Last Revised: 10.3.96.

ECS 227 — Modern Cryptography — Fall 96
Lecture 3

Collision-Intractable Hashing

3 October 1996

Reading for Lecture 3:

[**FIPS-180**] National Institute of Standards, Federal Information Processing Standards Publication— FIPS PUB 180, "Secure Hash Standard." May 11, 1993. *Another peculiar function, this one derivative of Ron Rivest's MD4. SHA was revised to* SHA-1*, as described in the Federal Register, vol. 59, No. 131, July 11, 1994.*

3 A First Reduction

Today we're going to study a cryptographic primitive called a *collision-intractable hash function.* We're going to show how to achieve this object from a lower-level primitive called a *collision-intractable compression function.* This will be our first example of a reduction.

Suppose $f\colon A \to B$ is a function mapping a domain A to a range B. A *collision* for f is a pair of strings $x, y \in A$ such that $x \neq y$ but $f(x) = f(y)$.

Of course a one-to-one function has no collisions, while any function which is not one-to-one does have collisions. In particular, any function from a set A to a set B where $|A| > |B|$ has collisions (a fact sometimes referred to as the "pigeonhole principle"). **But:** a (strangely-constructed) function could have collisions, but no "human being" might ever be able to find one! Such a function would be "computationally one-to-one."

Informally, a function f is called *collision-intractable* if it is hard to find a collision. That is, it is hard to find a pair of points $x \neq y$ for which $f(x) = f(y)$.

What does "hard to find" mean? Roughly, that we can't design an efficient algorithm A which outputs a collision. That is, it requires an in-ordinate amount of time to find a collision. You can think of this, very roughly, as saying the problem of finding collisions is not in **P**. But this is not really right, not only because **P** is for decision questions, but also because collisions can be hardwired into the code of an algorithm, as will be explained in a moment.

Coming up with a proper notion is hard. In the Secure Hash Standard (SHS) we find:

> The SHA *(Secure Hash Algorithm)* is designed to have the following properties: [*Condition A*] it is computationally infeasible to find a message which corresponds to [*ie., hashes to*] a given message digest, or [*Condition B*] to find two different message which produce the same message digest.

(Italics my own.)

Now Condition A is very hard to make any sense of: what does it mean a *given* message digest? Suppose I choose a string x and compute $y = \mathrm{SHA}(x)$. Then isn't y a *given* message digest for which it is computationally *trivial* to find a message (ie., x) which hashes to it? The above remark isn't flippant: indeed this problem seems to be very hard to sidestep— it is really unclear how we can exclude in a definition an adversary's ability to construct messages in exactly this sort of way.

To this Chris Wee objects: "No! 'Given' means that that it is *given*—say *I* give you the point to invert." To which Phil responds: "But how, Chris, do *you* choose the point you're going to give me?! Do you mean that the function should be hard to invert for *any* distribution on points which you might give to me? But that is impossible to achieve. Do you mean that it is hard to invert for any *reasonable* distributions on points? But what would *reasonable* mean? Maybe for random points? Or for the image of uniformly-distributed domain points? I can choose some notion like this last one and make it into a definition, but by now I've really read an awful lot into what was written!

Condition B might seem clearer, but it too hides difficulties. When we say it is computationally infeasible to find two different messages, we're somehow asserting the inexistence of any "reasonable" algorithm. That is, we might try to formalize Condition B by saying something like

> [**Bogus Definition**] There is no computationally efficient algorithm that, on input of the description of SHA, finds a pair of distinct strings M and M' such that $\mathrm{SHA}(M) = \mathrm{SHA}(M')$.

But this makes no sense at all: *of course* there is a computationally efficient algorithm that finds such an M and M': the particular algorithm which *knows* such an M and M' and immediately outputs them.

We will take Condition B to be our motivating notion for a collision-intractable hash function but we won't pursue formally defining this for now.

What we want to do instead is to proceed with our intuitive notion and design a collision-intractable hash function $H\colon \{0,1\}^* \to \{0,1\}^L$ for some fixed length L, like $L = 160$. We call L the output length. This looks to be —and is— a very hard task.

To make things easier, what people do is first try to design something called a *collision-intractable compression function.* This is a function h: $\{0,1\}^L \times \{0,1\}^b \to \{0,1\}^L$ for some b called the *block length.* We want $h(K,x) = h(K\|x)$ to be collision-intractable: it is hard to find $(K_1, x_1) \neq (K_2, x_2)$ such that $h(K_1, x_1) = h(K_2, x_2)$.

In the SHS, the underlying compression function sha has parameters $L = 160$ and $b = 512$. Compression functions are almost always true primitives of the "confusion/diffusion" type; they scramble the input in strange ways. The most well-known compression functions are those underlying hash functions MD4, MD5, and SHA.

We are going to assume that we have a collision-intractable compression function h. We want to use this to design a collision-intractable hash function H which takes inputs of any length.

