G22.3210-001/G63.2170 Introduction to Cryptography November 18, 2008

Lecture 12

Lecturer: Yevgeniy Dodis *Fall 2008*

We saw that ε-universal hash families are very useful in the design of pseudorandom functions (and, thus, message authenticated codes). We will see, however, that we will need stronger hash functions when we design digital signature schemes. Such hash functions are called *collision-resistant* and this lecture is primarily dedicated to their definition and constructions. Interestingly, we will see that CRHFs are also useful in the domain extension of MACs which are not necessarily PRFs.

At the end nof the lecture we will introduce digital signature schemes, and show how CRHFs simplify the design of such signatures via the "hash-then-sign" paradigm. The lecture will conclude with some advanced topics which are completely optional to study.

1 Motivation: Domain Extension of MACs

Recall, if g_s is a PRF, and h_t is ε-universal (for negligible ε), then $f_{s,t}(m) = g_s(h_t(m))$ is a PRF as well. What if g_s is "only" a MAC, and not necessarily a PRF? The proof we had before breaks down. In fact, one can construct and artificial MAC g_s which is *not* a PRF, for which the function $f_{s,t}$ is not secure even with *natural* ε-universal functions, like matrix-vector multiplication. Thus, if we hope for the "hash-then-mac" approach to work much like "hash-then-prf" did, we need a stronger kind of hash function h_t.

It turns out that such strong type of h_t exists, and is they are called *weakly collision-resistant* hash functions (WCRHFs). Intuitively, a WCRHF h_t has the property that the attacker cannot find two distinct inputs $x \neq y$ such that $h_t(x) = h_t(y)$, even *when having oracle access to* $h_t(\cdot)$ (recall, t is a secret key). In other words, for ε-universal hash functions the attacker had to find x and y *without* oracle access to h_t, while here we allow such access. We will not prove it (it's not very hard), but the following the the corresponding theorem for the domain extension of MACs:

Theorem 1 *If g_s is an ℓ-bit* MAC*, and h_t is a* WCRHF *from L bits to ℓ bits, then $f_{s,t}(m) = g_s(h_t(m))$ is an L-bit* MAC.

Intuitively, the attacker A forging the MAC $f_{s,t}$ and producing a forgery m either had to use m such that $h_t(m)$ is "new" (different from $h_t(m_i)$, where m_i are the message MAC'ed by A), or $h_t(m) = h_t(m_i)$, fior some i and $m \neq m_i$. The first case easily leads to the forgery of the MAC g_s (on message $h_t(m)$), while the second case leads to a collision (m, m_i) for h_t.

Unfortunately, most ε-universal hash functions are not weakly collision-resistant (in fact, WCRHFs imply one-way functions, but this is tricky to show). Still, it is possible to construct relatively simple WCRHFs (simpler than PRFs), but this topic is too advanced for this class. Instead, we notice that the domain extension of MACs would be even simpler if we had an even stronger type of function: a (strongly) *collision-resistant* hash function (CRHF). Intuitively, h_t where collisions are hard to find *even when giving the key t to the attacker*! We define such functions in the next section.

2 Collision-Resistant Hash Functions

Let k be the security parameter, and $\mathsf{Gen}(1^k)$ be the generation algorithm which outputs a public key PK, and, if needed, a description of the message space $\mathcal{M} = \mathcal{M}(PK)$ out output space $\mathcal{R} = \mathcal{R}(PK)$. We require that $|\mathcal{M}| > |\mathcal{R}|$. In fact, for simplicity, below we assume $\mathcal{M}(k) = \{0,1\}^{L(k)}$, $\mathcal{R}(k) = \{0,1\}^{\ell(k)}$, where $L(k) > \ell(k)$. However, all the discussion easily generalizes to any domain $\mathcal{M}$ and range $\mathcal{R}$, as long as $|\mathcal{M}| > |\mathcal{R}|$. In particular, as we explain later, in practice $\mathcal{M}$ will be equal to $\{0,1\}^*$ (all finite strings) rather than $\{0,1\}^L$.

Definition 1 A family of functions $\mathcal{H} = \{h_{PK} : \mathcal{M}(PK) \to \mathcal{R}(PK)\}$ generated by Gen is called a family of *collision-resistant hash functions* (CRHFs) if (a) $|\mathcal{M}| > |\mathcal{R}|$, (b) h_{PK} is efficiently computable for any PK, and (c) for any PPT attacker A,

$$\Pr[x \neq x' \ \wedge \ h_{PK}(x) = h_{PK}(x') \mid PK \leftarrow \mathsf{Gen}(1^k), (x, x') \leftarrow A(PK, 1^k)] \leq \mathsf{negl}(k)$$

$\diamondsuit$

To compare with ε-universal hash functions, there the description of the function is chosen at random *after* the attacker committed to $x \neq x'$. In contrast, here the attacker learns the description PK of the CRHF before trying to find a collision. In particular, we had information-theoretic ε-universal function families, while any family of CRHFs must be based on *computational* assumptions. Indeed, since $|\mathcal{M}| > |\mathcal{R}|$ for any public key PK, any PK has some non-trivial (i.e., $x \neq x'$) collisions, so the only reason A cannot find them is because A is computationally bounded.

Application to MACs. Notice, Theorem 1 is really trivial to see with CRHFs in place of WCRHFs. We leave it an an exercise, later proving an anlogous results with digital signatures.

3 Extending the Domain of CRHF

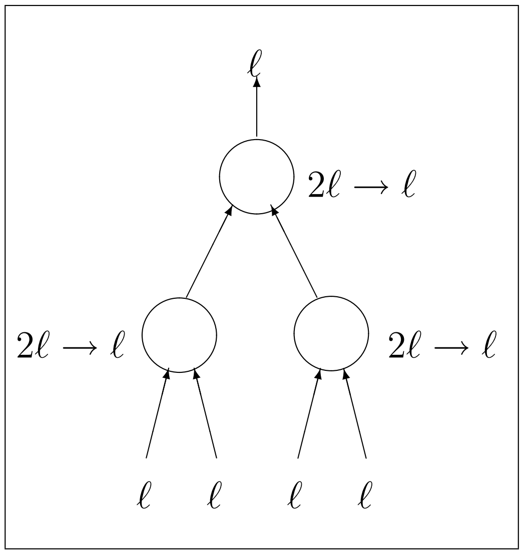
Before constructing specific CRHFs, we study the question of their domain extension.

General Composition Idea. Here is a general idea. If it is too general, you can skip this paragraph and look at our two main examples. Let T be any tree where each internal nodes of this tree has an *ordered* list of its chidren. A valid labeling of this tree corresponds to: (1) associating some string with each leaf of the tree; (2) associating some function with every internal node of the tree; (3) making sure the associations above are *consistent* in the following sense. Starting with the leaves and moving up towards the root, we require for each internal node N the following: if N is associated with a function f from b_1 to b_2 bits and has children $N_1 \ldots N_t$, then sum of lengths of strings $s_1 \ldots s_t$ associated with $N_1 \ldots N_t$ is exactly b_1. If this is true, we associate the string $s = f(s_1, \ldots, s_t)$ with N and move up the tree (thus, the string associated with N has length b_2). At the end, the label of the whole tree is the label of its root. Essentially, we want to ensure that "all the lengths match".

In our compositions, we start with some hash family $\mathcal{H}$, and construct a new hash family $\mathcal{H}'$ with much larger input length, by means of some conveniently chosen tree T. The functions we put in the internal nodes of T will be the hash functions from the corresponding original family $\mathcal{H}$, the labels of the leaves will be parts of the much longer input to the functions in $\mathcal{H}'$, while the label of the tree (the root) is the output of the new hash fucntion.

