Cryptographic Hash Functions

Sriram Venkatesh

August 19, 2024

Abstract

This paper discusses applications and standard definitions of cryptographic hash functions. It also proves the security of a well-known construction for cryptographic hash functions, and discusses extensions to the Random Oracle Model.

1 Introduction and Applications

A hash function is a mathematical function that can compress large messages into smaller digests. A cryptographic hash function is a specific kind of hash function that is computationally infeasible to invert. Hash functions are useful in ensuring the integrity of data sent through public channels.

Cryptographic hash functions are useful in creating digital signatures for messages. Suppose Alice wants to send a message m to Bob. She would like to sign the message digitally so that Bob knows that the message really came from Alice. Alice and Bob each have private and public keys, which are inverses of each other. Alice will first apply the hash function to her message to get $h(m)$. She will then sign $h(m)$ using her private key, and send both the message and the signature to Bob. Bob can then apply Alice's public key, then compare this to the hashed version of the message sent. If the two messages are equal, then Bob can be sure that the message was really sent by Alice.

2 Requirements for Cryptographic Hash Functions

Definition 2.1. A hash family is a 4-element tuple: $\{X, Y, K, H\}$, where

1. X is a possibly infinite set of possible messages
2. Y is the finite set of possible outputs, or digests.
3. K is the finite set of keys.
4. For $k \in K, \exists h_k \in H$, where $h_k : X \to Y$.

Given a hash family, we consider every element of H to be a hash function. In this paper, we will only consider non-keyed hash families, where $|K| = 1$ and $|H| = 1$.

Definition 2.2. A cryptographic hash function h is a hash function for which there exist no polynomial time algorithms in the message size that can compute the following:

1. Preimage: Given $y \in Y$, find $x \in X$ such that $h(x) = y$.
2. Second-preimage: Given $x \in X$, find $x' \in X$ such that $x' \neq x$ and $h(x) = h(x')$.
3. Collision: Find $x, x' \in X$ such that $h(x) = h(x')$.

Typically, $|X| > |Y|$, so there will almost always be a collision for a given hash function.

While there do not exist deterministic polynomial time algorithms for solving any of these problems, we can come up with probabilistic Las Vegas algorithms. A Las Vegas algorithm is a probabilistic algorithm that always outputs the correct result. If it is not able to compute the answer, it will inform about its failure.

To solve each of these problems, we must choose a polynomial-size search space. The Las Vegas algorithms for the three problems can be modeled as a pair (Q, ε), where Q is the search space and ε represents the probability of success.

Let $M = |X|$ and Q be the search space. We can easily show there exists a Las Vegas algorithm $(Q, 1 - (1 - \frac{1}{M})^Q)$ to solve Preimage. Such an algorithm simply scans the entire input space to invert a specific output of the hash function. Similarly, we can show that an algorithm $(Q, 1 - (1 - \frac{1}{M})^{Q-1})$ can solve Second-preimage. The collision problem is slightly more interesting.

The algorithm we will use to solve the collision problem is simple. We pick a search space X_0, and compute $h(x)$ for all $x \in X_0$. The probability of success is the probability that there exist two equal values of $h(x)$. We can show that the probability of success given $|X_0| = Q$ is

$$1 - \left(1 - \frac{1}{M}\right)\left(1 - \frac{2}{M}\right)\ldots\left(1 - \frac{Q-1}{M}\right).$$

Solving the collision problem is very similar to the Birthday Paradox, which asks for the number of people necessary for it to be more likely than not that two people share a birthday. It can be shown that we need $\sqrt{2n\log(2)}$ people before we expect a collision of two birthdays. Like birthdays, hash values are also random, so we would need a search space of $O(\sqrt{M})$ before we are likely to have a collision. Therefore, we can write our Las Vegas algorithm for the collision problem as $(\sqrt{M}, \frac{1}{2})$.

This, of course, means that for an effective hashing algorithm, M should be large enough so that $\sqrt{M}$ computations are not feasible. Even 40-bit hashing protocols, where there are 2^{40} possible hash values, are rendered obsolete with today's computational power.

3 Reductions

Definition 3.1. An oracle is a theoretical black-box that can solve a certain problem in a single operation.

We will show that if there exists an oracle machine that can solve Preimage or Second-preimage, then there exists a polynomial time algorithm to solve the collision problem. This relationship between two problems is called a polynomial time reduction.

It is fairly obvious that any oracle machine that can solve Second-preimage can also solve Collision. However, it is not so obvious for Preimage. To see this, we will prove the following theorem.

Theorem 3.2. *Suppose $h : X \to Y$ is a hash function such that $|X| \geq 2|Y|$. Suppose there exists an Oracle machine $(1, Q)$ that can solve Preimage. Then, there exists a Las Vegas algorithm $(\frac{1}{2}, Q + 1)$ to solve Collision.*

Before we begin the proof let us address the assumption we made initially. Assuming that $|X| \geq 2|Y|$ is valid because hash functions are used to compress bitstrings of arbitrary size. Therefore, $|X| \gg |Y|$ in general.

