Definitions and Results for Authentication Systems

Ivan Damgård, Cryptography Course, 2005

Version 4

1 Introduction

In this note, we look at systems for authentication of data, that is, how do we protect data from unwanted modification? Clearly, this is of utmost importance in E-commerce, home banking, database security, etc. If we could physically prevent unauthorized access to the data, this would of course solve the problem, but in most cases this is infeasible or even impossible. So if our data must be in an environment where we do not have physical control, of course we cannot *prevent* that the data is modified, but we can make sure that it cannot happen without us noticing that something is wrong. In other words, an adversary cannot make an honest user believe that a piece of data is authentic when it is not.

Usually, we model the scenario by having a sender that communicates data to a receiver over some channel, where an adversary can modify the communication any way he likes. Loosely speaking, the adversary wins if he can make the receiver accept a message that was not sent by the sender. Note that this also covers the case where a user stores data in a database and later, when retrieving the data, wants to make sure that it was not tampered with. In this case, sender and receiver is the same person, and we can think of the database as playing the role of communication channel.

All systems we know of for this purpose work by computing, from the data (say a message m) and possibly a secret key, some *authenticator value* s. The idea is that later, the message can be checked against the authenticator, and we hope that we will see a mismatch if the data or the authenticator was changed. The way the systems work on a more technical level then varies, depending on whether the adversary is able to tamper with both the message and the authenticator, or only the message; and also depending on whether sender and receiver share in advance a secret key or not. In slightly more detail:

Adversary can only change the message: a typical application would be a user that computes an authenticator s for the data m. He stores m in some insecure location, but brings the authenticator with him in secure storage (say, on a chipcard). Later, he wants to check that the data has not been changed. Such systems can be built from *cryptographic hash functions*.

Adversary can change both message and authenticator: This will be the case when a sender communicates with a receiver over an insecure channel. The are two subcases:

- **Sender and receiver share a secret key:** In this case, authenticator s is computed from both data m and the secret key k. m, s is sent on the channel, and m', s' is received. Then the receiver can, using the *same* key k, test if m', s' look OK. Of course, this should always be the case if $m', s' = m, s$,and we hope that anyone who does not know k cannot come up with a fake pair m', s' that the receiver would accept. Such systems can be built using *Message Authentication Codes* (MAC's) also called *Keyed Hash Functions*.

- **Sender and receiver share no secret keys in advance** In this case, we have a scenario similar to that of public-key encryption: the sender has a secret key sk, and has published a public key pk. Now, the sender computes authenticator s from data m and secret key sk. m, s is sent on the channel, and m', s' is received. Now the receiver takes the public key pk of the user he believes sent the message and tests, using pk, if m', s' look OK. Such systems can be built using *Digital Signature Schemes.* The name comes from the fact that the sender is the only one who can produce valid authenticators using sk, since he is the only who knows sk – but using the public key pk, anyone can verify them. Ordinary handwritten signatures also (supposedly) has this type of property:I'm the only one who can produce my signature, but it can be verified by anyone.

In the following, we give some definitions and general results for the different types of schemes.

2 Hash Functions

A hash function is defined by a *generator* $\mathcal{H}$, which on input a security parameter k outputs the description of a function $h : \{0,1\}^* \rightarrow \{0,1\}^k$, i.e., a function mapping any bit string to a string of length k. By "the description" we simply mean information allowing us to compute the function efficiently (formally speaking, in polynomial time in k and length of the input string). Note that there is no secret key involved here - hence in applications of this concept, we assume that everyone (including the adversary) knows the function we are using.

One application of this is for the case where a data string m is to be stored in an insecure location. We then compute the authenticator $h(m)$ and store it in a secure location. Later, we retrieve a (possibly different) string m', we then compute $h(m')$ and compare to $h(m)$. This application motivates why we want the output to have fixed length: it may be feasible to store a short fixed size value securely, while several megabytes are in most cases infeasible to handle. The value $h(m)$ is also sometimes called a *message digest* or a *message fingerprint*.

What properties do we need for such a function in order for the system to be secure? Clearly, an adversary sees the function h and the message m we stored. He would be in business, if he could find a different message m' such that $h(m) = h(m')$. This is called a *second preimage attack*. But it might also be relevant for the adversary if he could find *any* pair of messages $m \neq m'$, such that $h(m) = h(m')$ – a so-called collision. Now, if he can trick the honest user into storing m, he can later change this to m' without being detected. This is called a *collision attack*.