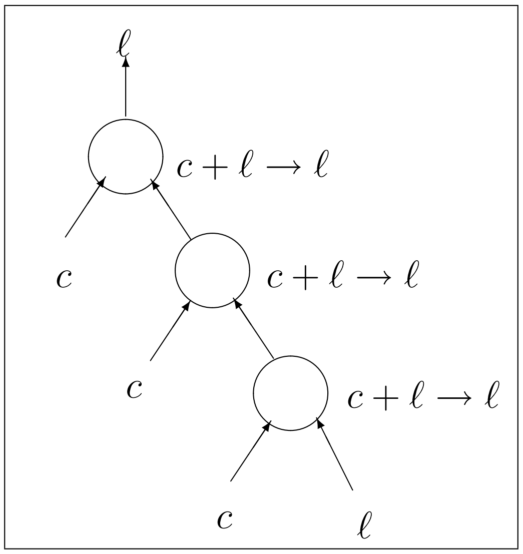
Merkle-Damgård and Merkle Tree. Two main example we will use are given below. In both examples, one chooses a single hash function h from $\mathcal{H}$ at random and uses it at all levels of the tree. However, for the sake of generality and because it will be useful a bit later, we denote the hash function associated with internal node i by h_i, despite the fact that for our current purposes all of them are really equal to the same h.

Merkle Tree



$2^d\ell \to \ell$ *for depth d*

Merkle-Damgård



$nc+\ell \to \ell$ *for depth n*

- **Cascade (or Merkle-Damgård).** Assume $\mathcal{H}$ goes from $\ell + c$ to ℓ bits for some $c > 0$ (in fact, c could be as small as 1). Set $L = \ell n + c$, write input $x \in \{0,1\}^L$ as $x = x_0 \circ x_1 \circ \ldots \circ x_n$, where $|x_0| = \ell$ and each $|x_i| = c$ for $i > 0$. Given functions $h_1 \ldots h_n \in \mathcal{H}$, set

$$H(x_0 \circ x_1 \circ \ldots \circ x_n) = h_n(x_n \circ h_{n-1}(x_{n-1} \circ \ldots \circ h_2(x_2 \circ h_1(x_1 \circ x_0)) \ldots))$$

  Notice that this function family corresponds to a tree which is a long path of depth n. A useful sub-example corresponds to $c = 1$ and $n = \ell - 1$. Then we get H going from 2ℓ to ℓ bits from h_i's going from $\ell + 1$ to ℓ bits each.

  The Merkle-Damgård transform is extensively used in practice. Although it uses a tree which is just a long path, it is very convenient since: (a) it works for any value of $c > 0$, making it very flexible; (b) it allows to process the data in one pass, from left to right (in particular, one does not need to know the length L before one starts); and (c) the lack of parallelism (compared with the second example below) is usually not very important in practice, since the data often comes in sequentially and/or parallel machines are anyway not readily available.

- **Merkle Tree.** A more interesting example comes from the complete binary tree (CBT), when we start from the family from 2ℓ to ℓ bits. Using a complete binary tree of depth d, we get a new family from $2^d\ell$ to ℓ bits.

  The most obvious advantage of the Merkle tree isea is that it allows to process data in parallel, as opposed to the Merkle-Damgård transform which is sequential. However, we already remarked that this is not very important in practice. Instead, the main reason Merkle Tress are extensively used in practice in the following. Assume you have a big file F consisting of 2^d blocks, and you wish to "commit" to its contexts by publishing a collision-resistant hash of the file. For example, you want to store F on an untrusted server, but remember the short hash value $y = H(F)$ of F. Now, you want to retrieve the i-block F_i of F from the server. The server can just send you F_i, but you do not trust the server. Alternatively, the server can send you the entire file F, and you verify that the hash of F is y. Here the server cannot cheat, but the protocol is very wasteful. You only wanted one short block F_i, but had to read all 2^d blocks, and recompute the entire hash function on all of F.

  Merkle trees allows you to do much better. As before, you only store the hash y, which is the root of the Merkle tree. You also let the server store the entire Merkle tree (i.e., the hash values on all the internal nodes, which at most doubles the storage on the server's side). Now, the server can only send you: (a) the block F_i you want; (b) all the internal hash values $y_d = F_i, y_{d-1}, \ldots, y_1, y_0 = y$ on a path from y to F_i; (c) all the "sibling" nodes corresponding to $y_{d-1}, \ldots, y_0$ (i.e., make sure the client gets back both children of $y_0, \ldots, y_{d-1}$; one child is anyway present because of (b), but sending the sibling ensures that the second child is present too). Thus, the server sends a total of $2d + 1$ blocks, which is dramtically less than 2^d. Now, the client just evaluates little h for each of the d internal nodes on the path from F_i to y, and if all the checks are valid, the client knows F_i is correct. This means only d evaluations and only $2d + 1$ blocks, which is logarithmic in the length 2^d of the file.

  To summarize, Merkle trees is extremely useful if one wants to reliably retrieve individual message blocks from some untrusted storage, without the necessity to read the entire file.

Global Collision Implies Local Collision. The main structural property of any tree (including the above examples) is the following.

Lemma 1 *Assume $H(x) = H(x')$ for some distinct $x, x' \in \{0,1\}^L$. Then there is (easy to find) internal node i of the tree (labelled by some $h_i \in \mathcal{H}$) such that: (1) it takes input $x_i \in \{0,1\}^\ell$ when evaluating $H(x)$ and input $x'_i \in \{0,1\}^\ell$ when evaluating $H(x')$; (2) $x_i \neq x'_i$; (3) $h_i(x_i) = h_i(x'_i)$. In other words, a "global collision" in H implies there exists a "local collision" in at least one of the h_i's.*

Proof: Imagine a moving "frontline", described below, and let's see how this frontline evolves in the "world" W of x vs. the "world" W' of x'. Initially, the fronline consists of all the (labels of the) leaves of the tree, so that $F = x$ in W and $F' = x'$ in W'. By assumption $F \neq F'$ originally. At each next step, we take one internal node, adjacent to the frontline (all its children are in the frontline), include it (or rather its label) in the frontline, but remove its children (or rather, their labels) from the frontline. We do it until we reach the fronline which just contains the root. Now, the final fronline is equal to the value of h' (the label of the root). Sicne $H(x) = H(x')$, the final values of the fronline are the same in W and W'. Thus, the fronlines were different at the beginning in W and W', but became the same at the end. Hence, the must be an intermediate step, where we included some node i in the fronline (with function h_i) so that: $F \neq F'$ before including i, but $F' = F$ after including i. But this literally means that the inputs x_i and x'_i to h_i were different (since this is the only way for the frontline to differ originally), but the outputs of h_i are the same. This exactly means we found the local collision requested. □

Concrete Examples. As we already mentioned, for our purposes we will use the same randomly selected function $h \in \mathcal{H}$ at each node of the tree, no matter how the tree looks like. Indeed, assuming $H(x) = H(x')$ for some distinct $x, x' \in \{0,1\}^L$, Lemma 1 implies that there exists some (easy to find) distinct $x_i, x'_i \in \{0,1\}^\ell$ so that $h(x) = h(x')$. Thus, if some A can fund a collision in H (build using the same $h \in \mathcal{H}$), one can use A to find a collision in h itself. It turns out we get the same paramenters irrespective of which "legal" tree we are using. For convenience, below we apply this composition to the "path", the "complete binary tree" ("CBT"), and their combination (first get length doubling by path, then apply CBT).

