation takes time independent of the message size, but also involves exponentiation, so again is expensive. We want to do better, on both counts.

1.2 The Randomize-then-combine Paradigm

We introduce a new paradigm for the construction of collision-free hash functions. The high level structure is quite simple. View the message x as a sequence of blocks, $x = x_1 \ldots x_n$, each block being b bits long, where b is some parameter to choose at will. First, each block x_i is processed, via a function h, to yield an outcome y_i. (Specifically, $y_i = h(\langle i \rangle \,.\, x_i)$ where $\langle i \rangle$ is a binary representation of the block index i and "$\,.\,$" denotes concatenation). These outcomes are then "combined" in some way to yield the final hash value $y = y_1 \odot y_2 \odot \ldots \odot y_n$, where $\odot$ denotes the "combining operation."

Here h, the "randomizing" function, is derived in practice from some standard hash function like SHA-1, and treated in the analysis as an "ideal" hash function or random oracle [BR]. (For completeness a brief discussion of this paradigm is provided in Appendix E). The combining operation $\odot$ is typically a group operation, meaning that we interpret $y_1, \ldots, y_n$ as members of some commutative group G whose operation is denoted $\odot$.

We call this the *randomize-then-combine* paradigm. It is described fully in Section 3. The security of this method depends of course on the choice of group, and we will see several choices that work. The key benefit we can observe straight away is that the resulting hash function is incremental. Indeed, if x_i changes to x'_i, one can re-compute the new hash value as $y \odot h(x_i)^{-1} \odot h(x'_i)$ where y is the old hash value and the inverse operation is in the group. Also it is easy to see the computation of the hash function is parallelizable.

By choosing different groups we get various specific, incremental, collision-free hash functions, as we now describe.

Notice that h needs itself to be collision-free, but applies only to fixed length inputs. Thus, it can be viewed as a "compression function." Like [Me, Da2], our paradigm can thus be viewed as constructing variable input length hash functions from compression functions. However, our construction is "parallel" rather than iterative. It is important to note, though, that even though our constructions seem secure when h is a good compression function (meaning one that is not only collision-free but also has some randomness properties) the proofs of security require something much stronger, namely that h is a random oracle.

1.3 MuHASH and its Features

MUHASH. Our first function, called MuHASH for "multiplicative hash," sets the combining operation to multiplication in a group G where the discrete logarithm problem is hard. (For concreteness, think $G = Z_p^*$ for a suitable prime p. In this case, hashing consists of "randomizing" the blocks via h to get elements of Z_p^* and then multiplying all these modulo p).

EFFICIENCY. How fast is MuHASH? The cost is essentially one modular multiplication per b-bit block. Notice that one computation of h per b-bit block is also required. However, the cost of computing h will usually be comparatively small. This is especially true if the block length is chosen appropriately. For example, if h is implemented via SHA, chosing b as a multiple of 512, the expensive padding step in computing SHA can be avoided and the total cost of computing h

for every block is about the same as a single application of SHA on the whole message. The cost of h will be neglected in the rest of the paper.

At first glance the presence of modular operations may make one pessimistic, but there are two things to note. First, it is multiplications, not exponentiations. Second, we can make the block size b large, making the amortized per-bit cost of the multiplications small. Thus, MuHASH is much faster than the previous incremental hash function. In fact it is faster than any number-theory based hash function we know. Note if hardware for modular multiplication is present, not unlikely these days, then MuHASH becomes even more efficient to compute.

The increment operation on a block takes one multiplication and one division, again much better than the previous construction.

SECURITY. We show that as long as the discrete logarithm problem in G is hard and h is ideal, MuHASH is collision-free. (This may seem surprising at first glance since there does not seem to be any relation between discrete logarithms and MuHASH. In the latter we are just multiplying group elements, and no group generator is even present!) That is, we show that if there is *any* attack that finds collisions in MuHASH then there is a way to efficiently compute discrete logarithms in G. The strength of this statement is that it makes no assumptions about the cryptanalytic techniques used by the MuHASH attacker: no matter what these techniques may be, the attacker will fail as long as the discrete logarithm problem in G is hard. This proven security means we are obviated from the need to consider the effects of any specific attacks. That is, it is not necessary to have an exhaustive analysis of a list of possible attacks.

The proven security provides a strong qualitative guarantee of the strength of the hash function. However, we have in addition a strong quantitative guarantee. Namely, we have reductions that are *tight*. To obtain these we have to use the group structure more carefully. We present separate reductions, with slightly different characteristics, for groups of prime order and for the multiplicative group modulo a prime. These are Theorem 4.2 and Theorem 4.3 respectively. In practice this is important because it means we can work with a smaller value of the security parameter making the scheme more efficient.

An interesting feature of MuHASH is that its "strength in practice" may greatly exceed its proven strength. MuHASH is proven secure if the discrete logarithm problem is hard, but it might be secure even if the discrete logarithm problem is easy, because we know of no attack that finds collisions *even if it is easy to compute discrete logarithms.* And in practice, collision-freeness of h seems to suffice.

1.4 AdHASH and its Features

AdHASH (for "additive hash") uses addition modulo a large enough integer M as the combining operation in the randomize-then-combine paradigm. In other words, to hash we first randomize the blocks of the message using h and then add all the results modulo M.

Replacing multiplication by addition results in a significant improvement in efficiency. Hashing now only involves n modular additions, and the increment operation is just two modular additions. In fact AdHASH is competitive with standard hash functions in speed, with the added advantages of incrementality and parallelizability.

AdHASH also has strong security guarantees. We show that it is collision-free as long as the "weighted knapsack problem" (which we define) is hard and h is ideal. But Ajtai [Aj] has given strong evidence that the weighted subset sum problem is hard: he has shown that this is true as long as there is no polynomial time approximation algorithm for the shortest vector problem in a lattice, in the worst case. But even if this approximation turns out to be feasible (which we don't expect) the weighted subset sum problem may still be hard, so that AdHASH may still be secure.

We also prove that AdHASH is a universal one-way hash function in the sense of Naor and Yung [NY], assuming the subset sum function of [IN1, IN2] is one-way and h is ideal. (Thus, under a weaker assumption, we can show that a weaker form but still useful form of collision-freeness holds. We note our reductions here are tight, unlike those of [IN1, IN2]. See Appendix D.)

In summary AdHASH is quite attractive both on the efficiency and on the security fronts.

1.5 Hashing from Lattice Problems

Ajtai introduced a linear function which is provably one-way if the problem of approximating the (Euclidean) length of the shortest vector in a lattice is hard [Aj]. (The function is matrix-vector multiplication, with particular parameters). Goldreich, Goldwasser and Halevi [GGH] observed that Ajtai's main lemma can be applied to show that the function is actually collision-free, not just one-way. We observe that this hash function is incremental. But we also point out some impracticalities.

We then use our randomize-then-combine paradigm to derive a more practical version of this function. (Our function is more efficient and has smaller key size). It is called LtHASH (for "lattice hash"). The group is $G = Z_p^k$ for some integers p, k, meaning we interpret the randomized blocks as k-vectors over Z_p and add them component-wise. Assuming h is ideal the security of this hash function can be directly related to the problem underlying the security of Ajtai's one-way function [Aj, GGH] so that it is collision-free as long as the shortest lattice vector approximation problem is hard.

Note that the same assumption that guarantees the security of LtHASH (namely hardness of approximation of length of the shortest vector in a lattice) also guarantees the security of AdHASH, and the efficiency is essentially the same, so we may just stick with AdHASH. However it is possible that LtHASH has some features of additional interest, and is more directly tied to the lattice hardness results, so it is worth mentioning.

1.6 Attack on XHASH

Ideally, we would like to hash using only "conventional" cryptography (ie. no number theory.) A natural thought is thus to set the combining operation to bitwise XOR. But we show in Appendix A that this choice is insecure. We present an attack on the resulting function XHASH, which uses Gaussian elimination and pairwise independence. It may be useful in other contexts.

We are loth to abandon the paradigm based on this: it is hard to imagine any other paradigm that yields incrementality. But we conclude that it may be hard to get security using only conventional cryptography to implement the combining operation. So we turned to arithmetic operations and found the above.

1.7 The balance problem

We identify a computational problem that can be defined in an arbitrary group. We call it the balance problem. It turns out that consideration of the balance problem unifies and simplifies the treatment of hash functions, not only in this paper but beyond. Problems underlying algebraic or combinatorial collision-free hash functions are often balance problems. We will see how the hardness of the balance problem follows from the hardness of discrete logs; how in additive groups it is just the weighted subset sum problem; and that it captures the matrix kernel problem presented in [Aj] which is the basis of lattice based hash functions [GGH].

The problem is simply that given random group elements $a_1, \ldots, a_n$, find disjoint subsets $I, J \subseteq \{1, \ldots, n\}$, not both empty, such that $\bigodot_{i \in I} a_i = \bigodot_{j \in J} a_j$, where $\odot$ is the group operation. Having reduced the security of our hash function to this problem in Lemma 3.1, our main technical effort will be in relating the balance problem in a group to other problems in the group.

1.8 Related Work

For a comprehensive survey of hashing see [MVV, Chapter 9].

Discrete logarithm or factoring based functions. To the best of our knowledge, all previous discrete logarithm or factoring based hash functions which have a security that can be provably related to that of the underlying number theoretic problem use at least one multiplication per *bit* of the message, and sometimes more. (For example this is true of the functions of [Da1], which are based on claw-free permutations [GMR].) In contrast, MuHASH uses one multiplication per b-bit block and can make b large to mitigate the cost of the multiplication. (But MuHASH uses a random oracle assumption which the previous constructions do not. And of course the previous functions, barring those of [BGG1], are non-incremental.)

Collision-free versus universal one-way. Collision-freeness is a stronger property than the property of universal one-wayness defined by Naor and Yung [NY]. Functions meeting their conditions are not necessarily collision-free. (But they do suffice for many applications.)

Subset-sum based hashing. Impagliazzo and Naor [IN1, IN2] define a hash function and prove that it is a universal one-way function (which is weaker than collision-free) as long as the subset-sum function is one-way. The same function is defined in [Da2, Section 4.3]. There it is conjectured to be collision-free as well, but no proof is provided. These functions have a key length as long as the input to be hashed (very impractical) and use one addition per bit of the message. In contrast, AdHASH has short key length and uses one addition per b-bit block of the message, and b can be made large.

Hashing by multiplying in a group. Independently of our work, Impagliazzo and Naor have also considered hashing by multiplying in a group. These results have been included in [IN2], the recent journal version of their earlier [IN1]. In their setup, a list of random numbers $a_1, \ldots, a_n$ is published, and the hash of message x is $\prod_{i=1}^{n} x_i a_i$ where x_i is the i-th bit of x and the product is taken in the group. Thus there is one group operation per bit of the message, and also the key size is proportional to the input to be hashed. Functions resulting from our paradigm use one group operation per b-bit block, which is faster, and have fixed key size. On the security side, [IN2] show that their hash function is universal one-way as long as any homomorphism with image the given group is one-way. (In particular, if the discrete logarithm problem in the group is hard.) In contrast we show that our functions have the stronger property of being collision-free. But the techniques are related and it is also important to note that we use a random oracle assumption and they do not. On the other hand our reductions are tight and theirs are not.

The general security assumption of [IN2] and their results provide insight into why MuHASH may be secure even if the discrete logarithm problem is easy.

Modular arithmetic hash functions. Several iterative modular arithmetic based hash functions have been proposed in the past. (These do not try to provably relate the ability to find collisions to any underlying hard arithmetic problems.) See Girault [Gi] for a list and some attacks. More recent in this vein are MASH-1 and MASH-2, designed by GMD (Gesellschaft fur Mathematik im Dataverarbeitung) and being proposed as ISO standards. However, attacks have been found by Coppersmith and Preneel [CP].