Proof. We first partition X into groups so that all elements in a group have the same hash value. We can see that there will be $C = |Y|$ such groups. Let's say we pick a value $x \in X$ and want to find another value $x_0 \in X$ such that $h(x) = h(x_0)$. This is essentially solving the second preimage problem, but it will be useful to us in our proof.

We can compute $z = h(x)$ and apply the Oracle machine to z to get some value $x_0 \in X$ such that $h(x_0) = z = h(x)$. The only time this will fail is if $x_0 = x$, and we would not have found a valid collision. Let $s(x)$ be the size of the group that x is in. The probability that the Oracle machine will not output the original value of x is $\frac{s(x)-1}{s(x)}$.

We now compute the average case success probability of this algorithm over all possible values of x. This average probability is

$$\frac{1}{|X|} \sum_{x \in X} \frac{s(x) - 1}{s(x)}.$$

We can break this summation into two, where we iterate over the groups first and then the values of x within each group. Given that C is the set of all the groups, we have

$$\frac{1}{|X|} \sum_{c \in C} \sum_{x \in c} \frac{s(x) - 1}{s(x)} = \frac{1}{|X|} \sum_{c \in C} \sum_{x \in c} \frac{|C| - 1}{|C|}.$$

The size of a group is constant for all $x \in c$, so we can write

$$\begin{aligned} \frac{1}{|X|} \sum_{c \in C} |C| \frac{|C| - 1}{|C|} &= \frac{1}{|X|} \sum_{c \in C} |C| - 1 \\ &= \frac{1}{|X|} \sum_{c \in C} |C| - \frac{1}{|X|} \sum_{c \in C} 1. \end{aligned}$$

We know that $\sum_{c \in C} |C| = |X|$ and $\sum_{c \in C} 1 = |Y|$, so we can simplify the above equation to

$$1 - \frac{|Y|}{|X|}.$$

Since we initially made the assumption that $|X| \geq 2|Y|$, the average case probability is at least $\frac{1}{2}$. This, we have shown that there exists a Las Vegas algorithm modeled as $(Q+1, \frac{1}{2})$ that can solve Collision. The reason we have $Q + 1$ instead of Q is because once we pick a

random $x \in X$, we must compute $h(x)$ before applying the Oracle machine that can solve Preimage, so this one extra computation causes the $Q + 1$ to appear.

■

From these reductions, we can see that if we can solve Preimage or Second-preimage efficiently, then we can solve Collision efficiently. Therefore, a hash function that is collision resistant is also resistant to Preimage and Second-preimage. We see that solving Preimage or Second-preimage is much harder than solving Collision, so attackers typically focus break the Collision problem.

Definition 3.3. Ideal cryptographic hash functions: Given any $X_0 \subseteq X$, we compute Y_0 where Y_0 is the hash operation applied to each element of X_0. If the knowledge of Y_0 does not make computing $h(x')$ easier for $x' \in X$ and $x' \notin X_0$, then h is an ideal cryptographic hash function.

4 Merkle–Damgård Construction

We will now discuss a construction that is used as a primary step in sophisticated cryptographic hash algorithms such as MD5, SHA-1, and SHA-2.

Definition 4.1. A compression function can transform an input of a fixed length to an output of a fixed length.

On the other hand, a hash function can compress an input of arbitrary length to an output of fixed length. The Merkle–Damgård Construction provides a way of building collision resistant hash functions given a collision resistant compression function.

Let us say that we have a compression function, $c(x)$ that can compress bitstrings of size $m + t$ to size m, for any m. We wish to hash a bitstring x of arbitrary length. For this algorithm we denote $||$ to mean concatenation. The construction is performed with the following algorithm. The original proposition of this algorithm can be found in [2]

1. We first partition x into chunks of size $t - 1$. We form $k = \lceil \frac{|x|}{t-1} \rceil$ groups, and label them $x_1, x_2, \ldots x_k$. If $|x|$ is not a multiple of $t - 1$, then $|x_k| < t - 1$.

2. We create a new string y such that y is $y_i = x_i$. However, y_k is padded at the end with $d = k(t - 1) - |x|$ zeroes, which is the amount by which x_k is less than all of the other partitions.

3. We apply an additional step called the Merkle–Damgård strengthening step, which pads y with the binary encoding of d. We represent this encoding as y_{k+1}. This step ensures that the transformation from x to y is injective, which we will see later is very useful for us.