Theorem 2 *Let $\mathcal{H}$ be a* CRHF *family of fucntions from $\ell + c$ to ℓ bits and key length p. Then there exists the following* CRHF *$\mathcal{H}'$ going from L to ℓ bits, and using the same key h of length p, where the evaluationg of $H \in \mathcal{H}'$ takes $\frac{L-\ell}{c}$ evaluations of h. For concrete examples,*

- *By using the Merkle-Damgård chaining mode, we can achieve $L = nc + \ell$, where evaluation of $H \in \mathcal{H}'$ takes $n = \frac{L-\ell}{c}$ evaluations of $h \in \mathcal{H}$. In paricular, if $c = 1$ we can get $L = 2\ell$ using ℓ evaluations of h.*
- *If $c = \ell$ (i.e., h goes from 2ℓ to ℓ bits), using the Merkle tree of depth d we can achieve $L = 2^d\ell$, where evaluation of $H \in \mathcal{H}'$ takes $(2^d - 1) = \frac{L}{\ell} - 1$ evaluations of $h \in \mathcal{H}$.*
- *Combining the above, if $c \leq \ell$, we can achieve $L = 2^d\ell$, where evaluation of $H \in \mathcal{H}'$ takes $\frac{\ell(2^d-1)}{c} = \frac{L-\ell}{c}$ evaluations of $h \in \mathcal{H}$. In particular, when $c = 1$ we use $(L - \ell)$ evaluations of $h : \{0,1\}^{\ell+1} \to \{0,1\}^\ell$.*

Corollary 2 *Given a fixed* CRHF *$\mathcal{H}$, we can make a new* CRHF *$\mathcal{H}'$ with essentially unbounded input size, without increasing the key size for $\mathcal{H}'$.*

Variable-Lengh Inputs. So far we considered the case of a fixed tree, corresponding to a fixed-length inputs (of length L). Just like for the case of MACs, in practice one want to de able to handle arbitrary-length messages; i.e., $\mathcal{M} = \{0,1\}^*$. We restrict our attention to how to extend the Merkle-Damgård chaining to handle such inputs, since this is what is being used in practice.

Assume we have a collision-resistant compression function h from $\ell + c$ to ℓ bits. We will assume that $c \geq \log L$, where L is the *largest* length of the message being hashed (i.e., all messages are of length at most 2^c). This is not a big restriction, since in practice $c \geq 128$ (usually 512), and 2^{128} is larger than the number of molecules in the universe. Finally, we will assume that input length (whatever it is) is a multiples of block size c: if not, uniquely pad the input to make its length a multiple of c. We set the initialization vector IV to arbitrary ℓ-bit constant (say, 0^ℓ), and consider first the usual cascade construction applied to $m = m_1 \dots m_n$, where $|m_i| = c$:

$$H(m_1 \dots m_n) = h(m_n, h(m_{n-1}, \dots h(m_1, IV), \dots))$$

We immediately notice that this Merkle-Damgård chaining preserves collision-resistance as long as the inputs are encoded in a *suffix-free* form. Namely, no legal input m is a suffix of another input m'. Indeed, if this is not the case, then traversing any collision backward on any $m \neq m'$ eventually leads to distinct inputs colliding wr.t. h in one of the blocks: namely, the first value $j \geq 0$ such that $m_{n-j} \neq m'_{n'-j}$, where n and n' are the number of blocks in m and m', respectively.

What if inputs are not in the suffix-free format? Well, it's easy to make them into such format. Simply add block $m_{n+1} = \langle n \rangle$: namely, add the number of message blocks as the last block of the encoded message. Clearly, if $n \neq n'$, then the last block is different, and we get suffix-freeness. If $n = n'$, the the last block is the same, but at least one of the message blocks is different since $m \neq m'$. This is called *Merkle-Damgård strengthening.*

$$H(m_1 \dots m_n) = h(\langle n \rangle, h(m_n, h(m_{n-1}, \dots h(m_1, IV), \dots)))$$

The nice thing about this method is that one does not have to know the length of the message n before hashing it. Simply keep counter of how many blocks you hashed so far, and when you encounter the end of the file marker, you know n and can apply the Merkle-Damgård strengthing at this point.

To summarize, when we construct CRHFs in the next section, it suffices just to get some level of compression, from which we can efficiently extend the domain to $\{0,1\}^*$ via Merkle-Damgård strengthening.

4 Constructions of CRHFs

It turns out, the existence of CRHFs is a powerful and strong assumption. In particular, we do not know how to build CRHFs from OWFs, OWPs, and even TDPs! In fact, there is some strong theoretical indication that it is impossible to build CRHFs from general TDPs. Luckily, we can construct CRHFs using:

- **Ideal Block Ciphers.** This is a *heuristic* construction (explained later in Section 4.1), but is extensively used in virtually all current hash functions such as SHA-1 or MD5. So we cover it because of its widespread use and the fact that we can give some partial theoretic justification.
- **Claw-Free Permutations.** This is a very strong assumption (studied later in Section 4.2), much stronger than TDPs, for example. Luckily, we can build simple claw-free permutations (CFPs) from all the number-theoretic assumptions we studied so

far, including RSA, factoring and discrete log. In particular, this shows how to build CRHFs from all these number-theoretic assumptions.

- **Number-Theoretic Assumptions.** We already mentioned above that one can build CFPs, and thus, CRHFs, from all the number-theoretic assumptions we studied so far, including RSA, factoring and discrete log. However, in each of these cases one can build more efficient CRHFs but going into the specifics of number theory. This is briefly mentioned in Section 4.3.

To summarize, although the existence of CRHFs is a strong assumption, we can build them effiicently using a variety of techniques, which means that in the sequel we will freely use CRHFs.

4.1 Construction from Ideal Cipher

In practice, suprisingly enough, CRHFs are built from block ciphers. Let $E_s(x)$ denote a block cipher with key x and input x, and $E_s^{-1}(y)$ denotes the corresponding inverse. We let $|x| = |y| = n$ and $|s| = k$. The following construction, called *Davies-Meyer constrcution*, is used quite extensively for most practical hash functions, including SHA and MD5 (the latter recently broken). It builds the following compression function $h : \{0,1\}^{n+k} \to \{0,1\}^n$:

$$h(s, x) = E_s(x) \oplus x$$

The claim is that the DM construction is a good CRHF "in practice". Namely, it is infeasible to find $(s, x) \neq (s', x')$ s.t. $E_s(x) \oplus x = E_{s'}(x') \oplus x'$. This claim might seem puzzling at first, and for a good reason. First, we said that it is unlikely we can construct CRHFs from OWFs, and we know (via Luby-Rackoff) and block ciphers can eventually be built from OWFs. More seriously, we see that the construction does something "un-kosher": the secret key s for the block cipher is simply an input to h *can be chosen by the attacker* trying to find collision! Indeed, it is easy to construct an artificial PRP for which the DM construction is insecure. I.e., assume $E_0(0) = 0$ and $E_1(1) = 1$. This does not contradict the PRP assumption, since it is very unlikely that a random key s will be 0 or 1. Thus, indeed, we can never prove security assumpting that our block cipher E is "only" a PRP.