XOR MACs. Our paradigm for hashing is somewhat inspired by, and related to, the XOR MACs of [BGR]. There, XOR worked as a combining operation. But the goal and assumptions were different. Those schemes were for message authentication, which is a private key based primitive.

In particular, the function playing the role of h was *secret*, computable only by the legitimate parties and not the adversary. (So in particular, the attack of Appendix A does not apply to the schemes of [BGR].) However, hash functions have to have a *public* description, and what we see is that in such a case the security vanishes if the combining operation is XOR.

INCREMENTALITY. Other work on incremental cryptography includes [BGG2, Mi]. The former consider primitives other than hashing, and also more general incremental operations than block replacement, such as block insertion and deletion. (Finding collision-free hash functions supporting these operations is an open problem.) The latter explores issues like privacy in the presence of incremental operations.

2 Definitions

2.1 Collision-free Hash Functions

FAMILIES OF HASH FUNCTIONS. A *family of hash functions* F has a *key space* $\mathit{Keys}(F)$. Each key $K \in \mathit{Keys}(F)$ specifies a particular function mapping $\mathit{Dom}(F)$ to $\mathit{Range}(F)$, where $\mathit{Dom}(F)$ is a domain common to all functions in the family, and $\mathit{Range}(F)$ is a range also common to all functions in the family. Formally, we view the family F as a function F: $\mathit{Keys}(F) \times \mathit{Dom}(F) \to \mathit{Range}(F)$, where the function specified by K is $F(K, \cdot)$.

The key space $\mathit{Keys}(F)$ has an associated probability distribution. When we want to pick a particular hash function from the family F we pick K at random from this distribution, thereby specifying $F(K, \cdot)$. The key K then becomes public, available to all parties including the adversary: these hash functions involve no hidden randomness.

In our constructions an "ideal hash function" h is also present. We follow the paradigm of [BR]: In practice, h is derived from a standard cryptographic hash function like SHA, while formally it is modeled as a "random oracle." The latter means h is initially drawn at random from some family of functions, and then made public. Parties have oracle access to h, meaning they are provided with a box which, being queried with a point x, replies with $h(x)$. This is the only way h can be accessed. We stress the oracle is public: the adversary too can access h.

Formally, h will be viewed as part of the key defining a hash function, and the random choice of a key includes the choice of h. Typically a key will have two parts, one being some short string σ and the other being h, so that formally $K = (\sigma, h)$. (For example, σ may be a prime p, to specify that we are working over Z_p^*). We treat them differently in the notation, writing F_σ^h for the function $F(K, \cdot)$. This is to indicate that although both σ and h are public, they are accessed differently: everyone has the complete string σ, but to h only oracle access is provided. It is to be understood in what follows that the families we discuss might involve a random oracle treated in this way, and when the key is chosen at random the oracle is specified too.

We provide in Appendix E a discussion of what security in this random oracle paradigm can guarantee and not guarantee. For yet more information the reader is referred to [BR].

We want hash functions that compress their data. A typical desired choice is that $\mathit{Dom}(F) = \{0,1\}^*$ and $\mathit{Range}(F)$ is some finite set, for example $\{0,1\}^k$ for some integer k. But other choices are possible too.

COLLISION-RESISTANCE. A *collision* for $F(K, \cdot)$ is a pair of strings $x, y \in \mathit{Dom}(F)$ such that $x \neq y$ but $F(K,x) = F(K,y)$. When $\mathit{Dom}(F)$ is larger than $\mathit{Range}(F)$, each $F(K, \cdot)$ will have many collisions. What we want, however, is that these are difficult to find. To formalize this, say a *collision-finder* is an algorithm C that given a key $K \in \mathit{Keys}(F)$ tries to output a collision for $F(K, \cdot)$. (When K includes a random oracle, this of course means the collision-finder gets oracle access to this same random oracle). We are interested in the probability that it is successful. This probability depends on the time t that is allowed C. (For convenience the "time" is the actual running time, on some fixed RAM model of computation, plus the size of the description of the algorithm C. In general we would also measure the amount of memory used, but for simplicity we only measure time. The model of computation is that used in any standard text on algorithms, for example [CLR], and we analyze the running time of algorithms in the same way as in any algorithms course). If a random oracle h is present, we consider the number of h-computations (formally, the number of oracle queries) as a separate resource of the collision-finder, and denote it by q. In this case we have the following.

Definition 2.1 *We say that collision-finder C (t, q, ϵ)-breaks a hash family F if given a key K it runs in time t, makes at most q oracle queries, and finds a collision in $F(K, \cdot)$ with probability at least ϵ. We say that F is (t, q, ϵ)-collision-free if there is no collision-finder which (t, q, ϵ)-breaks F.*

The probability above is over the choice of the key K from *Keys*(F) (which includes the choice of the random oracle h) and the coins of C. If the random oracle is not present, we simply drop the "q", and have (t, ϵ)-breaking and (t, ϵ)-security.

2.2 Incrementality

We follow [BGG1]. Suppose we have computed the hash value $y = F(K, x)$ of a message $x = x_1 \ldots x_n$. Now x is modified: block i is replaced by a new block x'_i. We want to update y to $y' = F(K, x')$, where x' is the message resulting from replacing block i of x by x'_i. We want to do it in some way faster than re-computing $F(K, x')$ from scratch. The job will be done by an *incremental algorithm*. It takes as input K, x, y, i, x'_i and outputs y'. Ideally it runs in time that is independent of the number of blocks in the messages.

2.3 Classes of groups

We will consider groups in which computational problem (example, computing discrete logarithms or solving weighted knapsacks) is hard. Formally, we must treat families (classes) of groups.

CLASSES OF GROUPS. Formally, a *class of groups* is some finite collection of groups such that given a description $\langle G \rangle$ of a group from the class, one can compute all the group operations. Also, there is some distribution on $\mathcal{G}$ according to which we can draw a (description of a) group. Finally we assume a representation of group elements under which any group element of any group is a L-bit string for some L, meaning $G \subseteq \{0,1\}^L$ for all $G \in \mathcal{G}$. This L is called the output length. For example $\mathcal{G} = \{ Z_p^* : p \text{ is prime and } |p| = k \}$, for some large enough k, is a class of groups. Here $\langle G \rangle = p$ is the prime describing a particular group, and it is drawn at random from all k-bit primes. The output length is $L = k$.

TIMING. In the security analyses we need to estimate running times of the algorithms in the reductions. The timing estimates depend on the groups. Accordingly given a class of groups $\mathcal{G}$ we let $T_{\text{rand}}(\mathcal{G}), T_{\text{mult}}(\mathcal{G}), T_{\exp}(\mathcal{G})$ denote, respectively, the time to pick a random element of G, the time to multiply two elements in G and the time to do an exponentiation in G, for $G \in \mathcal{G}$.

2.4 The balance problem in a group

For the purpose of analyzing the security of our hash functions we introduce a new computational problem, called the balance problem in a group. Lemma 3.1 will relate the security of our hash function to the assumed hardness of this problem. (Our task will then be reduced to finding groups with a hard balance problem. Typically we will do this by further reducing the balance problem to a conventional hard problem like discrete log finding or (weighted) subset sum.) Here we define the balance problem.

Let $\mathcal{G}$ be some family of groups and n an integer. In the $(\mathcal{G}, n)$-balance problem we are given (the description $\langle G \rangle$ of) a group $G \in \mathcal{G}$ and a sequence $a_1, \ldots, a_n$ of elements of G. We must find weights $w_1, \ldots, w_n \in \{-1, 0, +1\}$ not all zero, such that

$$a_1^{w_1} \odot \cdots \odot a_n^{w_n} \;=\; e$$

where $\odot$ is the group operation and e is the identity element in the group.[2] In other words we are asked to find two disjoint subsets $I, J \subseteq \{1, \ldots, n\}$, not both empty, such that $\bigodot_{i \in I} a_i = \bigodot_{j \in J} a_j$. We say that the $(\mathcal{G}, n)$-balance problem is (t, ϵ)-hard if no algorithm, limited to run in time t, can find a solution to an instance $G, a_1, \ldots, a_n$ of the problem with probability more than ϵ, the probability computed over a random choice of G from $\mathcal{G}$, a choice of $a_1, \ldots, a_n$ selected uniformly and independently at random in G, and the coins of the algorithm.

3 The Paradigm

We suggest a new paradigm for the construction of collision-free hash functions.

3.1 The Construction

The construction is depicted in Figure 1. We fix a block size b and let $B = \{0,1\}^b$. Think of the input $x = x_1 \ldots x_n$ as a sequence of blocks, meaning $x_i \in B$ for each $i = 1, \ldots, n$. Let N be larger than the number of blocks in any message we plan to hash, and let $l = \lg(N) + b$. We are given a set G on which some operation, which we call the *combining operation* and denote by $\odot$, has been defined. (The operation is at the very least associative, but, as we will see later, we prefer it be a full-fledged group operation.) We are also given a function h: $\{0,1\}^l \to G$ which we call the *randomizer* or *compression function.* Now what we do is:

1. For each block $i = 1, \ldots, n$, concatenate a $\lg(N)$-bit binary encoding $\langle i \rangle$ of the block index i to the block content x_i to get an *augmented block* $x'_i = \langle i \rangle \,.\, x_i$

2. For each $i = 1, \ldots, n$, apply h to x'_i to get a hash value $y_i = h(x'_i)$

3. Combine $y_1, \ldots, y_n$ via the combining operation to get the final hash value $y = y_1 \odot y_2 \odot \cdots \odot y_n$.

More succinctly we can write the function as

$$\mathrm{HASH}^h_{\langle G \rangle}(x_1 \ldots x_n) \;=\; \bigodot_{i=1}^n h(\langle i \rangle \,.\, x_i)\ , \tag{1}$$

where $\langle G \rangle$ denotes some indication of the group G which enables computation of the group operation. (For example if $G = Z_p^*$ then $\langle G \rangle = p$). We call this the *randomize then combine* construction.

If the output of our hash function (which is an element of G) is too long then optionally we can hash it to a shorter length by applying a standard collision-free hash function such as SHA-1.

Notice that padding the blocks with (a representation of) their indexes before applying h is important for security. Without this, re-ordering of the blocks in a message would leave the hash value unchanged, leading to collisions.

THE HASH FAMILY. Equation (1) specifies an individual function, depending on the group G. Formally, we actually have a family of hash functions, because we will want to draw G from some class of groups for which some computational problem (example, computing discrete logarithms or solving weighted knapsacks) is hard.

Let $\mathcal{G}$ be a class of groups, as defined in Section 2.3. The associated family of hash functions is denoted $\mathrm{HASH}(\mathcal{G}, b)$. An individual function $\mathrm{HASH}^h_{\langle G \rangle}$ of this family, as defined in Equation (1),

---

[2] For a multiplicative group, this means $\prod_{i=1}^n a_i^{w_i} \;=\; 1$. For an additive group it would mean $\sum_{i=1}^n w_i a_i \;=\; 0$.