Clearly, if you can do a second preimage attack, you can also do a collison attack, using the same amount of time and with the same success probability: just choose m yourself and then run whatever algorithm you have for a second preimage attack. Therefore, the best security is obtained, if even a collision attack is infeasible, so this is what one usually uses to define security of a hash function:

DEFINITION 1 *Consider the game where we run $\mathcal{H}$ on input k to get function h. We give h as input to adversary algorithm A, who outputs 2 strings m, m'.*

We say that A has success if $m \neq m'$ and $h(m) = h(m')$. We say that $\mathcal{H}$ is $(t(k), \epsilon(k))$-secure, if any adversary running in time at most $t(k)$ has success with probability at most $\epsilon(k)$.

From an asymptotic point of view, we say that $\mathcal{H}$ is *collision-resistant*, if no matter which polynomial we put for $t(k)$, the best $\epsilon(k)$ that can be achieved is negligible in k. What this means in practice is that already for moderate values of k, say less than 1000, we can have $(t(k), \epsilon(k))$-security for large values of $t(k)$ and small values of $\epsilon(k)$, say $t(k) \approx 2^{50}$ evalutations of h and $\epsilon(k) \approx 2^{-60}$.

Collision-resistant hash functions exist under well-known intractability assumptions: suppose we define $\mathcal{H}$ as follows: choose a prime $p = 2q + 1$, where q is a $k-1$-bit prime. Choose α, β of order q in Z_p^*. Now define a function $h : Z_q \times Z_q \to Z_p^*$ by $h(m_1, m_2) = \alpha^{m_1}\beta^{m_2} \bmod p$. Now, if one could find a collision for h, this would give $(m_1, m_2) \neq (m'_1, m'_2)$ with $\alpha^{m_1}\beta^{m_2} = \alpha^{m'_1}\beta^{m'_2} \bmod p$. We must have either that $m_1 \neq m'_1$ or that $m_2 \neq m'_2$. Suppose without loss of generality that we have the first case, which means that $(m'_1 - m_1)^{-1} \bmod q$ exists. Then by rearranging the equation, we get $\alpha = \beta^{(m_2 - m'_2)(m'_1 - m_1)^{-1} \bmod q} \bmod p$, so we have now found the discrete logarithm of α base β. Hence if discrete log is hard, then it is hard to find collisions for h. So by choosing large enough k, we can make functions where finding collisions is infeasible in practice.

Of course, this construction does not quite fit the definition because we cannot handle any input length. Fortunately, there is a general result that solves this problem – it is enough if we can compress the input, if only by a single bit. In other words, it is always enough to be able to make collision-resistant functions that map a fixed number of bits, say m bits to k bits, as long as $m > k$:

Theorem 1 *If there exists a hash function generator $\mathcal{H}'$ producing funtions with finite input length $m > k$ and $t(k), \epsilon(k)$ security, then there exists a generator $\mathcal{H}$ that produces functions taking arbitrary length inputs and has $(O(t(k)), \epsilon(k))$ security.*

Details of the construction behind this result can be found in Stinson, sec. 4.3.

Also the RSA assumption is sufficient to make collision-resistant functions:

Exercise 1: Given an RSA modulus $n = pq$, where $p = 2p' + 1, q = 2q' + 1$ and p', q' are also primes. Show that there exists elements in Z_n^* of order $2p'q'$. Let g be such an element, and define a function $h : \{0,1\}^* \rightarrow Z_n^*$ by $h(m) = g^m \bmod n$. Show that, given a collision for h, one easily factor n. Hint: recall that we already know that given any multiple of $\phi(n)$, we can easily factor.