Nevertheless, in practice people try not to chose block cipher with intentionally "weak" keys, like the artificial counter-example above. In fact, ideally, we hope that a good block-cipher should behave like a random permutation *for every key*. Unfortunately, this assumption is too strong, and it is hard to make it formal: once the code of the block cipher is fixed, it is *not* random for *every* key. In fact, for every particular key the permutation is fully determine. Still, in practice the cipher seems to be "good enough" to avoid any "silly counter-examples" like above.

Because of these considerations, theoreticians and practitioners converged on the following abstract model of block ciphers, called the *ideal cipher model* (ICM). The best way to imagine the ICM to to pretend that there exists a trusted third party Zak, who chose in his head 2^k *truly random permutations* on n bits, $E_s(\cdot)/E_s^{-1}(\cdot)$, for every $s \in \{0,1\}^k$. Now, whenever a party — either honest one (like Alice and Bob), or malicious (like Eve) — need to compute $E_s(x)$ or $E_s^{-1}(y)$, they simply ask Zak, and he gives them the answer.

Notice, this makes perfect mathematical sense, and will allow us to prove theorems about the ICM. On the other hand, in practice there is no Zak, and people simply use a good enough block cipher, like AES. And the hope is that the block cipher is so good, that no party — even Eve — can do much more than honestly evaluate it on a bunch of points, and more or less assume that the result will be totally unpredictable. i.e., despite having the code of E, Eve cannot underastand the complexity of the code and can only run the primitive forward/backward, always expecting a random(-looking) result.

DM is Collision-Resistant in ICM. If this was too abstract, let us semi-formally argue the collision-resistance of the DM construction in the ICM.

Theorem 3 *$h(s,x) = E_s(x) \oplus x$ is a collision-resistant hash function in the ICM.*

Proof: Assume there exists an attacker A, who expects oracle access to $E_{\cdot}(\cdot)$, $E_{\cdot}^{-1}(\cdot)$ (i.e., can chose both the key and the input/output), and find a collision $(s,x) \neq (s',x')$ for h with probability ε. We will prove that ε must be low, using an *information-theoretic argument* (this is possible because we are using ICM!). We do a sequence of games, and let p_i be the probability the attacker still finds collision in Game i.

- *Game 0.* The orginal game between A and Zak, whose chose a totally random ideal cipher.

- *Game 1.* This is lazy Zak, but otherwise the same as non-lazy Zak. Namely, instead of choose the cioher in full at the beginning, he defines it incremetally, as requested by A. He keeps a table T of tuples (s,x,y,z), which correspond to defined values $E_s(x) = y$ (and $E_s^{-1}(y) = x$), where for conveneince Zak also stores $z = x \oplus y$. Initially T is empty (nothing defined), but T grows with each query of A. namely, for forward query (s,x), if a tuples (s,x,y,z) is in the table, give back z. (Same for backways query (s,y), returning x.) But if such tuples is not there, choose a random y distinct from all y' already defined for the same key s (i.e., if $E_s(x') = y'$, then exclude y' from consideration). Same for E_s^{-1}. Obviously, p_1 is still ε, simnce this is the same game.

- *Game 2.* Identical to Game 1 above, except Zak does not exclude any y when defining a random forward query (or x for backward query). If A makes q queries, for each such query Zak could only run in trouble (violating permutation structure) with probability at most $q/2^n$, so the total probability of him running into trouble after all queries is at most $q^2/2^n$. Thus, $|p_2 - p_1| \leq q^2/2^n$.

- *Game 3.* Identical to Game 2 above, except Zak checks if the new values z_i that he puts in the table "for fun" ever collide. If so, he stops and loses the game. For each new forward query (s,x) or backward query (s,y) which defines a new z, we notice that z is defined at random: either one chooses random y and sets $z = x \oplus y$, or chooses random y and again sets $z = x \oplus y$. Thus, the chance that any of the z's repeat is simply a birthday bound $q^2/2^n$. Hence, $|p_3 - p_2| \leq q^2/2^n$.

- *Game 4.* Identical to Game 3 above, except when A outputs a claimed collision $(s,x) \neq (s',x')$, we make sure that we already defined the values $E_s(x)$ and $E_{s'}(x')$. I.e., that T contains tuples (s,x,y,z) and (s',x',y',z') for some y,z,y',z'. If this is not so, pretend that A asks Zak (from Game 3) to evaluate these two more queries for him. Clearly, this is the same as Game 3, except Zak could "screw up" answering the last two queries, which happens with probability $4q/2^n$ (each query can mess up with probability at most $q/2^n$, by the analyses of Games 2 and 3). Hence $|p_4 - p_3| \leq 4q/2^n$.

But now, let us argue that $p_4 = 0$. Indeed, in Game 4 both inputs (s, x), (s', x') have two distinct entries in T, for which z and z' are distinct (by Game 3). Thus,

$$h(s, x) = E_s(x) \oplus x = x \oplus y = z \neq z' = x' \oplus y' = E_s(x') \oplus x' = h(s', x')$$

Tracking back, we see that $\varepsilon = p_0 \leq (2q^2 + 4q)/2^n$, which is negligible. $\square$

We saw that the above ICM proof was information-theoretic, despite the fact that CRHFs imply OWFs. This is because ICM model is very strong and should be used with extreme caution!

4.2 Construction from Claw-Free Permutations

We now turn to constructions in the standard model. W estart with a general construction. It uses a notion of *claw-free permutations*. We define them below

Definition 2 A family of functions $\mathcal{F} = \{(f_{0,PK}, f_{1,PK}) : \mathcal{M}(PK) \to \mathcal{M}(PK)\}$ generated by Gen is called a family of *claw-free permutations* (CFPs) if (a) $f_{0,PK}$, $f_{1,PK}$ are efficiently computable *permutations* for any PK; and (b) for any PPT attacker A,

$$\Pr[x \neq x' \ \wedge \ f_{0,PK}(x) = f_{1,PK}(x') \mid PK \leftarrow \mathsf{Gen}(1^k), (x, x') \leftarrow A(PK, 1^k)] \leq \mathsf{negl}(k)$$

$\diamondsuit$

Notice, we will omit PK from our notation, to avoid messiness, and also assume $|\mathcal{M}| \approx 2^n$ for concreteness. Before giving examples of CFPs, let us right away construct a CRHF from a family of CFPs $\{(f_0, f_1)\}$. Given any (fixed, but possibly huge) parameter L, view the message M as a pair (x, m), where $x \in \mathcal{M}$ and $m = m_1 \ldots m_L$, and let

$$h(x, m) = f_{m_L}(\ldots f_{m_2}(f_{m_1}(x)) \ldots)$$

We claim that h is a CRHF from roungly $L + n$ bits to n bits. In particular, it already compresses for $L = 1$. Also notice we can process the input in an on-line manner. Now, take any two messages $(x, m) \neq (x', m')$ and assume $h(x, m) = h(x', m') = z$. We have two cases:

Case 1: Assume $m = m'$. Notice, although f_0^{-1} and f_1^{-1} might not be efficient, they are well defined mathmatically. In particular, $x = f_{m_1}^{-1}(\ldots f_{m_L}^{-1}(z) \ldots)$, $x' = f_{m'_1}^{-1}(\ldots f_{m'_L}^{-1}(z) \ldots)$. Thus, if $m = m'$, we get $x = x'$ as well.