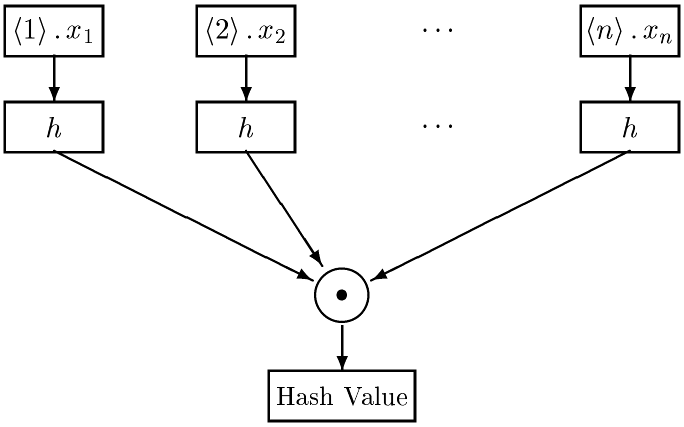

Figure 1: Our paradigm for hashing message $x = x_1 \ldots x_n$: *Process individual blocks via a function h and then combine the results via some operation* $\odot$.

is specified by a random oracle h: $\{0,1\}^l \to G$ and a description $\langle G\rangle$ of a group $G \in \mathcal{G}$. Here $l = b + \lg(N)$ as above. We can set N to a constant like 2^{80}. (We will never need to hash a message with more than 2^{80} blocks!). Thus $l = b + O(1)$. So think of l as $O(b)$. This is assumed in estimates. The key defining $\text{HASH}^h_{\langle G\rangle}$ consists, formally, of $\langle G\rangle$ and h. (See Section 2.1). The domain of this hash family is $B^{\leq N} = B \cup B^2 \cup \ldots \cup B^N$ where $B = \{0,1\}^b$, namely all strings over B of length at most N. The range of this family is $\{0,1\}^L$ where L is the output length of $\mathcal{G}$.

3.2 Incrementality and parallelizability

Since the combining operation is associative, the computation is parallelizable. In order to get an incremental hash function we will work in a commutative group, so that $\odot$ is also commutative and invertible. In such a case, increments are done as follows. If block x_i changes to x'_i then the new hash is $y \odot h(\langle i\rangle . x_i)^{-1} \odot h(\langle i\rangle . x'_i)$ where $(\cdot)^{-1}$ denotes the inverse operation in the group and y is the old hash, namely the hash of x.

3.3 Choosing the randomizer

For security the randomizer h must definitely be collision-free: it is easy to see that the entire construction fails to be collision-free otherwise. In practice h is derived from a standard hash function. (We suggest that the derivation be keyed. For example, $h(x') = H(\kappa . x' . \kappa)$ where κ is a random string viewed as part of the key specifying the hash function and $H(y)$ is an apprporiate length prefix of SHA-1$(0 . y)$. SHA-1$(1 . y) \ldots$.) In the analyses, we in fact assume much more, namely that it is an "ideal" hash function or random oracle [BR].) Its computation is assumed fast.

3.4 Choosing the Combining Operation

Making the right choice of combining operation is crucial for security and efficiency.

COMBINING BY XORING DOESN'T WORK. Ideally, we would like to hash using only "conventional" cryptography. (Ie. no number theory.) A natural thought towards this end is to set the combining operation to bitwise XOR. But this choice is insecure. Let us look at this a bit more closely.

Let $G = \{0,1\}^k$ for some fixed length k, like $k = 128$. If we set the combining operation to bitwise XOR, denoted $\oplus$, the resulting function is

$$\mathrm{XHASH}^h(x_1 \ldots x_n) \;=\; \bigoplus_{i=1}^{n} h(\langle i \rangle \,.\, x_i)$$

The incrementality is particularly efficient in this case since it takes just a couple of XORs. The question is whether XHASH^h is collision-free. At first glance, it may seem so. However XHASH is in fact not collision-free. Indeed, it is not even one-way. (One-wayness is necessary, but not sufficient, for collision-resistance). The attack is interesting, and may be useful in other contexts, so we present it in Appendix A. Given a string $z \in \{0,1\}^k$ we show there how to find a message $x = x_1 \ldots x_n$ such that $\mathrm{XHASH}^h(x) = z$. (The attack succeeds with probability at least 1/2, the probability being over the choice of h, and works for $n \geq k+1$.) The attack makes $2n$ h-computations, sets up a certain linear system, and then uses Gaussian elimination to solve it. The proof that it works exploits pairwise independence arguments.

OTHER COMBINING OPERATIONS. Thus we see that the choice of combining operation is important, and the most tempting choice, XOR, doesn't work. We are loth to abandon the paradigm based on this: it is hard to imagine any other paradigm that yields incrementality. But we conclude that it may be hard to get security using only conventional cryptography to implement the combining operation. So we turn to arithmetic operations.

We consider two: multiplication in a group where the discrete logarithm problem is hard, and addition modulo an integer of appropriate size. It turns out they work. But we need to be careful about security given the experience with XOR.

To this end, we begin below by relating the security of the hash function to the balance problem in the underlying group. A reader interested more in the constructions should skip to Section 4.

3.5 The balance lemma

The security of the hash functions obtained from our paradigm can be related to the balance problem in the underlying class of groups, as defined in Section 2.4. Specifically, in order to prove the security of a particular hash function family $\mathrm{HASH}(\mathcal{G}, b)$, it will be sufficient to show that the balance problem associated with the corresponding group family is hard. To understand the theorem below, it may be helpful to refer to the definitions in Section 2. Recall that q refers to the number of computations of h and the theorem assumes h is ideal, ie. a random function of $\{0,1\}^l$ to G. The theorem says that if the balance problem is hard over $\mathcal{G}$ then the corresponding family of hash functions is collision-free. Moreover it tells us precisely how the parameters describing the security in the two cases relate to each other. Below $c > 1$ is a small constant, depending on the model of computation, which can be derived from the proof.

Lemma 3.1 *Let $\mathcal{G}$ and q be such that the $(\mathcal{G}, q)$-balance problem is (t', ϵ')-hard. Then* $\mathrm{HASH}(\mathcal{G}, b)$ *is a (t, q, ϵ)-collision-free family of hash functions where $\epsilon = \epsilon'$ and $t = t'/c - q \cdot b$.*

Proof: We are given a collision-finder C, which takes $\langle G \rangle$ and an oracle for h, and eventually outputs a pair of distinct strings, $x = x_1 \ldots x_n$ and $y = y_1 \ldots y_m$, such that $\mathrm{HASH}^h_{\langle G \rangle}(x) = \mathrm{HASH}^h_{\langle G \rangle}(y)$. We want to construct an algorithm K that solves the $(\mathcal{G}, q)$-balance problem. It takes as input $\langle G \rangle$ and a list of values $a_1, \ldots, a_q$ selected uniformly at random in G. K runs C on input $\langle G \rangle$, answering its oracle queries with the values $a_1, a_2, \ldots, a_q$ in order. (We assume oracle queries are not repeated.) Notice the answers to oracle queries are uniformly and independently distributed over G, as they would be if $h\colon \{0,1\}^l \to G$ were a random function. We will let Q_i denote the i-th oracle query of C, namely the one answered by a_i, so that $h(Q_i) = a_i$, and we let $Q = \{Q_1, \ldots, Q_q\}$.

Finally, C outputs two strings $x = x_1 \dots x_n$ and $y = y_1 \dots y_m$, such that $x \neq y$ but $\mathrm{HASH}^h_{\langle G\rangle}(x) = \mathrm{HASH}^h_{\langle G\rangle}(y)$. We know this means

$$h(\langle 1\rangle . x_1) \odot \dots \odot h(\langle n\rangle . x_n) \;=\; h(\langle 1\rangle . y_1) \odot \dots \odot h(\langle m\rangle . y_m)\ , \tag{2}$$

the operations being in G. (Note that the strings x and y are not necessarily of the same size; that is, m may not be equal to n.) We will construct a solution to the balance problem from x and y. Let $x'_i = \langle i\rangle . x_i$ for $i = 1, \dots, n$ and $y'_i = \langle i\rangle . y_i$ for $i = 1, \dots, m$. We can assume wlog that $x'_1, \dots, x'_n, y'_1, \dots, y'_m \in Q$. We let $f_x(i)$ be the (unique) value $j \in [q]$ such that $x'_i = q_j$ and we let $f_y(i)$ be the (unique) $j \in [q]$ such that $y'_i = q_j$. We then let $I = \{\, f_x(i) \,:\, i = 1, \dots, n \,\}$ and $J = \{\, f_y(i) \,:\, i = 1, \dots, m \,\}$ be, respectively, the indices of queries corresponding to x and y. Equation (2) can be rewritten as

$$\textstyle\bigodot_{i \in I} a_i \;=\; \bigodot_{j \in J} a_j\ . \tag{3}$$

We know that $x \neq y$, and so $I \neq J$. Now for $i = 1, \dots, q$ let us define

$$w_i \;=\; \begin{cases} -1 & \text{if } i \in J - I \\ 0 & \text{if } i \in I \cap J \\ +1 & \text{if } i \in I - J. \end{cases}$$

Then the fact that $I \neq J$ means that not all $w_1, \dots, w_q$ are 0, and Equation (3) implies $a_1^{w_1} \odot \cdots \odot a_q^{w_q} \;=\; e$. The probability that we find a solution to the balance problem is exactly that with which C outputs a collision, and the time estimates can be checked. ▌

4 MuHASH: The Multiplicative Hash

Here we present our first concrete construction, the multiplicative hash function (MuHASH), and analyze its efficiency and security.

4.1 Construction and efficiency

We set the combining operation in our paradigm to multiplication in a group where the discrete logarithm problem is hard. (For example $G = Z_p^*$ for an appropriate prime p, or some subgroup thereof.) To emphasize multiplication, we call the function MuHASH rather than the general HASH of Section 3. So the function is

$$\mathrm{MuHASH}^h_{\langle G\rangle}(x_1 \dots x_n) \;=\; \textstyle\prod_{i=1}^n h(\langle i\rangle . x_i)\ . \tag{4}$$

The product is taken in the group G over which we are working. (Thus if we are working in Z_p^*, it is just multiplication modulo p. In this case $\langle G\rangle = p$ describes G.) Here all the notation and conventions are as in Section 3.1. A class of groups $\mathcal{G}$ gives rise to a family $\mathrm{MuHASH}(\mathcal{G}, b)$ of hash functions as described in Section 2.3.

If $G = Z_p^*$ then for security $k = |p|$ should be at least 512 or even 1024, making the final hash value of the same length. A hash of this size may be directly useful, for example for signatures, where the message is hashed before signing. (For RSA we want a string in Z_N^* where N is the modulus, and this may be 1024 bits.) In other cases, we may want a smaller hash value, say 160 bits. In such cases, we allow a final application of a standard collision-free hash function to the above output. For example, apply SHA-1 to $\mathrm{MuHASH}^h_{\langle G\rangle}(x)$ and get a 160 bit string.

Computing our hash function takes one multiplication per block, ie. one multiplication per b bits of input. (This is in contrast to previous methods which required one multiplication per bit.) To minimize the cost, one can increase the block size. The increment operation is performed as per Section 3.2, and takes one inverse and two multiplication operations in the group, plus two applications of h. Thus it is cheap compared to re-computing the hash function.

Note that the computation of $\text{MuHASH}^h_{\langle G \rangle}$ is entirely parallelizable. The applications of h on the augmented blocks can be done in parallel, and the multiplications can also be done in parallel, for example via a tree. This is useful when we have hardware for the group operation, as well might be the case.

4.2 The discrete logarithm problem

The security of MuHASH depends on the discrete logarithm problem in the underlying group. Let us begin by defining it.