How does the qustion of existence of collision intractable hash functions relate to the other big theoretical questions in cryptography? it is very easy to see that if it is easy to invert a hash function h, then it is also easy to find collisions:

LEMMA 1 *Given function $h : \{0,1\}^{k+1} \rightarrow \{0,1\}^k$, and assume we are given an algorithm A running in time t that, when given $h(m)$ for uniform m, returns a preimage of $h(m)$ with probability ϵ. Then a collision for h can be found in time t plus one evaluation of h and with probability at least $\epsilon/4$.*

PROOF. Suppose for simplicity that h is surjective. For every $y \in \{0,1\}^k$, choose some $x_y \in \{0,1\}^{k+1}$ with $h(x_y) = y$. Let B be the set of all such strings – there are exactly 2^k of them. Then we find a collision for h as follows: choose randomly $x_0 \in \{0,1\}^{k+1}$, compute $y = h(x_0)$ and give h, y as input to A, let x be A's output, and return x_0, x. Note that with probability $1/2$, $x_0 \notin B$, in which case the preimages of y are x_0, x_y and perhaps more elements. Since A only sees y, it has no information on which of the preimages we have chosen as x_0, so, given that A finds a preimage, the probability that it is different from x_0 is at least $1/2$. If h is not surjective, we replace $\{0,1\}^k$ by the smaller set $Im(h)$, which only makes the probability of success larger. $\triangle$

This immediately implies that if collision-resistant hash functions exist, then one-way functions exist. But the converse implication is not known to be true, and also no connection in either direction is known between existence of collision-resistant hash functions and one-way (trapdoor) permutations.

2.1 Hash Functions in Practice

How large should we choose k in practice? There is one general lower bound that comes from an attack one could try against *any* hash function: simply choose a set of random inputs, evaluate the function on these and hope that we happen to run into a collision. If we choose the inputs at random of sufficiently large length, it is a reasonable assumption that the outputs would then be uniformly chosen in $\{0,1\}^k$. So we want to know how many random values we must choose in $\{0,1\}^k$ before there is a significant probability that at least two of the are the same. This is the well known "birthday paradox" question: if the number of values is the square root of the total number of values, then there is a constant probability of collision – as we go below the square root, the probability drops very quickly towards 0. In other words:

THEOREM 2 *Given function h, and assume that it is possible to sample input values to h causing the outputs to be uniformly random in $\{0,1\}^k$. Then a collision for h can be found with constant probability in time corresponding to $2^{k/2}$ evaluations of h.*

With current state of the art, this implies that k should be 128 at the very least, but 160 is preferable.

The contructions we have seen based on RSA and Discrete log are nice from a theoretical point of view, but not efficient enough in practice, in particular because we also want to use hash functions together with digital signature schemes, as we shall see. There we will need hash functions that are much faster to evaluate than public-key like operations. Therefore people have put much effort into designing hash functions that would be efficient both in hard- and software implementations. The best known examples are MD5 (Message Digest nr. 5, 128 bitsoutput) and SHA-1 (Secure Hash Algorithm, 160 bits output). MD5 was broken, however, in the fall of 2004, and was considered to be of shaky security status even before that, so the US standard SHA-1 is by far the most used. Also for SHA-1, however, attacks have been found in the fall of 2005 that run somewhat faster than brute force. Fortunately, SHA-1 exists these days in 256 and 512 bit variants, and one can expect that application will gradually move towards these more secure versions. All these functions were designed specially with 32 bit architecture in mind and can process several Mbits pr. second on a decent machine. The details of the SHA-1 design can be found in Stinson.

The reader might complain here that a function like SHA does not really fit our definition: SHA is just one given function, and not a generator we can run to make a function with any desired output size. But remember that a function like SHA does not just come falling from the sky, it is the result of a long design process. In fact, it makes good sense to think of this design process as being the generator and SHA as the function produced as output from the generator.

2.2 The random oracle model

Suppose I give you hash function h and input/output values m, $h(m)$. Suppose m' is equal to m, but with 1 bit flipped. Can we say anything about what $h(m')$ will look like? for instance, will it resemble $h(m)$ in any obvious way? if h is secure, and so is a complex and hard to invert function, the answer seems to be that we cannot say very much in general. We can evaluate h on m', and the value coming out will probably "seem like" a random value with no obvious relation to $h(m)$.

In the Random Oracle Model, we try to formalize this intuitive idea that the output values "might as well" have been random. The idea is that we take whatever application of the hash function we have in mind and replace the hash function by a *random oracle*. A random oracle is an oracle that will receive any bit string m as input and will return $R(m)$, where $R(m)$ is a randomly chosen k-bit string. If it later receives m as input, the same string $R(m)$ will be returned. But every time a new string m' is given as input, a fresh random string $R(m')$ is returned, chosen independently from anything else. In the random oracle model, we asume that *everyone*, including the adversary, has access to such an oracle, and it is the *same* oracle for everyone, i.e., whoever sends input m will receive back the same string $R(m)$.