We like to think of a compression function as defining a family of functions indexed by the first argument. That is, let $h_K(x) = h(K,x)$. Now h_K: $\{0,1\}^b \to \{0,1\}^L$.

Try 1. Here's a thought. Let iv be a fixed, random l-bit string called the *initialization vector.* Break the input to H into b-bit blocks, $M = M_1 M_2 \ldots M_n$. (Assume for now that the input is some integral number of blocks.)

$$H_1(M_1 ... M_n) = h_{\mathrm{iv}}(M_1) \oplus \cdots \oplus h_{\mathrm{iv}}(M_n).$$

This indeed compresses the input. Is it a collision-intractable hash function, assuming that h is a collision-intractable compression function?

The answer is no, regardless of h. For example, you can permute the input blocks: $H_1(M_1 M_2) = H_1(M_2 M_1)$. That is, for any $M_1 M_2$ with $M_1 \neq M_2$, the strings $M_1 M_2$ and $M_2 M_1$ form a collision.

The is called an attack on the candidate function H_1 and has lead us to discard it. Notice an important element of this attack. It works even though h is collision-intractable. We had a strong property of h but we failed to exploit it wisely.

Try 2. Let's try to fix this. How about:

$$H_2(M_1 ... M_n) = h_{\mathrm{iv}+1}(M_1) \oplus \cdots \oplus h_{\mathrm{iv}+n}(M_n).$$

That is, each time add 1 to the iv. Now notice the above attacks fails.

$$\begin{aligned} H_2(M_1 M_2) &= h_{\mathrm{iv}+1}(M_1) \oplus h_{\mathrm{iv}+2}(M_2) \\ H_2(M_2 M_1) &= h_{\mathrm{iv}+1}(M_2) \oplus h_{\mathrm{iv}+2}(M_1) \,. \end{aligned}$$

There is no particular reason to think these will be the same.

Can you break this? We tried, and at first could not succeed. What's the next step? Shall we put this into our network security server?

This is the basic underlying problem of protocol design. In the first lecture we talked about protocol design having two approaches:

The iterative, or attack-based approach: If you find an attack on one candidate construction, come up with another try and try to attack it. If you fail you are done. Under this approach we might now adopt H_2.

A better approach is to put the burden of proof on the protocol designer rather than the attacker. The designer must justify the construction, not claim that it is justified by virtue of an *absence* of known attacks.

If we want to think more deeply about our contruction H_2 we would ask how its collision-intractablity relates to the assumed collision-intractability of h. After you think about this for a while the answer is: not clear. Apparently unrelateed. This should make us wary.

We can exploit the aspects which make the construction "not clearly correct" and design a collision-intractable compression function h for which the resulting hash function H_2 is *not* collision-intractable. In class we did this. I won't repeat the contruction here. Come up with your own.

This is the sense in which we do *attacks* in the reductionist-security tradition. We find a way to instantiate the lower-level primitive that achieves what we said that lower-level primitive was supposed to acieve, but for which the higher-level construct whic uses it does *not* behave as it is supposed to.

In this case one would conclude: *even if h is collision-intractable, the function H_2 need not be.*

But how can we do better? There are an infinite number of possible attacks. How can we ever be sure that all of them fail?

The provable security approach: Come up with another try H_3 and justify that it is collision-intractable using the assumption that h is collision intractable.

Notice what we are trying to do is exactly to show that *all* attacks fail. This raises the question of how we could possible do such a thing. We will see.

Such a construction was provided by Ralph Merkle a long time ago. It is one of the clasics of cryptographic protocol design— simple and easily seen to be correct. The construct is shown in Figure 1. (In class draw it also showing the intermediate variables K_i). For now we continue to assume that the message is some integral number of blocks, and that we are hashing message all of a single length.

The idea of the construction is to hash the first block and use the output as a "key" to the another iteration of the compression function. This is used to hash the next block, and so on. So:

$$\begin{aligned} K_0 &= \mathrm{iv} \\ K_1 &= h_{K_0}(M_1) \\ K_2 &= h_{K_1}(M_2) \end{aligned}$$

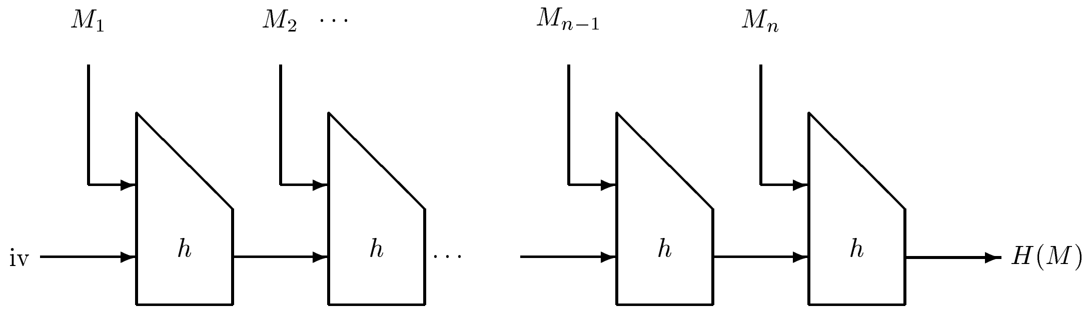

Figure 1: *The iterated construction of a hash function given a compression function h.*

$$\begin{aligned} \vdots \quad &= \quad \vdots \\ K_{n-1} &= h_{K_{n-2}}(M_{n-1}) \\ K_n &= h_{K_{n-1}}(M_n) \\ & \quad \text{Output } K_n \end{aligned}$$

Now our claim is this is collision-intractable.