Let $\mathcal{G}$ be a class of groups, for example $\mathcal{G} = \{ Z_p^* : p \text{ is a prime with } |p| = k \}$. Let $G \in \mathcal{G}$, g a generator of G, and $y \in G$. A *discrete log finder* is an algorithm I that takes $g, y, \langle G \rangle$ and tries to output $\log_g(y)$. Its success probability is taken over a random choice of G from $\mathcal{G}$ (for the example $\mathcal{G}$ above, this means we choose a random k-bit prime p) and a random choice of $y \in G$. We say that the discrete logarithm problem in $\mathcal{G}$ is (t', ϵ')-hard if any discrete logarithm finder that runs in time t' has success probability at most ϵ'.

4.3 Security of MuHASH

The attack on XHASH we saw above indicates that we should be careful about security. Moving from XOR to multiplication as the "combining" operation kills that attack in the case of MuHASH. Are there other attacks?

We indicate there are not in a very strong way. We show that as long as the discrete logarithm problem in G is hard and h is ideal, $\text{MuHASH}^h_{\langle G \rangle}$ is collision-free. That is, we show that if there is *any* attack that finds collisions in $\text{MuHASH}^h_{\langle G \rangle}$ then there is a way to efficiently compute discrete logarithms in G. This proven security obviates us from the need to consider the effects of any specific attacks.

At first glance this relation of the security of MuHASH to the discrete logarithm problem in G may seem surprising. Indeed, the description of $\text{MuHASH}^h_{\langle G \rangle}$ makes no mention of a generator g, nor is there even any exponentiation: we are just multiplying group elements. Our proofs illustrate how the relationship is made.

We look first at general groups, then, to get better quantitative results (ie. better reductions) we look at special classes of groups.

APPROACH. All our proofs have the same structure. First it is shown that if the discrete log problem is hard in $\mathcal{G}$ then also the balance problem is hard in $\mathcal{G}$. The security of the hash function is then derived from Lemma 3.1. The main technical question is thus relating the balance and discrete logarithm problems in groups.

Notice this is a question just about computational problems in groups: it has nothing to do with our hash functions. Accordingly, we have separated the materiel on this subject, putting it in Appendix B. There we prove a sequence of lemmas, showing how the quality of the reduction changes with the group. These lemmas could be of independent interest. We now proceed to apply these lemmas to derive the security of MuHASH for various groups.

SECURITY IN GENERAL GROUPS. The following theorem says that the only way to find collisions in MuHASH (assuming h is ideal) is to solve the discrete logarithm problem in the underlying group. The result holds for any class of groups with hard discrete logarithm problem. Refer to Sections 4.1, 4.2 and 2.3 for notation. Below $c > 1$ is a small constant, depending on the model of computation, which can be derived from the proof.

Theorem 4.1 *Let $\mathcal{G}$ be a class of groups with output length L. Assume the discrete logarithm problem in $\mathcal{G}$ is (t', ϵ')-hard. Then for any q, $\mathrm{MuHASH}(\mathcal{G}, b)$ is a (t, q, ϵ)-collision-free family of hash functions, where $\epsilon = q\epsilon'$ and $t = t'/c - q \cdot [T_{\mathrm{rand}}(\mathcal{G}) + T_{\exp}(\mathcal{G}) + L + b]$.*

Proof: Follows from Lemma 3.1 and Lemma B.1. ▌

In the above reduction, if the probability one can compute discrete logarithms is ϵ' then the probability of breaking the hash function may be as high as $\epsilon = q\epsilon'$. A typical choice of q is about 2^{50}. This means the discrete logarithm problem in G must be very hard in order to make finding collisions in the hash function quite hard. To make ϵ appreciably small, we must make ϵ' very small, meaning we must use a larger value of the security parameter, meaning it takes longer to do multiplications and the hash function is less efficient. It is preferable to have a stronger reduction in which ϵ is closer to ϵ'. (And we want to do this while maintaining the running time t' of the discrete logarithm finder to be within an additive amount of the running time t of the collision-finder, as it is above. Reducing the error by repetition does not solve our problem.)

We now present better reductions. They exploit the group structure to some extent. We look first at groups of prime order (where we have an essentially optimal reduction), then at multiplicative groups modulo a prime (where we do a little worse, but still very well, and much better than the naive reduction above).

SECURITY IN GROUPS OF PRIME ORDER. The recommended group G in which to implement $\mathrm{MuHASH}^h_{\langle G \rangle}$ is a group of prime order. (For example, pick a large prime p of the form $p = 2p' + 1$ where p' is also prime, and let G be a subgroup of order p' in Z_p^*. The order of Z_p^* is $p - 1$ which is not prime, but the order of G is p' which is prime.) The reason is that the reduction is tight here. As usual $c > 1$ is a small constant, depending on the model of computation, which can be derived from the proof.

Theorem 4.2 *Let $\mathcal{G}$ be a class of groups of prime order with output length L. Assume the discrete logarithm problem in $\mathcal{G}$ is (t', ϵ')-hard. Then for any q, $\mathrm{MuHASH}(\mathcal{G}, b)$ is a (t, q, ϵ)-collision-free family of hash functions, where $\epsilon = 2\epsilon'$ and $t = t'/c - q \cdot [T_{\mathrm{rand}}(\mathcal{G}) + T_{\mathrm{mult}}(\mathcal{G}) + T_{\exp}(\mathcal{G}) + L + b] - L^2$.*

Proof: Follows from Lemma 3.1 and Lemma B.2. ▌

The form of the theorem statement here is the same as in Theorem 4.1, but this time the probability ϵ of breaking the hash function is no more than twice the probability ϵ' of computing discrete logarithms, for an attacker who runs in time which is comparable in the two cases.

SECURITY IN Z_p^*. The most popular group in which to work is probably Z_p^* for a prime p. Since its order is $p - 1$ which is not prime, the above theorem does not apply. What we can show is that an analogous statement holds. The probability ϵ of breaking the hash function may now be a little more than the probability ϵ' of computing discrete logarithms, but only by a small factor which is logarithmic in the size k of the prime p. As usual $c > 1$ is a small constant, depending on the model of computation, which can be derived from the proof.

Theorem 4.3 *Let $k \geq 6$ and let $\mathcal{G} = \{ Z_p^* : p \text{ is a prime with } |p| = k \}$. Suppose the discrete logarithm problem in $\mathcal{G}$ is (t', ϵ')-hard. Then for any q, $\mathrm{MuHASH}(\mathcal{G}, b)$ is a (t, q, ϵ)-collision-free family of hash functions, where $\epsilon = 4\ln(0.694k) \cdot \epsilon'$ and $t = t'/c - qk^3 - qb$.*

Proof: Follows from Lemma 3.1 and Lemma B.3. ▌

The factor multiplying ϵ' will not be too large: for example if $k = 512$ it is about 24.

SECURITY IN PRACTICE. We have shown that computation of discrete logarithms is necessary to break MuHASH as long as h is ideal. Yet, it could be that MuHASH is even stronger. The reason is that even computation of discrete logarithms does not seem *sufficient* to find collisions in MuHASH. That is, we suspect that finding collisions in $\text{MuHASH}^h_{\langle G\rangle}$ remains hard even if we can compute discrete logarithms. In particular, we know of no attacks that find collisions in MuHASH even if discrete logarithm computation is easy. In this light it may be worth noting that the natural attempt at a discrete logarithm computation based attack is to try to "reduce" the problem to finding additive collisions in the exponents and then apply the techniques of Section A. But this does not work. (See Appendix C for more details.) The underlying problem is a kind of knapsack problem which is probably hard. In fact this suggests that the hash function obtained by setting the combining operation in our paradigm to addition might be already collision-free. This function and its security are discussed in Section 5.

Some evidence that breaking MuHASH is harder than computing discrete logarithms comes from the results of [IN2] who indicate that multiplication in G is a one-way hash as long as *any* homomorphism with image G is hard. We can extend their proofs, with added conditions, to our setting. This indicates that unless all such homomorphisms are invertible via discrete logarithm computation, MuHASH will be collision-free.

Also, although the proofs make very strong assumptions about the function h, it would appear that in practice, the main thing is that h is collision-free. In particular if h is set to SHA-1 then given the modular arithmetic being done on top of the h applications, it is hard to see how to attack the function.

5 AdHASH: Hashing by Adding

AdHASH is the function obtained by setting the combining operation in our paradigm to addition modulo a sufficiently large integer. Let us give the definition more precisely and then go on to look at security.

5.1 Construction and Efficiency

We let M be a k-bit integer. As usual let $x = x_1 \ldots x_n$ be the data to be hashed, let b denote the block size, let N be such that all messages we will hash have length at most N and let $l = b + \lg(N)$. We let h: $\{0,1\}^l \to Z_M$ be a hash function, assumed ideal. The function is–

$$\text{AdHASH}^h_M(x_1 \ldots x_n) \;=\; \textstyle\sum_{i=1}^n h(\langle i\rangle \,.\, x_i) \mod M \;.$$

Thus, the "key" of the function is the integer M. We let $\text{AdHASH}(k, b)$ denote the corresponding family, consisting of the functions AdHASH^h_M as M ranges over all k-bit integers and h ranges over all functions of $\{0,1\}^l$ to Z_M. The distribution on the key space is uniform, meaning we draw M at random from all k-bit integers and h at random from all functions of $\{0,1\}^l$ to Z_M, in order to define a particular hash function from the family.

AdHASH is much faster than MuHASH since we are only adding, not multiplying. Furthermore, it would seem k can be quite small, like a few hundred, as compared to the sizes we need for MuHASH to make sure the discrete logarithm problem is hard, making the gain in efficiency even greater. In fact the speed of AdHASH starts approaching that of standard hash functions. And of course it is incremental, with the cost of incrementality also now reduced to just adding and subtracting. Thus it is a very tempting function to use. Next we look at security.

5.2 Standard and weighted subset sum problems

The security of AdHASH can be related to the difficulty of certain modular subset-sum or knapsack type problems which we now define.

Standard modular knapsack. In the (k, q)-knapsack problem we are given a k-bit integer M, and q numbers $a_1, \ldots, a_q \in Z_M$. We must find weights $w_1, \ldots, w_q \in \{0, 1\}$, not all zero, such that

$$\textstyle\sum_{i=1}^{q} w_i a_i \equiv 0 \pmod{M} .$$

In other words, we are being asked to find a non-empty subset $I \subseteq \{1, \ldots, q\}$ such that $\sum_{i \in I} a_i$ is 0 modulo M. We want to assume this task is hard. The assumption is that this is true when M and $a_1, \ldots, a_q$ are random. Formally, we say that the (k, q)-knapsack problem is (t', ϵ')-hard if no algorithm, limited to run in time t', can find a solution to an instance $M, a_1, \ldots, a_q$ of the (k, q)-knapsack problem with probability more than ϵ', the probability computed over a random choice of k-bit integer M, a choice of $a_1, \ldots, a_q$ selected uniformly and independently at random in Z_M, and the coins of the algorithm.

This hardness assumption is essentially the same as Impagliazzo and Naor's assumption that the subset sum function is one-way [IN1, IN2].

We must be careful how we choose the parameters: it is well known that for certain values of k and q, the problem is not hard. We must avoid these values. Specifically, we will make sure that $\Omega(\log q) < k < q$. It turns out this choice will not be a restriction for us anyway. Nice discussions of what is known are available in [Od] and [IN2, Section 1.2], and the reader is referred there for more information.

This assumption will suffice to show that AdHASH is a universal one-way hash function (cf. Appendix D). In order to show collision-freeness we make a stronger assumption.