Case 2: Assume $m \neq m'$. Let i be the smallest index such that $m_i \neq m'_i$ (but $m_j = m'_j$ for $j > i$). Say, $m_i = 0$, $m'_i = 1$. Let $z' = f_{m_{i+1}}^{-1}(\ldots f_{m_L}^{-1}(z) \ldots)$, $x_0 = f_{m_{i-1}}(\ldots f_{m_1}(x) \ldots)$, $x_1 = f_{m'_{i-1}}(\ldots f_{m'_1}(x) \ldots)$. Then, $f_0(x_0) = z' = f_1(x_1)$, so (x_0, x_1) form a claw (which is efficiently computable, see above formula).

Notice, the construction easily generalizes for any suffix-free messages. Thus, by appending a block containing $\langle L \rangle$, we can maintain suffix-freeness, and still preserve the on-line nature of the function!

Examples of CFPs. We now argue that standard one-way permutation constructions, such as exponentiation mod p, squaring mod n, and RSA, easily yield CFPs. We demondtrate it for exponentiation, leaving the other two cases as simple exercises (done analogously to exponentiation).

The public key PK here consists of a prime p, a generator g of any subgroup $\mathcal{G}$ of $\mathbb{Z}_p^*$ (including $\mathbb{Z}_p^*$ itself) where discrete log is hard, and a random element $y \in \mathcal{G}$. We define $f_0(x_0) = g^{x_0} \bmod p$, and $f_1(x_1) = y \cdot g^{x_1} \bmod p$. Now, if $f_0(x_0) = f_1(x_1)$, then $g^{x_0} = yg^{x_1} \bmod p$, meaning that $y = g^{x_0 - x_1} \bmod p$, meaning that $(x_0 - x_1)$ is a discrete log of y. Since y was random, it should be hard to compute discrete log of y. Thus, it must be infeasible by the attacker to compute x_0, x_1 above.

Notice, this construction is elegant, but very inefficient: it roughly requires an exponentiation per bit of the input. Also, for $\mathcal{G} = \mathbb{Z}_p^*$, we indeed have compression starting from $L = 1$. However, for smaller subgroups $\mathcal{G}$, the output is an element of $\mathcal{G}$, and it might not be easy to compress it to $\log |\mathcal{G}|$. Thus, we prefer to either use $\mathbb{Z}_p^*$ itself, or let $p = 2q + 1$, and consider the prime order q subgroup of quadratic residues mod p (like we did for ElGamal). This is compressing for any $L \geq 2$, and can be made compressing even for any $L = 1$ using the bijection between $QR(p)$ and $\mathbb{Z}_q$ we introduced in the lecture covering the ElGamal encryption.

4.3 Optimized Constructions using Number Theory

We can optimize the above generic construction using the specific algebraic properties of concrete one-way permutations, such as exponentiation and RSA. We only do it for the former, mentioning that the other examples can also be optimized.

Construction from Discrete Log. For that, let us recall the CFP construction based on the discrete log, and let us look at it when $L = 1$. In this case, the input to h is a message consisting of a bit b and an integer $x \in \{1 \ldots |\mathcal{G}|\}$, and the output is $h(b, x) = y^b g^x \bmod p$ (yielding g^x for $b = 0$ and yg^x for $b = 1$). For elegancy sake, we will only consider the cases $\mathcal{G} = \mathbb{Z}_p^*$, in which case the output lies in $\mathbb{Z}_p^* \equiv \mathbb{Z}_{p-1}$, and $\mathcal{G} = QR(p)$, where $p = 2q + 1$ and q is prime, in which case we can compress the output to lies in $QR(p) \equiv \mathbb{Z}_q$.

Looking at the formula above, one may wonder why we restrict b to be a bit. Indeed, let us not assume that b is a bit, and see if we can argue if $y^b g^x \bmod p$ is a CRHF. Take any $(b, x) \neq (b', x')$ such that $y^b g^x = y^{b'} g^{x'}$. This means $y^{b-b'} = g^{x'-x}$. Now, if $b = b'$, then $x = x'$, so we conclude that $b \neq b'$. We would then like to conclude that the discrete log of y is equal to $(x' - x) \cdot (b - b')^{-1} \bmod |\mathcal{G}|$. But this is only true if $b - b'$ is relatively prime to $|\mathcal{G}|$. For $\mathcal{G} = \mathbb{Z}_p^*$, $|\mathcal{G}| = p - 1$, which is composite, so we cannot elegantly conclude that $(b - b')$ is invertible. There are ways to deal with this problem, but they are messy. Instead, we will look at the simpler case of $\mathcal{G} = QR(p)$, where $p = 2q + 1$. In this case, $|\mathcal{G}| = q$ is prime, so we can indeed compute the discrete log of y as $(x' - x) \cdot (b - b')^{-1} \bmod q$.

In particular, we can make both $x, b \in \mathbb{Z}_q$. However, the result will be more "symmetric" if we rename x, b into x_1, x_2 and g, y into g_1, g_2. We get the following elegant family of two-to-one CRHF. Choose random k-bit prime $p = 2q + 1$, where q is prime. Let $\mathcal{G} = QR(p)$, and notice that $\mathcal{G} \equiv \mathbb{Z}_q$ (with efficient mapping both ways, see ElGamal lecture). Under this isomporphism, define the following hash function $h_{p,g_1,g_2} : \mathbb{Z}_q^2 \to \mathbb{Z}_q$. The public key of h consists of p and two random generators $g_1, g_2 \in \mathcal{G}$. Then, for $x_1, x_2 \in \mathbb{Z}_q$, let $h_{p,g_1,g_2}(x_1, x_2) = g_1^{x_1} g_2^{x_2} \bmod p$ (with the result in $\mathcal{G}$ mapped back to $\mathbb{Z}_q$).

Theorem 4 *$\mathcal{H} = \{h_{p,g_1,g_2}\}$ above is a two-to-one* CRHF *from roughly $2k$ to k bits, under the discrete log assumption.*

Notice, one can naturally extend the compression ratio to t-to-1 of this construction, by using more generators $g_1 \dots g_t$ (homework?). Also, one can have similar optimized constrctions for the RSA and squaring functions.

Direct construction from factoring. Of course, there are other constructions of CRHFs. Here we mention one simple one, based on factoring. Here one chooses the modulus $n = (2p + 1)(2q + 1)$. Also, as part of the public key, one chooses a random $y \in QR(n)$. Notice, $|\mathbb{Z}_n^*| = 2p \cdot 2q = 4pq$, and $|QR(n)| = |\mathbb{Z}_n^*|/4 = pq$. Thus, the order of y is pq (with high probability). Now, we define the following functions h over the integers $m \in \mathbb{Z}$: $h(m) = y^m \bmod n$. Now, if $h(a) = h(b)$, and $a \neq b$, then $a - b$ must divide pq, which is the order of y, and be different from 0 (since $a \neq b$). Hence, $4(a - b)$ must divide $\varphi(n)$. However, it is well known that a non-zero multiple of $\varphi(n)$ is enough to factor n.

5 Digital Signatures

The remainder of this lecture is dedicated to *public-key signature schemes (*PKS*)*, which are the public-key counterparts of the *message authentication codes (*MAC*)* that we studied earlier.

Motivation and intuition. The use of signatures as a form of authentication in the "real world" is very old and widespread. It is based on the assumption that it is very hard to emulate one's handwriting well enough or to modify a document so that differences cannot be detected. If one accepts that, signing is then a very efficient way of ensuring that a given public document bears one's approval.

"Physical" signatures must be reproducible by the signer (that is, every person must have a definite procedure for signing as often as needed) and recognizable by others (also by means of some definite procedure) . Their usefulness comes from the fact that lots of entities (e.g. the government, banks, credit card companies, family and friends) can recognize what one's signature looks like (i.e. one's signature is known by the *public*) and yet they cannot forge it.