Informal Theorem: *If h is collision-intractable then so is the hash function H defined from h via Merkle's construction.*

First notice that this is a strong statement because it precludes *all* attacks which don't find collision in h. As long as you have confidence in h you don't any more need to look for attacks on h. It's a waste of your time. People outside reductionist-cryptography rarely understand this.

For example, I claim permuting message blocks won't lead to a collision. Why? Because the theorem tells me there is no feasible way to find a collision without finding a collision in h.

The theorem statement makes no mention of any particular attack of algorithm. It doesn't talk of how an attacker might work. It doesn't say whether she permutes messages. Or uses linear algebra. Or probability. Or who-knows-what fancy math. Yet it covers any such possibilities.

This is an important point and needs to be well appreciated. In the rest of this course we will see many arguments about proven security. But the first important step is to understand what it says.

We want to prove this theorem. We argue by contradiction We will a ssume that H is not collision-intractable and show that, in this case, h is not collision-intractable either.

What does it mean that H is not collision-intractable? There must be an algorithm, call it A_H which finds collisions in H. Using A_H I will define a second algorithm, A_h which finds collisions in h.

Let $M^1 = M_1^1 \dots M_n^1$ and $M^2 = M_1^2 \dots M_n^2$ be *distinct* messages such that $H(M^1) = H(M^2)$. Let's look at what's happening inside the construction. Recall our goal is to output a collision in h.

Say $n = 3$. In class we drew a picture of this case, labeling the intermediate strings:

$$\begin{array}{lcl|lcl} K_0 & = & \text{iv} & K_0 & = & \text{iv} \\ K_1^1 & = & h_{K_0}(M_1^1) & K_1^2 & = & h_{K_0}(M_1^2) \\ K_2^1 & = & h_{K_1^1}(M_2^1) & K_2^2 & = & h_{K_1^2}(M_2^2) \\ K_3^1 & = & h_{K_2^1}(M_3^1) & K_3^2 & = & h_{K_2^2}(M_3^2) \end{array}$$

But $K_3^1 = K_3^2$. This is a collision in the compression function on inputs (K_2^1, M_2^1) and (K_2^2, M_2^2). Well, is it? Just one problem: these two pairs of inputs could be the same, in which case it is not a collision. But, if so, we have $K_2^1 = K_2^2$ and also $M_1^1 M_2^1 \neq M_1^2 M_2^2$. (For the latter we are using the assumption that the original messages are distinct and the last blocks are the same.) So we are set to continue. The algorithm A_h, using nothing but M and M' —the outputs of A_H— soon finds the collision in h. We know that A_h is able to find a collision eventually, because we know the original messages are distinct: we can't keep getting equal message blocks. When we find it, we output it.

Summary of Approach and Lessons: The goal was to construct a function H having some desired cryptographic property.

- There is a *notion*, or *definition*, of the security that we want. In this case, collision-intractable. Today we were pretty informal here. In the future we won't be.
- There is an *underlying primitive* — in this case, the compression function.
- There is an *assumption* about this primitive — namely that it is collision-intractable. What we do is *based* on this primitive and the assumption about it.
- There is a *construction* or *scheme* — namely Merkle's scheme which defines the primitive we are trying to build.
- There is a *theorem* saying that our construction is secure as long as the primitive is secure. That is, there is a *reduction* of the security of the construction to the security of the underlying primitive.
- The theorem has a proof, and this proof is done *by contradiction.* We assume there is an attack on our construction and use this to show there is an attack on the underlying primitive. This is pretty much the *only way* to prove reductionist theorems on security. Always think this way.

When you have to design some scheme to solve some problem, think through these steps and follow this approach.

Now I want to talk of *quantitative security.* We have not been quantitative above. We have said it is hard to find collisions in H if it is hard to find collisions in h. But how hard? It could be that what we are saying is that if it is very very very hard to find collisions in h, then it is moderately hard to find collisions in H. This wouldn't be such a strong claim. We need to measure more accurately the resources used.

Theorem: Suppose there is an algorithm A_H which uses time t to find a collision for H, and this collision consists of a pair of n-block messages. Then there is an algorithm A_h which uses time $t + 2n \cdot (T_h + b + L) + O(1)$ to find a collision in h, where b is the block length of h, L is the key/output length of h, and T_h is the running time of h.

Reductions can be classified according to how good they are. Better reductions are those that result in the attack on the underlying primitive taking time about that of the given attacker on the constructed primitive.