Weighted knapsack problem. In the (k, q)-weighted-knapsack problem we are given a k-bit integer M, and q numbers $a_1, \ldots, a_q \in Z_M$. We must find weights $w_1, \ldots, w_q \in \{-1, 0, +1\}$, not all zero, such that

$$\textstyle\sum_{i=1}^{q} w_i a_i \equiv 0 \pmod{M}$$

Thus the difference is that now a weight is allowed to take on the value -1. We say that the (k, q)-weighted-knapsack problem is (t', ϵ')-hard if no algorithm, limited to run in time t', can find a solution to an instance $M, a_1, \ldots, a_q$ of the (k, q)-weighted-knapsack problem with probability more than ϵ', the probability computed over a random choice of k-bit integer M, a choice of $a_1, \ldots, a_q$ selected uniformly and independently at random in Z_M, and the coins of the algorithm.

Notice this is just the $(\mathcal{G}, q)$-balance problem for the class of groups $\mathcal{G} = \{ Z_M : |M| = k \}$. But it is worth re-stating it for this case.

The hardness of the weighted problem is a stronger assumption than the hardness of the standard problem, but beyond that the relation between the problems is not known. However, there is important evidence about the hardness of the weighted knapsack problems that we discuss next.

Relation to lattice problems. A well-known hard problem is to approximate the length of the shortest vector in a lattice. The best known polynomial time algorithms [LLL, SH] achieve only an exponential approximation factor. It has been suggested that there is no polynomial time algorithm which achieves a polynomial approximation factor. Under this assumption, Ajtai showed that both the standard and the weighted subset-sum problems are hard [Aj]. (Actually he allows any small integer weights, not just $-1, 0, +1$ like we do). That is, there is no polynomial time algorithm to solve these problems.

This is important evidence in favor of both the knapsack assumptions discussed above. As long as approximating the length of a shortest lattice vector is hard, even in the worst case, the knapsack problems are hard. This increases the confidence we can have in cryptosystems based on these knapsack assumptions.

Values of t', ϵ' for which the standard and weighted knapsack problems are (t', ϵ')-hard can be derived from Ajtai's proof, as a function of the concrete parameters for which one assumes shortest vector length approximation is hard. Since Ajtai's proof is quite complex we do not know exactly what the relation is.

We note however that even more is true. Even if the assumption about lattices fails (meaning an efficient approximation algorithm for the shortest lattice vector problem emerges), the knapsack problems may still be hard. Thus, we present all our results in terms of the knapsack assumptions.

5.3 Security of AdHASH

We relate the collision-freeness of AdHASH to the weighted knapsack problem. Below $c > 1$ is a small constant, depending on the model of computation, which can be derived from the proof.

Theorem 5.1 *Let k and q be integers such that the (k, q)-weighted-knapsack problem is (t', ϵ')-hard. Then* AdHASH(k, b) *is a (t, q, ϵ)-collision-free family of hash functions where $\epsilon = \epsilon'$ and $t = t'/c - qM$.*

Proof: Follows from Lemma 3.1 and the observation that weighted knapsack is a particular case of the balance problem, as mentioned in Section 5.2. ▌

6 Incremental Hashing via Lattice Problems

Ajtai introduced a function which he showed was one-way if the problem of approximating the shortest vector in a lattice to polynomial factors is hard [Aj]. Goldreich, Goldwasser and Halevi observed that Ajtai's main lemma could be applied to show that the same function is in fact collision-free [GGH]. Here we observe this hash function is incremental, and consider its practicality. We then use our paradigm to derive a more practical version of this function whose security is based on the same assumption as in [Aj, GGH] plus the assumption that our h is ideal. Let us begin by recalling the problem shown hard by Ajtai's main lemma.

6.1 The Matrix Kernel Problem

In the (k, n, s)-matrix-kernel problem we are given p, M where p is an s-bit integer and M is a k by n matrix with entries in Z_p. We must find a non-zero n-vector w with entries in $\{-1, 0, +1\}$ such that $Mw = 0 \bmod p$. (The operation here is matrix-vector multiplication, with the operations done modulo p). We say this problem is (t', ϵ')-hard if no algorithm, limited to run in time t', can find a solution to an instance p, M of the (k, n, s)-matrix-kernel problem with probability more than ϵ', the probability computed over a random choice of p, a random choice of matrix M, and the coins of the algorithm.

Suppose $ks < n < 2^s/(2k^4)$. Ajtai showed that with these parameters the matrix-kernel problem is hard under the assumption that there is no polynomial time algorithm to approximate the length of a shortest vector in a lattice within a polynomial factor. (Ajtai's result was actually stronger, since he allowed entries in w to be any integers of "small" absolute value. However [GGH] observed that weights of $-1, 0, +1$ are what is important in the context of hashing and we restrict our attention to these).

A close examination of Ajtai's proof will reveal specific values of t', ϵ' for which we can assume the matrix kernel problem is (t', ϵ')-hard, as a function of the assumed hardness of the shortest vector approximation problem. Since the proof is quite complex we don't know what exactly these values are.

Notice that the matrix kernel problem is just an instance of our general balance problem: it is the $(\mathcal{G}, n)$-balance problem for $\mathcal{G} = \{\ Z_p^k\ :\ |p| = s\ \}$. This shows how the balance problem unifies so many hash functions.

6.2 The Ajtai-GGH Function

The function. Let M be a random k by n matrix with entries in Z_p and let x be an n vector with entries in $\{0,1\}$. The function of [Aj, GGH] is–

$$H_{M,p}(x) \ = \ Mx \bmod p\ .$$

Note $Mx \bmod p$ is a k-vector over Z_p, meaning it is $k\lg(p)$ bits long. Since the parameters must obey the restriction $k\lg(p) < n < p/(2k^4)$, the function is compressing: the length n of the input x is more than the length $k\log(p)$ of the output $Mx \bmod p$. Thus it is a hash function. Now, if the matrix kernel problem is hard this function is one-way [Aj]. Moreover, under the same assumption it is collision-free [GGH]. It follows from [Aj] that the function is collision-free as long as shortest vector approximation is hard.

Incrementality. We observe the above function is incremental. Let M_i denote the i-th column of M, for $i = 1, \ldots, n$. This is a k-vector over Z_p. Let $x = x_1 \ldots x_n$ with $x_i \in \{0,1\}$ for $i = 1, \ldots, n$. Now we can write the function as–

$$H_{M,p}(x) \ = \ \textstyle\sum_{i=1}^{n} x_i M_i \ \bmod p\ .$$

In other words, we are summing a subset of the columns, namely those corresponding to bits of x that are 1. Now suppose bit x_i changes to x_i'. If y (a k-vector over Z_p) is the old hash value then the new hash value is $y + (x_i' - x_i)M_i \bmod p$. Computing this takes k additions modulo p, or $O(k\log(p))$ time, a time which does not depend on the length n of x.

Drawbacks of this function. A serious drawback of H is that the description of the function is very large: $(nk+1)\lg(p)$ bits. In particular, the description size of the function grows with the number of bits to be hashed. This means we must set an a priori limit on the number of bits to be hashed and use a function of size proportional to this. This is not feasible in practice.

One way to partially overcome this problem is to specify the matrix entries via an ideal hash function. For example if h: $[k] \times [n] \to Z_p$ is such a function, set $M[i,j] = h(i,j)$. But we can do better. The function we describe next not only has small key size and no limit on input length, but is also more efficient.[3]

6.3 LtHASH

Our function is called LtHASH for "lattice based hash."

The construction. We apply the randomize-then-combine paradigm with the group G set to Z_p^k. That is, as usual let $x = x_1 \ldots x_n$ be the data to be hashed, let b denote the block size, let N be such that all messages we will hash have length at most N and let $l = b + \lg(N)$. We let h: $\{0,1\}^l \to Z_p^k$ be a hash function, assumed ideal. Think of its output as a k-entry column vector over Z_p. Our hash function is–

$$\mathrm{LtHASH}_p^h(x_1 \ldots x_n) \ = \ \textstyle\sum_{i=1}^{n} h(\langle i \rangle \,.\, x_i) \ \bmod p\ .$$

---

[3] Another way to reduce the key size is define $H_{M,P}$ only on relatively short data, and then, viewing it as a compression function, apply Damgård's iteration method [Da2]. But then incrementality is lost. Also, the key sizes, although no longer proportional to the data length, are still larger than for the construction we will describe.

Namely, each application of h yields a column vector, and these are added, componentwise modulo p, to get a final column vector which is the hash value.

Notice that there is no longer any matrix M in the function description. This is why the key size is small: the key is just the s-bit integer p. Also LtHASH_p^h is more efficient than the function described above because it does one vector addition per b-bit input block rather than per input bit, and b can be made large.

We let $\mathrm{LtHASH}(k, s, b)$ denote the corresponding family, consisting of the functions LtHASH_p^h as p ranges over s-bit integers and h ranges over all functions of $\{0,1\}^l$ to Z_p^k. The key defining any particular function is the integer p, and the distribution on the key space is uniform, meaning we draw p at random from all s-bit integers in order to define a particular hash function from the family.

Notice that AdHASH is the special case of LtHASH in which $k = 1$ and $p = M$.

SECURITY. We relate the collision-freeness of LtHASH to the hardness of the matrix-kernel problem. The relation may not be evident a priori because LtHASH does not explicitly involve any matrix. But, intuitively, there is an "implicit" k by q matrix M being defined, where q is the number of oracle queries allowed to the collision-finder. This matrix is not "fixed:" it depends on the input. But finding collisions in LtHASH_p^h relates to solving the matrix kernel problem for this matrix. Below $c > 1$ is a small constant, depending on the model of computation, which can be derived from the proof.

Theorem 6.1 *Let k, q, s be integers such that the (k, q, s)-matrix-kernel problem is (t', ϵ')-hard. Then $\mathrm{LtHASH}(k, s, b)$ is a (t, q, ϵ)-collision-free family of hash functions where $\epsilon = \epsilon'$ and $t = t'/c - qks$.*

Proof: Follows from Lemma 3.1 and the observation, made in Section 6.1, that the matrix kernel problem is a particular case of the balance problem when the group is Z_p^k. ▮

We will choose the parameters so that $ks < q < 2^s/(2k^4)$. (Recall $s = |p|$). In this case, we know that the required matrix kernel problem is hard as long as shortest lattice vector approximation is hard.

To actually implement the function we must have some idea of what values to assign to the various security parameters. Opinions as to the concrete complexity of the shortest lattice vector approximation problem vary across the community: it is not clear how high must be the dimension of the lattice to get a specific desired security level. (Although the best known algorithm for shortest vector approximation is only proven to achieve an exponential factor [LLL], its in practice performance is often much better. And Schnorr and Hörner [SH] have found heuristics that do better still). In particular, it does not seem clear how big we need take k (which corresponds to the dimension of the lattice) before we can be sure of security. One must also take into account the exact security of the reductions, which are far from tight. (Some discussion is in [GGH, Section 3]).

Keeping all this in mind let us look at our case. It seems safe to set $k = 500$. (Less will probably suffice). We want to allow q, the number of oracle queries, to be quite large, say $q = 2^{70}$. To ensure $q < 2^s/(2k^4)$ we must take s about 110. Namely p is 110 bits long. This is longer than what the function of [Aj, GGH] needs, making operations modulo p slower for LtHASH, but this is compensated for by having much fewer such operations to do, since we can make the block size b large.

Of course LtHASH is still incremental. Incrementing takes one addition and one subtraction over Z_p^k.

COMPARISON WITH OUR OTHER PROPOSALS. LtHASH is very similar to AdHASH. In fact it is just AdHASH implemented over a different domain, and the security can be proven based on the same underlying problem of hardness of shortest lattice vector approximation. Notice also that AdHASH can be considered a special case of LtHASH, namely, the case $k = 1$. However the proof of security of LtHASH does not immediately carry over to AdHASH because the shortest lattice vector problem in dimension $k = 1$ is easily solved by the Euclidean algorithm. So, the concrete security of LtHASH might be better because the relation to shortest lattice vector approximation is more direct.