In this class we discuss the computational counterpart of "physical" signatures, which are called *digital signatures*. Those are intended to provide the *sender* with the means to authenticate his/her *messages* (here understood to be any information he/she intends to make available to the world) in a way that *can be checked by anyone* but that *cannot be copied by others*.

The message authentication codes (MAC) that we studied last class do not quite fit into the niche of digital signatures. For they are *secret-key authentication schemes* and implicit in that concept is that all parties that share the secret information (which is necessary for authenticity verification) must be trustworthy if one does not want to lose all hope for security. It is then clear that anything that deserves the name *digital signature* should be a *public-key authentication scheme*: even people in which one does not trust completely should be able to check the authenticity of one's signature. That is, one does not want to impose any restrictions on the parties that may want to verify the authenticity of one's signature. Of course, the signing algorithm must use secret information (that is, a *secret key*), which roughly corresponds to one's unique way of signing.

5.1 Basic Definition

In this subsection we define the notion of a *public-key signature scheme* as a public-key analog of MAC and then present a definition of security for it. $\mathcal{M}$ is the message space (e.g. $\mathcal{M} = \{0,1\}^k$ or $\mathcal{M} = \{0,1\}^*$).

DEFINITION 3 [Public-Key Signature Scheme] A **Public-Key Signature Scheme** (PKS) is a triple (Gen,Sign,Ver) of PPT algorithms:

a) The key generating algorithm Gen outputs the secret (private) and verification (public) keys: $(SK, VK) \leftarrow \mathsf{Gen}(1^k)$.

b) The message signing algorithm Sign is used to produce a signature for a given message: $\sigma \leftarrow \mathsf{Sign}_{SK}(m)$, for any $m \in \mathcal{M}$.

c) The signature verification algorithm checks the correctness of the signature: $\mathsf{Ver}_{VK}(m,\sigma) \in \{accept, reject\}$

The correctness property must hold: $\forall m, \quad \mathsf{Ver}_{VK}(\mathsf{Sign}_{SK}(m)) = \text{accept}.$ $\diamondsuit$

Remark 1 *One can also consider* **stateful** PKS*; we shall encounter those towards the end of the lecture.*

Remark 2 *We shall adopt the convention that VK is a substring appended at the end of SK (i.e. SK contains in VK) whenever this is useful. Of course, this entails no loss of generality.*

As in the case of a MAC, we can consider the notions of existential or universal unforgeability against VK-only, random-message or chosen-message attacks. To assume that an adversary might be able to query the receiver of the messages to check the validity of given pairs (m,σ) makes no difference in the present case, as long as the adversary has the public key, he can test that by himself. Therefore, the natural counterpart of the standard notion of security for MAC in the present context is:

DEFINITION 4 [Standard notion of security for PKS] A PKS (Gen, Sign, Ver) is said to be secure, that is, existentially unforgeable against chosen-message attack (CMA) if for all PPT A we have that

$$\Pr(\mathsf{Ver}(m,\sigma) = accept \mid (SK,VK) \leftarrow \mathsf{Gen}(1^k), (m,\sigma) \leftarrow A^{\mathsf{Sign}_{SK}}(VK)) \leq \mathsf{negl}(k)$$

where A cannot query the oracle Sign_{SK} on the message string m it outputs. $\diamondsuit$

Weaker Notions. We will also study weaker notions of signatures as we try to satisfy the ambitious definition above. They vary according to the attack capabilities and the goal of the attacker:

- **Capabilities of $\mathcal{A}$.** Currently $\mathcal{A}$ is capable of launching chosen message attack: i.e., he has unrestricted access to the signing oracle. We can place a restriction of the number of times q that $\mathcal{A}$ is allowed to call the signining oracle. Currently, $q = \mathsf{poly}(k)$. If $q = 0$, we get *no message* or *VK-only* attack. When $q = 1$ (this is an important case we study later), we get what is called *one-time signature*: the signer can securely sign at least one message and be sure no other signature can be forged. One can also restrict adaptivity of $\mathcal{A}$ (i.e., $\mathcal{A}$ cannot select the messages whose signature he sees, or has to select all messages at once), but we will not study these variants.
- **Goal of $\mathcal{A}$.** Currently, the goal of $\mathcal{A}$ is *existential unforgeability.* Namely, $\mathcal{A}$ succeeds as long as he forges a new signature, irrespective of hoe "ridiculous" the message m he is forging is. A more ambitious goal is to forge a signature of *any given message* with non-negligible probability. There are two flavors of it. More formally, $\mathcal{A}$ is given a *random* message and succeeds if he can sign this message with non-negligible probability. A signature scheme secure against this variant is called *universally unforgeable.* Clearly, existential unforgeability is much more desirable than universal unforgeability.

5.2 "Hash-then-Sign" Method

We develop the following simple "hash-and-sign" method, which illustrates that, using CRHFs, we only need to construct secure signature schemes on "short" domains, and automatically get secure signatures on "long" domains. The lemma below work for both regular ("many-time") and one-time signature scheme: the former case being extensively used in practice, and the latter will be used by us in the the Naor-Yung construction from the next lecture. For concreteness, the rpoofs below is for regular (multi-time) signature scheme, since the one-time case will be a special case.

Lemma 3 (Secure schemes with CRHF) *If $\mathcal{H} = \{h_s\}$ is a CRHF and $\mathsf{SIG}' = (\mathsf{Gen}', \mathsf{Sign}', \mathsf{Ver}')$ is a (one-time) secure signature scheme for ℓ-bit messages, then the signature scheme $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ defined below is (one-time) secure for L-bit messages:*

a) $SK = SK'$, $VK = (VK', h)$, where $(SK', VK') \leftarrow \mathsf{Gen}'(1^k)$ and $h \leftarrow \mathcal{H}$.

b) $\mathsf{Sign}_{SK}(m) = \mathsf{Sign}'_{SK'}(h(m))$.

c) $\mathsf{Ver}_{VK}(m, \sigma) = \mathsf{Ver}'_{VK'}(h(m), \sigma)$.

Proof: Assume that Lemma 3 is false for some SIG', that is, there exists an adversary A such that

$$\Pr(\mathsf{Ver}'_{SK'}(h(m), \sigma) = 1 \mid (SK', VK') \leftarrow G(1^k), h \leftarrow \mathcal{H}, (m, \sigma) \leftarrow A^{\mathsf{Sign}_{SK'}}(VK', h)) = \varepsilon$$

where A queried the oracle on messages $m_1 \dots m_t$, and, when successful, outputs the signature σ of $m \notin \{m_1, \dots, m_t\}$. Then at least one of the following events happens with non-negligible probability $\varepsilon/2$:

(a) $h(m) \in \{h(m_1), \dots, h(m_t)\}$.

(b) $h(m) \notin \{h(m_1), \dots, h(m_t)\}$.

In case (a), we can break the collision-resistance property of $\mathcal{H}$. Indeed, it implies that for some i, $h(m) = h(m_i)$, while by assumption $m \neq m_i$, so m and m_i form a collision. In case (b), we break the security of our original signature SIG'. Indeed, $h(m)$ is a "new message" w.r.t. SIG', since A only saw $\mathsf{Sign}(m_i) = \mathsf{Sign}'(h(m_i))$, so A managed to forge a new signature. Translating this into a formal proof (i.e., building the actual B breaking SIG') is straightforward. Indeed, B picks its own $h \leftarrow \mathcal{H}$, and simulates oracle calls to $\mathsf{Sign}(m_i)$ by oracle calls to $\mathsf{Sign}'(h(m_i))$. When A forges (m, σ), B outputs its own forgery $(h(m), \sigma)$. $\square$

6 Advanced: Universal One-Way Hash Functions

This material is optional and was not covered in class. You can skip it and move to the next lecture.