4. We assign z_1 to be y_1 padded in the front with $m + 1$ zeroes so that it is of size $m + t$. Let $g_1 = c(z_1)$.

5. We loop i from 1 to k, and each time we assign z_{i+1} to be $g_i||1||y_{i+1}$, where $||$ indicates concatenation. We assign $g_{i+1} = c(z_{i+1})$.

6. We finally ouptut $h(x) = g_{k+1}$.

Theorem 4.2. *Given that c is a collision-resistant compression function, then the cryptographic hash function generated by the Merkle–Damgård Construction is also collision-resistant.*

Proof. We will prove that if the hash function is not collision-resistant, then the compression function is also not collision-resistant, which is equivalent to proving the original statement.

Assume we have two bitstrings x, x', and hash function h such that $x \neq x'$ and $h(x) = h(x')$, which is a collision. We construct strings y and g from x, and strings y' and g' from x'. Let $k = \lceil \frac{|x|}{t-1} \rceil$ and $l = \lceil \frac{|x'|}{t-1} \rceil$.

To show that there must be a collision in the compression function, we will consider different cases.

Case 1: $|x| \not\equiv |x'| \pmod{t-1}$

Since $h(x) = h(x')$, we know that $g_{k+1} = g'_{l+1}$. We know that $g_{k+1} = c(g_k||1||y_{k+1})$ and $g_{l+1} = c(g'_l||1||y_{l+1})$. Therefore, we have,

$$c(g_k||1||y_{k+1}) = c(g'_l||1||y_{l+1}).$$

However, we know that $y_{k+1} \neq y_{l+1}$ because each of these represent the binary encoding of the remainder $k(t-1) - |x|$ and $l(t-1) - |x'|$ Therefore, the two strings within the compress function are not the same and we have found a collision.

Case 2: $|x| \equiv |x'| \pmod{t-1}$

We first consider the case where $|x| = |x'|$. This means that $k = l$, and $g_{k+1} = g'_{k=1}$, which means that

$$c(g_k||1||y_{k+1}) = c(g'_k||1||y'_{k+1}).$$

We know that $y_{k+1} = y'_{k+1}$ because the two string lengths are congruent modulo $t-1$. If $g_k \neq g'_k$, then we have found a collision. Otherwise, we let $g_k = g'_k$, from which we can write

$$c(g_{k-1}||1||y_k) = c(g'_{k-1}||1||y'_k).$$

If the two strings inside the compress function are not equal, we have again found a collision. If they are, this means that $g_{k-1} = g'_{k-1}$ and $y_k = y'_k$. We can continue this relationship on and finally write that $y_1 = y'_1$. However, this means that $y = y'$, and since the transformation from x to y was injective, this implies that $x = x'$. However, this is a contradiction because we initially assumed that $x \neq x'$. Therefore, at some point, either $g_i = g'_i$ or $y_i = y'_i$, which means that we have found a collision.

We finally consider the case where $|x| \neq |x'|$ but $|x| \equiv |x'| \pmod{t-1}$. Assuming that x was split into k groups and x' was split into l groups, we again have that $g_{k+1} = g'_{l+1}$. Without loss of generality, we assume that $l > k$..

We can follow the same procedure as the previous case and continue writing down equalities among the compressed values. Assuming we have not found a collision, the final equality will be $g_1 = g'_{l-k+1}$, which means that the equalities among the compressed values will be

$$c(0^{m+1}||y_1) = c(g'_{l-k}||1||y'_{l-k+1}).$$

However, the values within c cannot be equal on both sides. We know that the size of g'_{l-k} is m. The $m+1$-th bit of $0^{m+1}||y_1$ is a 0. However, the $m+1$-th bit of $g'_{l-k}||1||y'_{l-k+1}$ is a 1. Here, we see the importance of concatenating a 1 instead of 0 in step 5. Therefore, we have found two values that compress to the same output and we have found a collision. ■

5 Random Oracle Model

Definition 5.1. Random Oracle: A random oracle is a theoretical black box that responds to any query with a truly random output that is not influenced by past queries.

We refer to hash functions follow the random oracle model as ideal cryptographic hash functions. In such functions, knowledge of previous hashed values will not give any additional clue for hashing a new value. In all the proofs above, we have assumed that our cryptographic hash functions are essentially random oracles. Any system that is proven secure on ideal hash functions are said to be secure in the random oracle model. Proving the same without this assumption is often difficult, so most proofs relating to hash function are done in the random oracle model. Certain systems have been proven secure in the standard model of cryptography (a model without the random oracle assumption)but there are others that still require such proofs. For more information on extending the Merkle–Damgård construction to the practical hash functions, read [1].

References

[1] M. Backes, G. Barthe, M. Berg, B. Grégoire, C. Kunz, M. Skoruppa, and S. Z. Béguelin. Verified security of merkle-damgård. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 354–368, 2012.

[2] I. B. Damgård. A design principle for hash functions. In G. Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 416–427, New York, NY, 1990. Springer New York.