Comparison with MuHASH is difficult, depending much on how parameters are set in both functions, but AdHASH and LtHASH are likely to be more efficient, especially because we can make the block size b large.

Acknowledgments

We thank Russell Impagliazzo for telling us about the relations between subset-sum and lattices, and for bringing [IN2] to our attention. We thank the (anonymous) referees of Eurocrypt 97 for comments which improved the presentation of this paper.

References

[Aj] M. Ajtai, "Generating hard instances of lattice problems," *Proceedings of the* 28th *Annual Symposium on Theory of Computing*, ACM, 1996.

[BGG1] M. Bellare, O. Goldreich and S. Goldwasser, "Incremental cryptography: The case of hashing and signing," *Advances in Cryptology – Crypto* 94 *Proceedings*, Lecture Notes in Computer Science Vol. 839, Y. Desmedt ed., Springer-Verlag, 1994.

[BGG2] M. Bellare, O. Goldreich and S. Goldwasser, "Incremental cryptography with application to virus protection," *Proceedings of the* 27th *Annual Symposium on Theory of Computing*, ACM, 1995.

[BGR] M. Bellare, R. Guérin and P. Rogaway, "XOR MAcs: New methods for message authentication using finite pseudorandom functions," *Advances in Cryptology – Crypto* 95 *Proceedings*, Lecture Notes in Computer Science Vol. 963, D. Coppersmith ed., Springer-Verlag, 1995.

[BR] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," *Proceedings of the First Annual Conference on Computer and Communications Security*, ACM, 1993.

[Co] D. Coppersmith, "Two Broken Hash Functions," IBM Research Report RC-18397, IBM Research Center, Yorktown Heights, NY, October 1992.

[CP] D. Coppersmith and B. Preneel, "Comments on MASH-1 and MASH-1," Manuscript, February 1995.

[CHP] D. Chaum, E. Heijst and B. Pfitzmann, "Cryptographically strong undeniable signatures, unconditionally secure for the signer," *Advances in Cryptology – Crypto* 91 *Proceedings*, Lecture Notes in Computer Science Vol. 576, J. Feigenbaum ed., Springer-Verlag, 1991.

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

[Da1] I. Damgard "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.

[Da2] I. Damgard "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.

[DBP] H. Dobbertin, A. Bosselaers and B. Preneel, “RIPEMD-160: A strengthened version of RIPEMD,” *Fast Software Encryption*, Lecture Notes in Computer Science 1039, D. Gollmann, ed., Springer-Verlag, 1996.

[GJ] M. Garey and D. Johnson, “Computers and intractability: A guide to the theory of NP-completeness,” W. H. Freedman and Company, San Francisco, 1979.

[Gi] M. Girault, “Hash functions using modulo-N operations,” *Advances in Cryptology – Eurocrypt* 87 *Proceedings*, Lecture Notes in Computer Science Vol. 304, D. Chaum ed., Springer-Verlag, 1987.

[GGH] O. Goldreich, S. Goldwasser and S. Halevi, “Collision-Free Hashing from Lattice Problems,” *Theory of Cryptography Library* (`http://........yr.s.l..tore.rests.mit.edu/~tcryptol/`) 96-09, July 1996

[GMR] S. Goldwasser, S. Micali and R. Rivest, “A digital signature scheme secure against adaptive chosen-message attacks,” *SIAM Journal of Computing,* Vol. 17, No. 2, pp. 281–308, April 1988.

[IN1] R. Impagliazzo and M. Naor, “Efficient cryptographic schemes provably as secure as subset sum,” *Proceedings of the* 30th *Symposium on Foundations of Computer Science*, IEEE, 1989.

[IN2] R. Impagliazzo and M. Naor, “Efficient cryptographic schemes provably as secure as subset sum,” *Journal of Cryptology,* Vol. 9, No. 4, Autumn 1996.

[LLL] A. Lenstra, H. Lenstra and L. Lovász, “Factoring polynomials with rational coefficients,” *Mathematische Annalen* Vol. 261, pp. 515–534, 1982.

[MVV] A. Menezes, P. Van Oorschot and S. Vanstone, “Handbook of Applied Cryptography,” CRC Press, 1996.

[Me] R. Merkle “One Way Hash Functions and DES,” *Advances in Cryptology – Crypto* 89 *Proceedings*, Lecture Notes in Computer Science Vol. 435, G. Brassard ed., Springer-Verlag, 1989.

[Mi] D. Micciancio, “Oblivious data structures: applications to cryptography,” *Proceedings of the* 29th *Annual Symposium on Theory of Computing*, ACM, 1997.

[NY] M. Naor and M. Yung, “Universal one-way hash functions and their cryptographic applications,” *Proceedings of the* 21st *Annual Symposium on Theory of Computing*, ACM, 1989.

[Od] A. Odlyzko, “The rise and fall of knapsack cryptosystems,” Advances in computational number theory, C. Pomerance ed., *Proc. Symp. Applied Math* No. 42, pp. 75–88, AMS, 1990.

[PGV] B. Preneel, R. Govaerts and J. Vandewalle, “Hash functions based on block ciphers: a synthetic approach,” *Advances in Cryptology – Crypto* 93 *Proceedings*, Lecture Notes in Computer Science Vol. 773, D. Stinson ed., Springer-Verlag, 1993.

[Ri] R. Rivest, “The MD5 Message-Digest Algorithm,” IETF RFC 1321, April 1992.

[RS] J. Rosser and L. Schoenfeld, “Approximate formulas for some functions of prime numbers,” *Illinois Journal of Math* Vol. 6, 1962.

[SH] C. Schnorr and H. Hörner, “Attacking the Chor-Rivest cryptosystem with improved lattice reduction,” *Advances in Cryptology – Eurocrypt* 95 *Proceedings*, Lecture Notes in Computer Science Vol. 921, L. Guillou and J. Quisquater ed., Springer-Verlag, 1995.

[SHA] FIPS 180-1. “Secure Hash Standard,” Federal Information Processing Standard (FIPS), Publication 180-1, National Institute of Standards and Technology, US Department of Commerce, Washington D.C., April 1995.

A Attack on XHASH

In Section 3 we presented XHASH as a plausible candidate for a incremental collision-free hash function but indicated that it was in fact insecure. Here we present the attack showing this. Recall that the function is $\mathrm{XHASH}^h(x_1 \ldots x_n) = h(\langle 1 \rangle . x_1) \oplus \cdots \oplus h(\langle n \rangle . x_n)$. Here each x_i is a b-bit block, and $l = b + \lg(N)$ is large enough to accommodate the block plus an encoding of its index, by dint of making N larger than the number of blocks in any message to be hashed. Our assumption is that h: $\{0,1\}^l \to \{0,1\}^k$ is ideal, ie. a random function of $\{0,1\}^l$ to $\{0,1\}^k$.

Our claim is that there is an attack that easily finds collisions in XHASH^h. We will in fact show something stronger, namely that XHASH^h is not even a one-way function. Given any k bit string z, we can efficiently compute a string x such that $\mathrm{XHASH}^h(x) = z$. (To see that this means XHASH^h is not collision-free, let $z = \mathrm{XHASH}^h(y)$ for some random y and then apply the algorithm to produce x. With high probability $x \neq y$ so we have a collision).

We reduce the problem to solving linear equations. See [Co] for other attacks that exploit linear equations.

The attack. Given $z \in \{0,1\}^k$ we now show how to find x so that $\mathrm{XHASH}^h(x) = z$. Fix two messages $x^0 = x_1^0 \ldots x_n^0$ and $x^1 = x_1^1 \ldots x_n^1$ with the property that $x_i^0 \neq x_i^1$ for all $i = 1, \ldots, n$. (We will see later how to set n. In fact $n = k+1$ will suffice.) For any n-bit string $y = y[1] \ldots y[n]$ we let $x^y = x_1^{y[1]} \ldots x_n^{y[n]}$. We claim that we can find a y such that $\mathrm{XHASH}^h(x^y) = z$. Let us first say how to find such a y, then see why the method works.

We compute the $2n$ values $\alpha_i^j = h(\langle i \rangle . x_i^j)$ for $j = 0, 1$ and $i = 1, \ldots, n$. We want to find $y[1], \ldots, y[n] \in \mathrm{GF}(2)$ such that

$$\alpha_1^{y[1]} \oplus \alpha_2^{y[2]} \ldots \oplus \ldots \alpha_n^{y[n]} \; = \; z \; .$$

Let us now regard $y[1], \ldots, y[n]$ as variables. We want to solve the equation

$$\bigoplus_{i=1}^{n} \alpha_i^0 y[i] \oplus \alpha_i^1 (1 - y[i]) \; = \; z \; .$$

To solve this, we turn it into a system of equations over GF(2). We first introduce new variables $\bar{y}[1], \ldots, \bar{y}[n]$. We will force $\bar{y}[i] = 1 - y[i]$. Then we turn the above into k equations, one for each bit. The resulting system is:

$$\begin{array}{lll} y[i] \oplus \bar{y}[i] \; = \; 1 & & (i = 1, \ldots, n) \\ \bigoplus_{i=1}^{n} \alpha_i^0[j] y[i] \oplus \alpha_i^1[j] \bar{y}[i] \; = \; z[j] & & (j = 1, \ldots, k) \end{array}$$

Here we have $n + k$ equations in $2n$ unknowns, over the field GF(2). Below we show that if $n = k+1$ then there exists a solution with probability 1/2. We now set $n = k+1$ and solve the set of equations, for example via Gaussian elimination, to get values for $y[1], \ldots, y[n] \in \mathrm{GF}(2)$. (The system is slightly under-determined in that there are $n + k = 2k + 1$ equations in $2n = 2k + 2$ unknowns. It can be solved by setting one unknown arbitrarily.) This completes the description of the attack. Now we have to see why it works.

Analysis. There are two main claims. The first is that a solution y to the above does exist (with reasonable probability as long as n is sufficiently large). The second is that given that some y exists, the algorithm finds such a y. The latter is clear from the procedure, so we concentrate on the first. The following lemma implies that with $n = k+1$ a solution exists with probability at least one-half.

Lemma A.1 *Fix $z \in \{0,1\}^k$. Fix two messages $x^0 = x_1^0 \ldots x_n^0$ and $x^1 = x_1^1 \ldots x_n^1$ with the property that $x_i^0 \neq x_i^1$ for all $i = 1, \ldots, n$. For any n-bit string $y = y[1] \ldots y[n]$ let $x^y = x_1^{y[1]} \ldots x_n^{y[n]}$. Then*

$$\Pr[\, \exists y \in \{0,1\}^n \; : \; \mathrm{XHASH}^h(x^y) = z \,] \; \geq \; 1 - \frac{2^k}{2^n} \; .$$

The probability here is over a random choice of h from the set of all functions mapping $\{0,1\}^l \to \{0,1\}^k$.

Proof: For $y \in \{0,1\}^n$ let X_y be the random variable which is 1 if $\mathrm{XHASH}^h(x^y) = z$ and 0 otherwise. Let $X = \sum_{y\in\{0,1\}^n} X_y$. Now $\Pr[\forall y \in \{0,1\}^n : \mathrm{XHASH}^h(x^y) \neq z] = \Pr[X = 0]$. So we want to show that $\Pr[X = 0] \leq 2^k/2^n$. The first step is to observe the following.

Claim. The collection of random variables $\{X_y\}_{y\in\{0,1\}^n}$ is pairwise independent.