Definition 5 A family of functions $\mathcal{H} = \{h_{PK} : \mathcal{M}(PK) \rightarrow \mathcal{R}(PK)\}$ generated by Gen is called a family of *universal one-way hash functions* (UOWHFs) if (a) $|\mathcal{M}| > |\mathcal{R}|$, (b) h_{PK} is efficiently computable for any PK, and (c) for any PPT attacker $A = (A_1, A_2)$,

$$\Pr[x \neq x' \wedge h_{PK}(x) = h_{PK}(x') \mid (x, st) \leftarrow A_1(1^k), PK \leftarrow \mathsf{Gen}(1^k), x' \leftarrow A_2(PK, st)] \leq \mathsf{negl}(k)$$

$\diamondsuit$

6.1 "Hash-then-Sign" with UOWHFs

Lemma 4 (Secure schemes with UOWHF) *If $\mathcal{H} = \{h_s\}$ is a UOWHF and $\mathsf{SIG}' = (\mathsf{Gen}', \mathsf{Sign}', \mathsf{Ver}')$ is a (one-time) secure signature scheme for $(\ell + p)$-bit messages, then the signature scheme $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ defined below is (one-time) secure for L-bit messages:*

a) $SK = SK'$, $VK = VK'$, where $(SK', VK') \leftarrow \mathsf{Gen}'(1^k)$.

b) $\mathsf{Sign}_{SK}(m) = (h, \mathsf{Sign}'_{SK'}(h \circ h(m))$, where $h \leftarrow \mathcal{H}$ and $\circ$ is concatenation.

c) $\mathsf{Ver}_{VK}(m, (h, \sigma)) = \mathsf{Ver}'_{VK'}(h \circ h(m), \sigma)$.

Proof: Assume that Lemma 4 is false for some SIG', that is, there exists an adversary A such that

$$\Pr(\mathsf{Ver}'_{SK'}(h \circ h(m), \sigma) = 1 \mid (SK', VK') \leftarrow G(1^k), (m, (h, \sigma)) \leftarrow A^{\mathsf{Sign}_{SK'}}(VK')) = \varepsilon$$

where A queried the oracle on messages $m_1 \dots m_t$, and, when successful, outputs the signature (h, σ) of $m \notin \{m_1, \dots, m_t\}$. Let also (h_i, σ_i) denotes the signature of m_i returned by the oracle (i.e., each h_i is truly random, even though h used in the forgery could be chosen by A arbitrarily). Then at least one of the following events happens with non-negligible probability $\varepsilon/2$:

(a) $h \circ h(m) \in \{h_1 \circ h_1(m_1), \ldots, h_t \circ h_t(m_t)\}$.

(b) $h \circ h(m) \notin \{h_1 \circ h_1(m_1), \ldots, h_t \circ h_t(m_t)\}$.

In case (a), we can break the universal one-wayness property of $\mathcal{H}$. Indeed, it implies that for some i, $h = h_i$ and thus $h_i(m) = h_i(m_i)$, while by assumption $m \neq m_i$. Thus, if we set $x_0 = m_i$, we get that $h = h_i$ was chosen at random *after* x_0 was chosen by A, and we can set $x_1 = m$, thus creating a collision $x_0 \neq x_1$ for this randomly selected h_i. Translating this into a formal proof is easy and is omitted.

In case (b), we break the security of our original signature SIG'. Indeed, $h \circ h(m)$ is a "new message" w.r.t. SIG', since A only saw $\mathsf{Sign}'(h_i \circ h_i(m_i))$, so A managed to forge a new signature. Translating this into a formal proof (i.e., building the actual B breaking SIG') is straightforward. Indeed, B simulates oracle calls to $\mathsf{Sign}(m_i)$ by picking a random $h_i \leftarrow \mathcal{H}$ and, getting $\sigma_i = \mathsf{Sign}'(h_i \circ h_i(m_i))$ from its own oracle, and returning (h_i, σ_i). When A forges $(m, (h, \sigma))$, B outputs its own forgery $(h \circ h(m), \sigma)$. $\square$

6.2 Composition for UOWHF's

The situation is a bit more difficult with UOWHF's. The reason is the following. Assume $\mathcal{H}'$ (build from $\mathcal{H}$ using some tree, where we are not specifying yet how to select the h_i's) is not a UOWHF. Then some A can specify $x \in \{0,1\}^L$, and after $h' \in \mathcal{H}'$ is selected, A can collisde x with some x'. We would like to construct B that breaks the fact that $\mathcal{H}$ is UOWHF. First, B must select some $x_i \in \{0,1\}^\ell$ *before* $h = h_i$ is selected. Probably, B should use A to get $x \in \{0,1\}^L$. The question is: how to use x to produce the needed x_i? If B randomly selects h', then it can use A to find a local collision (x_i, x_i') to some $h_i \in \mathcal{H}$ (by Lemma 1). But then it's too late: $h = h_i$ is already selected, and B has to compute x_i *before* h_i is selected. Well, B can guess the internal node i that will produce the collision (and has non-negligible chance of being correct). Thus, if B can use x to compute the input x_i to the h_i *without selecting* h_i yet, we will be done. But if we use the same h at all nodes i, B cannot compute this x_i since it will have to select $h = h_i$ for that!

On the other extreme, if all the h_i are independently sampled from $\mathcal{H}$, B can succeed with ease. Indeed, it only selects the random h's that are needed to compute the input to h_i, without yet selecting h_i. This way it can compute x_i, then when a random h is selected, it can set $h_i = h$, select the remaining functions, and then use A with Lemma 1 to find the collising x_i' (provided its guess for i is correct). The argument above is correct, but the reduction is very wasteful in terms of the key: for each internal node i we have to choose a new h_i! Can we do better? The answer is positive. Notice, in the argument above we only ran into trouble if the same h was used between a node i and some of its descendant j: $h_i = h_j$. In this case, if i was the local node "resposible" for the "global" collision, computing the input x_i to node i (from the global input x) requires to compute $h_j = h_i$ along the way. And B cannot do this since h_i cannot be selected as of yet. On the other hand, if no node shares the same seed with its descendant, we can easily complete the argument, as before.