Proof. This follows from the definition of XHASH^h. Suppose $y \neq y'$. Then there is a point in the set $\{\, \langle i\rangle . x_i^{y[i]} : 1 \leq i \leq n \,\}$ which is not in the set $\{\, \langle i\rangle . x_i^{y'[i]} : 1 \leq i \leq n \,\}$. Now assume we know the value of

$$\mathrm{XHASH}^h(x^{y'}) \;=\; h(\langle 1\rangle . x_1^{y'[1]}) \oplus \cdots \oplus h(\langle n\rangle . x_n^{y'[n]}) \; . \tag{5}$$

Even then, the value of $\mathrm{XHASH}^h(x^y) = h(\langle 1\rangle . x_1^{y[1]}) \oplus \cdots \oplus h(\langle n\rangle . x_n^{y[n]})$ is still undetermined (ie. uniformly distributed) because this sum includes the value of h on a point at which h was not evaluated in the computation of Equation (5). In other words, for any $\alpha, \alpha' \in \{0,1\}^k$,

$$\Pr\Big[\, \mathrm{XHASH}^h(x^y) = \alpha \mid \mathrm{XHASH}^h(x^{y'}) = \alpha' \,\Big] \;=\; 2^{-k} \; .$$

From this the claim follows. □

We also note that $\mathrm{XHASH}^h(x^y)$ is uniformly distributed over $\{0,1\}^k$ for each fixed y. Thus $\mathbf{E}[X_y] = 2^{-k}$ and $\mathbf{E}[X] = 2^n \cdot 2^{-k}$. Since the random variables are pairwise independent, we can apply Chebyshev's inequality to see that

$$\Pr[\, X = 0\,] \;\leq\; \Pr[\, |X - \mathbf{E}[X]| \geq 2^{n-k}\,] \;\leq\; \frac{\mathbf{Var}[X]}{2^{2(n-k)}} \;=\; 2^{2(k-n)} \cdot \textstyle\sum_{y\in\{0,1\}^n} \mathbf{Var}[X_y] \; .$$

But $\mathbf{Var}[X_y] = \mathbf{E}[X_y] \cdot (1 - \mathbf{E}[X_y]) \leq \mathbf{E}[X_y] = 2^{-k}$ so the above is $2^{2(k-n)} \cdot 2^n \cdot 2^{-k} = 2^k/2^n$. ▌

B The balance problem and discrete logs

In this section we show how the intractability of the discrete logarithm in a group implies the intractability of the balance problem in the same group. These are the technical lemmas underlying the theorems on the security of MuHASH presented in Section 4.3.

We stress that the question here is purely about computational problems in groups, having nothing to do with our hash functions. We first prove a very general, but quantitatively weak result for arbitrary groups. Then we prove strong results for groups of prime order and the group of integers modulo a prime. Refer to Section 2.4 for a definition of the balance problem and Section 4.2 for a definition of the discrete logarithm problem.

General groups. The following says that if computing discrete logs in some class of groups is hard, then so is the balance problem. As usual $c > 1$ is a small constant, depending on the model of computation, which can be derived from the proof.

Lemma B.1 *Let $\mathcal{G}$ be a class of groups with output length L. Assume the discrete logarithm problem in $\mathcal{G}$ is (t', ϵ')-hard. Then for any q, the $(\mathcal{G}, q)$-balance problem is (t, ϵ)-hard, where $\epsilon = q\epsilon'$ and $t = t'/c - q \cdot [T_{\mathrm{rand}}(\mathcal{G}) + T_{\mathrm{exp}}(\mathcal{G}) + L]$.*

Proof: We are given an algorithm A, which takes $\langle G\rangle$ and a sequence of elements $a_1, \ldots, a_q$ in G and outputs weights $w_1, \ldots, w_q \in \{-1, 0, +1\}$, not all zero, such that $\Pi_{i=1}^q a_i^{w_i} = 1$. Let g be a generator of the group G. We want to construct a discrete logarithm finding algorithm I. It takes as input $\langle G\rangle$, g, and $y \in G$, the last randomly chosen, and returns $\log_g(y)$.

We let $\rho = |G|$ be the order of G. We will use A to build I. I first picks at random an integer q^* in the range $1, \ldots, q$. I then computes elements a_i ($i = 1, \ldots, q$) as follows. If $i = q^*$ then $a_i = y$. Otherwise it chooses at random $r_i \in Z_\rho$ and sets $a_i = g^{r_i}$. (Notice that since y is random and g is a generator, all a_i are uniformly distributed over G.) Finally, I runs A on input $\langle G \rangle, a_1, \ldots, a_q$ and gets a sequence of weights $w_1, \ldots, w_q$, not all zero, such that $a_1^{w_1} \cdot \cdots \cdot a_q^{w_q} = 1$. Let i^* be such that $w_{i^*} \neq 0$. Since the choice of q^* was random and unknown to A, with probability at least $1/q$ it will be the case that the $q^* = i^*$. For notational convenience, assume $q^* = i^* = 1$. Now, substituting, we have

$$y^{w_1} \cdot g^{w_2 r_2} \cdots g^{w_q r_q} \; = \; 1 \; .$$

Re-arranging the temrs and noticing that $w_1^{-1} = w_1$ (in Z_ρ) gives us

$$y \; = \; g^{-w_1(w_2 r_2 + \cdots + w_q r_q) \bmod \rho}$$

Thus, $r = -w_1(w_2 r_2 + \cdots + w_q r_q) \bmod \rho$ is the discrete logarithm of y and I can output it and halt. The probability that I is successful is ϵ times the probability that $w_{q^*} \neq 0$, and we saw the latter was at least $1/q$. That is, $\epsilon' = \epsilon / q$.

Since I runs A it incurs time t. Computing each a_i takes one random choice and one exponentiation (except for a_{q^*} which only needs to be copied), meaning $T_{\mathrm{rand}}(\mathcal{G}) + T_{\exp}(\mathcal{G})$ steps per element. The output of C may be up to t bits long so reading it is another investment of time upto t. The final modular additions take $O(qL)$ time. The total time for the algorithm is thus $t' = t + q \cdot [T_{\mathrm{rand}}(\mathcal{G}) + T_{\exp}(\mathcal{G}) + L]$. ∎

This is a very general result, but quantitatively not the best. We now tighten the relationship between the parameters for special classes of groups.

GROUPS OF PRIME ORDER. Let $\mathcal{G}$ be some class of groups of prime order for which the discrete logarithm problem is hard, as discussed in Section 4.3. Below we see that $\epsilon = 2\epsilon'$ rather than $\epsilon = q\epsilon'$ as before, which is quite an improvement. As usual $c > 1$ is a small constant, depending on the model of computation, which can be derived from the proof.

Lemma B.2 *Let $\mathcal{G}$ be a class of groups of prime order with output length L. Assume the discrete logarithm problem in $\mathcal{G}$ is (t', ϵ')-hard. Then for any q, the $(\mathcal{G}, q)$-balance problem is (t, ϵ)-hard, where $\epsilon = 2\epsilon'$ and $t = t'/c - q \cdot [T_{\mathrm{rand}}(\mathcal{G}) + T_{\mathrm{mult}}(\mathcal{G}) + T_{\exp}(\mathcal{G}) + L] - L^2$.*

Proof: We follow and modify the proof of Lemma B.1. By assumption G has prime order. We let $\rho = |G|$ be this order. So $G = \{ \, g^i \, : \, i \in Z_\rho \, \}$. Note that computation in the exponents is modulo ρ and takes place in a field, namely Z_ρ. We will make use of this.

Given A we are constructing I. I takes as input $\langle G \rangle$, g, and $y \in G$, the last randomly chosen. If $y = 1$ (the "1" here standing for the identity element of G), then I can immediately answer $\log_g(y) = 0$. So, we can assume that $y \neq 1$. The key point where we differ from the previous proof is in how the input to A is computed. For each $i = 1, \ldots, q$, algorithm I chooses at random $r_i \in Z_\rho$ and also chooses at random $d_i \in \{0, 1\}$ and sets $a_i = g^{d_i} y^{r_i}$. (Notice that g^{d_i} is either 1 or g and we don't need to perform a modular exponentiation to compute it. Notice also that since G has prime order every element of G except 1 is a generator. In particular y is a generator and hence a_i is uniformly distributed over G.) Now we continue to follow the proof of Theorem 4.1. We run A on input $\langle G \rangle, a_1, \ldots, a_q$ and get weights $w_1, \ldots, w_q$, not all zero, such that $a_1^{w_1} \cdot \cdots \cdot a_q^{w_q} = 1$. Substituting the values for a_i we have

$$y^{w_1 r_1} g^{w_1 d_1} \cdots y^{w_q r_q} g^{w_q d_q} \; = \; 1 \; .$$

Re-arranging terms gives us

$$y^{w_1 r_1 + \cdots + w_q r_q \bmod \rho} = g^{-w_1 d_1 - \cdots - w_q d_q \bmod \rho} ,$$

Now let

$$\begin{aligned} r &= w_1 r_1 + \cdots + w_q r_q \mod \rho \\ d &= -w_1 d_1 - \cdots - w_q d_q \mod \rho , \end{aligned}$$

so that our equation is $y^r = g^d$. Now, observe that $r \neq 0$ with probability at least 1/2. (This is because the value of d_1 remains equi-probably 0 or 1 from the point of view of A, and is independent of other d_i values. At most one of the two possible values of d_1 can make $d = 0$ and hence $r = 0$.) If it is the case that $r \neq 0$ then, since ρ is a prime, r has an inverse modulo ρ. I computes the inverse of r modulo ρ and denotes it by r^{-1}. I outputs $r^{-1} d \bmod \rho$. We have $g^{dr^{-1}} = y^{rr^{-1}} = y$ so the output is indeed $\log_g(y)$.

To show the algorithm outputs $\log_g(y)$ with the claimed probability ϵ', we just need to observe that the input distribution to A is that required by the balance problem. A solves this problem with probability ϵ and we get $\log_g(y)$ with probability at least one half of that. ▌

THE GROUP Z_p^*. Finally we look at the group Z_p^* where p is prime. This group has order $p - 1$, which is not prime, so Lemma B.2 does not apply, but we can still do much better than Lemma B.1. As usual $c > 1$ is a small constant, depending on the model of computation, which can be derived from the proof.

Lemma B.3 *Let $k \geq 6$ and let $\mathcal{G} = \{ Z_p^* \ : \ p$ is a prime with $|p| = k \}$. Suppose the discrete logarithm problem in $\mathcal{G}$ is (t', ϵ')-hard. Then for any q, the $(\mathcal{G}, q)$-balance problem is (t, ϵ)-hard, where $\epsilon = 4\ln(0.694k) \cdot \epsilon'$ and $t = t'/c - qk^3 - b$.*

The following, which we will use in the proof, can be derived from inequalities in Rosser and Schoenfeld [RS].

Lemma B.4 *For any integer $N \geq 23$ it is the case that*

$$\frac{\varphi(N)}{N} \geq \frac{1}{4 \cdot \ln \ln N} .$$

We will have $N = p - 1$, and it is to guarantee $N \geq 23$ that we let the length k of p be at least 6 in the theorem.

Proof of Lemma B.3: We let $G = Z_p^*$ and let $\rho = |G| = p - 1$. Thus $\langle G \rangle = p$. We now follow and modify the proofs of Lemma B.1 and Lemma B.2. Given A we are constructing I.

The key point where we differ from the previous proof is in how the input to A is computed. For each $i = 1, \ldots, q$, algorithm I chooses at random $r_i \in Z_\rho$ and also chooses at random $d_i \in Z_\rho$. It sets $a_i = g^{d_i} y^{r_i}$. (Notice that a_i is uniformly distributed in G because d_i is random and g is a generator.)

Finally, we run A on input $\langle G \rangle, a_1, \ldots, a_q$. We define r and d as in the previous proof and get to the equation $y^r = g^d$. We would like to compute $r^{-1} \bmod \rho$. The problem is that since ρ is no longer prime, this inverse may not exist. However, we claim (to be justified later) that r is uniformly distributed in Z_ρ. This means that $\gcd(r, \rho) = 1$ with probability

$$\frac{\varphi(\rho)}{\rho} \geq \frac{1}{4 \ln \ln(\rho)} \geq \frac{1}{4 \ln \ln(2^k)} \geq \frac{1}{4 \ln(k \ln(2))} \geq \frac{1}{4 \ln(0.694k)} ,$$

having used Lemma B.4 and the fact that $\rho = p - 1 \leq 2^k$. We can compute $\gcd(r, \rho)$, and, if it is one, compute $r^{-1} \bmod \rho$, in which case we can output $\log_g(y)$ as before, and the probability we succeed is the above.

Now we must justify the claim that r is uniformly distributed in Z_{p-1}. Note A has no information on the r_i values, since the a_i values are uniformly and independently distributed of the r_i values, thanks to the d_i values. So we are adding a non-zero number of uniformly distributed values. So the result is uniformly distributed. ▌

C MuHASH: An Attack that Fails

Here we ask the following question. Suppose we could compute discrete logarithms in G. How would we, even then, find collisions in MuHASH? We see that some lines of attack fail, re-enforcing our feeling that MuHASH remains hard even if discrete logarithm computation is easy.

Recall that the function is $\text{MuHASH}^h_{\langle G \rangle}(x_1 \ldots x_n) = \prod_{i=1}^n h(\langle i \rangle . x_i)$. We will try to set up an attack similar to that on XHASH and use the ability to compute discrete logarithms in G to turn the multiplicative equation associated to $\text{MuHASH}^h_{\langle G \rangle}(x) = z$ into a linear equation.

Fix two messages $x^0 = x_1^0 \ldots x_n^0$ and $x^1 = x_1^1 \ldots x_n^1$ with the property that $x_i^0 \neq x_i^1$ for all $i = 1, \ldots, n$. For any n-bit string $y = y[1] \ldots y[n]$ we let $x^y = x_1^{y[1]} \ldots x_n^{y[n]}$. We would like to find a y such that $\text{MuHASH}^h_{\langle G \rangle}(x^y) = z$. i.e.

$$\prod_{i=1}^n h(\langle i \rangle . x_i^{y[i]}) = z. \tag{6}$$

Taking the discrete logarithm of both sides of 6 and rearranging the terms we get the equivalent modular linear equation

$$\sum_{i=1}^n \alpha_i y[i] = \zeta \pmod{\rho}. \tag{7}$$

where $\alpha_i = \log_g(h(\langle i \rangle . x_i^0)) - \log_g(h(\langle i \rangle . x_i^1))$, $\zeta = \log_g(z) - \sum_{i=1}^n \alpha_i^1$ and $\rho = |G|$ is the order of the group.

Even assuming that we can compute discrete logarithms and therefore compute the values α_i's and z, how can we solve equation 7?

This is not just a modular linear equation, because the variables $y[i]$, $i = 1, \ldots, n$, may take only the values 0 and 1. This is a modular version of the subset sum problem (aka knapsack), which is known to be NP-complete [GJ]. In [IN1, IN2], Impagliazzo and Naor conjecture that equation 7 is hard to solve when $|G| = 2^k$ and $n \approx k = \lg(|G|)$. Notice that in our case n cannot be much smaller than than $\log(|G|)$ if we want equation 7 to have a solution and cannot be much bigger than $\log(|G|)$ for efficiency reasons.

Therefore, even assuming that we can compute discrete logarithms in G, it is not clear how collisions in $\text{MuHASH}^h_{\langle G \rangle}$ can be found.

This does not *prove* that finding collision in $\text{MuHASH}^h_{\langle G \rangle}$ is harder than computing discrete logarithms, as there could be better ways to use discrete logarithms to set up an attack on $\text{MuHASH}^h_{\langle G \rangle}$. However, it suggests that $\text{MuHASH}^h_{\langle G \rangle}$ might be even more secure than what we proved.

D AdHASH: Universal One-wayness

UNIVERSAL ONE-WAYNESS. In this notion of Naor and Yung [NY] the adversary must first choose x. Then K, defining $F(K, \cdot)$, is chosen, and the adversary must find y so that x, y is a collision. Formally, a *weak collision-finder* is a pair of algorithms (C_1, C_2). The first outputs a pair (x, σ)

where $x \in \mathit{Dom}(F)$ and σ is some state information. The second takes K, x, σ and outputs y. The weak collision-finder is successful if (x, y) is a collision.

Definition D.1 *We say that weak collision-finder (C_1, C_2) (t, q, ϵ)-breaks a hash family H if given oracle h, it runs in time t, makes at most q oracle queries, and finds a collision in H_K^h with probability at least ϵ. We say that H is (t, q, ϵ)-universal-one-way if there is no weak collision-finder which (t, q, ϵ)-breaks H.*

Again the probability above is over the coins of (C_1, C_2), and the (later) choice of the random oracle h and the key K from $\mathit{Keys}(H)$. If the random oracle is not present, we simply drop the "q", and have (t, ϵ)-breaking and (t, ϵ)-security. Note since the adversary is not allowed to choose x as a function of K (and h), it is a weaker notion of security: if H is collision-free then it is universal one-way, but not necessarily vice-versa.

Notice that, since the random oracle h is part of the definition of the hash function and the weak collision-finder (C_1, C_2) must output x before knowing H_K^k, only C_2 is given access to the random oracle and C_1 cannot make queries to h. (To make this meaningful when the "random oracle" is instantiated, it is important to not just set h to some hash function. Instead, we set it to a keyed hash function, as discussed in Section 3.3. The key is part of the description of the function.)

We relate the universal one-wayness of AdHASH to the standard modular knapsack problem. The constant c in the following depends only on the model of computation.

Theorem D.2 *Let k and q be integers such that the (k, q)-knapsack problem is (t', ϵ')-hard. Then* AdHASH(k, b) *is a (t, q, ϵ)-universal-one-way family of hash functions where $t = t' - cq \cdot [l + k]$ and $\epsilon = 2\epsilon'$.*

Proof: We are given a weak collision-finder (C_1, C_2) which (t, q, ϵ)-breaks AdHASH(k, b). Algorithm C_1 outputs a pair (x, σ) where $x = x_1 \dots x_n$ is a string to be hashed and σ is some state information. Algorithm C_2 takes σ, K and an oracle for h, and eventually outputs a string $y = y_1 \dots y_m$, different from x, such that $\mathrm{AdHASH}_M^h(x) = \mathrm{AdHASH}_M^h(y)$. We want to construct an algorithm K that solves the (k, q)-knapsack problem. It takes as input M and a list of values $a_1, \dots, a_q$ selected uniformly at random in Z_M. K first runs C_1 and obtains a pair (x, σ). It then runs C_2 on σ, M, answering its oracle queries according to one of the two following strategy, chosen at random.

- Strategy one: for $i = 1, \dots, n$ answer $h(\langle i \rangle . x_i)$ with a number b_i chosen uniformly and independently at random in Z_M, and answer any other query $h(\langle i_j \rangle . z_j)$ $(j = 1, 2, \dots, q - n)$ with $b_{i_j} + a_j$, where $b_{i_j} = 0$ for $i_j > n$.
- Strategy two: for $i = 1, \dots, n - 1$ answer $h(\langle i \rangle . x_i)$ with a_i, then answer $h(\langle n \rangle . x_n)$ with $\sum_{i=n}^{q} a_i \bmod M$, and finally answer any other query $h(\langle i_j \rangle . z_j)$ $(j = 1, 2, \dots, q - n)$ with a_{n+j}.

(We assume oracle queries are not repeated.) Notice that in both strategies, the answers to the queries are uniformly and independently distributed over Z_M, as it would be if h: $\{0,1\}^l \to Z_M$ were a random function. Finally, C_2 outputs another string $y = y_1 \dots y_m$, such that $x \neq y$ but $\mathrm{AdHASH}_M^h(x) = \mathrm{AdHASH}_M^h(y)$. Notice that for any $i = 1, \dots, m$, it must be either $y_i = x_i$ or $(\langle i \rangle . y_i) = (\langle i_j \rangle . z_j)$ for some j. Let J be the set of indices j such that $(\langle i \rangle . y_i) = (\langle i_j \rangle . z_j)$ for some i. Now we distinguish two cases:

- if $(n \leq m)$, assume that the first strategy was chosen. We have $\mathrm{AdHASH}_M^h(y) = \sum_{i=1}^{n} b_i + \sum_{i \in J} a_i = \mathrm{AdHASH}_M^h(x) + \sum_{i \in J} a_i$. From $\mathrm{AdHASH}_M^h(x) = \mathrm{AdHASH}_M^h(y)$ and $x \neq y$, it follows that $\sum_{i \in J} a_i = 0$ and $J \neq \emptyset$, i.e., J is a solution to the given knapsack problem.

- if $(n > m)$, assume that the second strategy was chosen. We have $\mathrm{AdHASH}^h_M(x) = \sum_{i=1}^q a_i$ and $\mathrm{AdHASH}^h_M(y) = \sum_{i \in J'} a_i$ where $J' = \{j + n : j \in J\} \cup \{1, \ldots, n\} \setminus \{i_j : j \in J\}$. Let $J'' = \{1, \ldots, q\} \setminus J'$. Since $\mathrm{AdHASH}^h_M(x) = \mathrm{AdHASH}^h_M(y)$ and $n \notin J'$, we have $\sum_{i \in J''} a_i = 0$ and $J'' \neq \emptyset$, i.e., J'' is a solution to the given knapsack problem.

Notice that the two strategy are identical from the viewpoint of of C_2. Therefore K has exactly 1/2 probability to use a strategy that will enable him to solve the given knapsack problem if (C_1, C_2) succeeds in finding a collision. ▮

E The Random Oracle Paradigm

It may be worth saying a few words about the random oracle paradigm we are using. We take this discussion from [BR].

As we have seen, the idea is to use a function h in the schemes, and prove security of the schemes assuming h is a truly random function, or random oracle. Later, in practice, h is set to some fixed function, derived from some standard hash function like SHA-1. The schemes typically (as here) involve other hard functions as well, such as number-theoretic based ones.

It is important to neither over-estimate nor under-estimate what this paradigm buys you in terms of security guarantees. First, one must be clear that this is not standard provable security. The function h that we actually use in the final scheme is not random. Thus the question is: what has it bought us to have done the proof in the first place?

The overly skeptical might say the answer is "nothing." This is not quite true. Here is one way to see what it buys. In practice, attacks on schemes involving a SHA-1 derived h and number theory will *themselves treat h as random.* In other words, cryptanalysis of these "mixed" schemes is usually done by assuming h is random. But then the proofs apply, and indeed show that such attacks will fail unless the underlying number-theoretic problems are easy. In other words, the analysis at least provably excludes a certain common class of attacks.

Of course this doesn't include all attacks. But what it tells us is that to successfully attack schemes proven secure in a random oracle model, the cryptanalyst *must* exploit the structure of a specific hash function h. But when both number-theory and hashing are involved, getting profitable cryptanalytic interaction between h and the number-theory is very hard, because the two problems are so "independent" in structure. However this feeling, although it seems to be true, is heuristic. Certainly it relies very much on the idea that the problems are independent.

This explains why the random oracle model is viewed in [BR] as a "bridge between theory and practice."