This suggests the following more efficient way to select h's. Assume our tree has depth d. Then we must[1] use at least d distinct independent h's. On the other hand, d such h's indeed suffice: simply use a different h at each level of the tree, and let all the nodes at depth i use the same h_i. We get

Theorem 5 *Let $\mathcal{H}$ be a* UOWHF *family of fucntions from ℓ_1 to ℓ bits and key length p. Assume $\mathcal{H}'$ from L to ℓ bits is constructed from $\mathcal{H}$ by using a "legal" tree T of depth d by selecting an independent function h_i for each level of T. Then $\mathcal{H}'$ is a* UOWHF *family with key length $p' = dp$, where evaluation of h' takes $(L-\ell)/(\ell_1-\ell)$ evaluations of various $h_i \in \mathcal{H}$. For concrete examples,*

- *If $\ell_1 = \ell + c$ for some $c > 0$ and T is a path of depth n, we can achieve $L = nc + \ell$ and $p' = pn = \frac{p(L-\ell)}{c}$, where evaluation of $h' \in \mathcal{H}'$ takes $n = \frac{L-\ell}{c}$ evaluations of $h \in \mathcal{H}$. In paricular, if $c = 1$ we can get $L = 2\ell$, $p' = \ell p$ using ℓ evaluations of h.*
- *If $\ell_1 = 2\ell$ and T is a CBT, we can achieve $L = 2^d\ell$ and $p' = d\ell$, where evaluation of $h' \in \mathcal{H}'$ takes $(2^d - 1) = \frac{L}{\ell} - 1$ evaluations of $h \in \mathcal{H}$.*
- *Combining the above, if $\ell_1 = \ell + c$, we can achieve $L = 2^d\ell$, and $p' = \frac{p\ell d}{c} = \frac{p\ell\log(L/\ell)}{c}$ where evaluation of $h' \in \mathcal{H}'$ takes $\frac{\ell(2^d-1)}{c} = \frac{L-\ell}{c}$ evaluations of $h \in \mathcal{H}$.*

The above result says that if two trees T_1 and T_2 yield the same output length L, we should select (provided we use our technique) the tree with smaller depth. More precisely, given ℓ, ℓ_1 and L, we should select the smallest depth "legal" tree with this parameters. It is easy to see that $d = \Omega(\log L/\ell)$ (proof omitted), so the last item in Theorem 5 is nearly optimal. In fact, we mainly care about the dependence of our key size on the input length L (in applications, ℓ_1, ℓ and p can be thought as fixed). Thus, the optimal dependance of of d on L is logarithmic. We summarize this in

Corollary 5 *Given a fixed* UOWHF *$\mathcal{H}$ with fixed parameters ℓ_1, ℓ, p, we can build a* UOWHF *$\mathcal{H}'$ with input size L and output size ℓ, so that the key size p' of $\mathcal{H}'$ grows proportionally to $\log L$.*

We remark that the above composition of UOWHF's is not the best that one can do, but it is nearly optimal, and certainly good enough for our purposes.

6.3 Construction of UOWHF

It turns out that UOWHF's are equivalent to OWF's. The fact that they imply OWF's is left as a homework. The converse implication from OWF's is more interesting, but also much more difficult. We will give a simpler construction from OWP's instead. As we stated, the construction will shrink by only 1 bit: $L = k$, $\ell = k - 1$. Even that will be non-trivial, but we will later see how to have a better tradeoff.

So let f be a fixed OWP (or, more generally, it can be chosen at random from a family of OWP's and fixed). We will use the following auxiliary function family $Chop = \{g_a :$

[1] Meaning if we want to follow this specific proof technique. Of course, there could be, and in fact *there are*, other composition techniques for UOWHF's.

$\{0,1\}^l \to \{0,1\}^{k-1}\}$. This family will be defined over the finite field F having 2^k elements (if this is too abstract, you can think for of F as being $\mathbb{Z}_p$, where p is some k-bit prime). Here each function $g_{a,b}$ will be given by a non-zero element $a \in F\backslash\{0\}$, and will be defined as $g_a(y) = \mathit{chop}(ay)$, where ay is computed in F (and takes k bits to represent), while $\mathit{chop}(ay)$ simply deletes the last bit of the representation of ay, thus truncating it to the needed $(k-1)$ bits. This might seem complicated, but we will only be using the following two elementary properties of the family Chop.

(1) For every $z \in \{0,1\}^{k-1}$, and every $g_a \in \mathit{Chop}$, there are exactly two distinct points y_0 and y_1 such that $g_a(y_0) = g_a(y_1) = z$.

(2) The following, very strange method, nevertheless select a *uniformly random* function $g_a \in \mathit{Chop}$. First, fix *any* adversarilly chosen value $y_0 \in \{0,1\}^k$. Then select y_1 *at random*. Next, choose a at random, but *subject to* $g_a(y_0) = g_a(y_1)$. More precisely, if $y_0 = y_1$, choose random non-zero a, else set $a = 1/(y_0 - y_1)$. Output g_a as your function.

To verify (1), notice that $y_0 = z0/a$ and $y_1 = z1/a$, where $z0$ and $z1$ are completions of z corresponding to the chopped bit being 0 or 1. Thus, all the functions g_a are 2-to-1. To verify (2), first notice that g_a is uniform provided the unlikely event $y_0 = y_1$ happens (in any event, we can ignore this event since it happens with negligible). Else, if $y_0 \neq y_1$, we get $a = 1/(y_0 - y_1)$ is also random and non-zero since $(y_0 - y_1)$ is random and non-zero. To see that the latter indeed has $g_a(y_0) = g_a(y_1)$, notice that $\mathit{chop}(ay_0) - \mathit{chop}(ay_1) = \mathit{chop}(ay_0 - ay_1) = \mathit{chop}(1) = 0$.

Now we can construct our UOWHF $\mathcal{H} = \{h_a : \{0,1\}^k \to \{0,1\}^{k-1}\}$, where $h_a(x) = g_a(f(x))$ (recall that f is a OWP).

Theorem 6 *$\mathcal{H}$ above is a* UOWHF*.*

Proof: Assume $\mathcal{H}$ is not a UOWHF. Thus, there exists $x_0 \in \{0,1\}^k$ and an adversary A, such that when $a \neq 0$ is selected at random, $A(x_0, a)$ outputs $x_1 \neq x_0$ such that $g_a(f(x_0)) = g_a(f(x_1))$ with probability ε.

Using this A, we construct B that inverts our OWP f. B gets an input $y = f(x)$ for a randomly chosen (unknown) x. B sets $y_0 = f(x_0)$ and $y_1 = y$. Then B uses property (2) above to sample the function g_a subject to $g_a(y_0) = g_a(y_1) = z$. Notice, since x was random and f is a permutation, then $y_1 = y = f(x)$ is random as well, and hence by property (2) we get that a is random, as is expected by A. Now B runs $x_1 \leftarrow A(x_0, a)$. By assumption, with probability ε we indeed have $g_a(f(x_1)) = g_a(f(x_0)) = g_a(y_0) = g_a(y_1) = z$. Since $x_0 \neq x_1$ and f is a permutation, we have that $y_0 = f(x_0) \neq f(x_1)$. But by property (1), g_a has only two preimages of z: namely, y_0 and y_1. Since $f(x_1)$ is also a preimage of z and is different from y_0, it must be equal to y_1. Thus, $f(x_1) = y_1 = y$, so $x_1 = x$ indeed. □

6.4 Comparing CRHF's and UOWHF's

From what we have seen, we can compare the pros and cons of using CRHF's vs. UOWHF's.

1. UOWHF's are equivalent to OWF's, while CRHF's probably form a stronger assumption of their own. Thus, basing something of UOWHF's is more general.

2. CRHF's give a more efficient hash-then-sign method, since the hash function does not have to be signed.

3. CRHF's are easier to compose than UOWHF's. Namely, the same h can be used all the time. While UOWHF's require the key to grow (only logarithmically though) with the length of the message hashed. Combined with the previous point, this gives even more preference to hash-then-sign using CRHF's.

4. Most number theoretic assumption that we use for constructing OWF's (like factoring or discrete log) actually suffice for provably secure CRHF's as well.

5. In practice, people use a single hash function (like MD5 or SHA-1) and hope "it is collision-resistant". Thus, in practice people anyway assume that what they are using is a "collision-resistant function".

To summarize, in theory the distinction is very important, while in practice assuming CRHF's is OK.