Of course, no such oracle could be available in a real-life scenario. In real life, we have a concrete, given function h which is of course not random. What we can hope for (but not prove) is that an adversary would not be able to do better with the concrete hash function than he could with a random oracle. Now, in the random oracle model, any adversary would have to only use attacks that exploit no particular properties of the hash function. Consider for instance the application where we store m in an insecure location and later check it against $h(m)$. In the random oracle model this becomes a random value $R(m)$, and the adversary still has to find a collision to attack the system. Thus all he can do is to send a set of inputs to the oracle and hope that some of the answers happen to be the same. In other words, he has to use the generic attack from Theorem 2.

Summarizing, if an application of a hash function has been proved secure in the random oracle model, what this means for the real life application is that any attack that considers the hash function h as a "random black-box" and does not use any special properties of h, is doomed to failure. But there is no guarantee for attacks directed specifically against h. Of course, there would be no reason to evn consider this model, if we were able to go and prove our constructions secure in the standard way. But unfortunately, there are many constructions that are important in practice, where no security proof is known, but where we are able to say something in the random oracle model. This is certainly not what we would like best, but it is much better then having no proof at all.

3 Definitions for MAC's and Digital Signatures

An *authentication system* $\Sigma = (G, A, V)$ is defined, first by a probabilistic key generation algorithm G which outputs a pair of keys k_v, k_a. Algorithm A gets input a message m and the key k_a and produces an authenticator $A_{k_a}(m)$. Finally algorithm V gets as input an authenticator s, a message m and public key k_v, and outputs $V_{k_v}(s, m)$ which is equal to *accept* or *reject*. It is required that we always have $V_{k_v}(A_{k_a}(m), m) = accept$.

In *conventional* systems, we always have $k_a = k_v$. Here, G takes no input, it simply chooses the key uniformly from some fixed set $\mathcal{K}$. Also, there is one fixed set of $\mathcal{P}$ messages and one fixed set $\mathcal{A}$ of authenticators. Such systems are also called message authentication codes (MACs).

In *digital signature* systems, the keys are different, and in fact the key for verification can be made public without compromising the key for authentication. Here, G takes a security parameter k as input. This parameter measures the amount of security we are after: the larger k is, the more secure the keys are (hopefully). In addition, all algorithms should of course be efficient (polynomial time in k). Finally, G outputs, in addition to the keys (k_a, k_v), descriptions of the sets $\mathcal{P}$ of messages and the set $\mathcal{A}$ of authenticators that can be used with this particular key pair. In the following we will use write pk for k_v and sk for k_a to emphasize which key is public and which is secret. An example would be plain RSA signatures, where the key generation is the same as in the RSA cryptosystem, so $sk = (n, e), pk = (n, d)$ and $\mathcal{P} = \mathcal{A} = Z_n$. Here, k is usually the bit length of the modulus n. The signature on message m is $s = m^d \bmod n$, and to verify s, one checks that $s^e \bmod n = m$.

Note a very important difference between MAC's and digital signatures: if use MAC's, then since both parties know the secret key, there is no way a third party can later tell if a given authenticated piece of data was created by the sender or the receiver. Thus, such a system cannot be used to resolve, for instance, a conflict between a bank and a bank customer over the authenticity of a payment order. A digitial signature system can do this, provided the implementation ensures that only the user could have accessed his own secret key.

3.1 Security of Conventional Systems (MAC's)

To define the security of a conventional authentication system, we use a similar approach as for cryptosystems: Given some system (G, A, V), the enemy E gets access to an oracle, which knows a key k_a chosen by algorithm G. We now play the following game: E may, as many times as it wants, send some message m to the oracle, who will return to E the MAC $A_{k_a}(m)$. We say that E gets to do a chosen message attack. At the end of the game, E outputs a message m_0 and an authenticator a_0. Now Adv_E, the advantage of E is defined to be the probability that m_0 is not one of the messages the oracle was asked to authenticate, and that $V_{k_a}(a_0, m_0) = accept$.

We say that the system is (t, q, ϵ)-secure if any adversary that runs in time at most t and asks at most q quiries, has advantage at most ϵ.

3.2 Security of Signature Schemes

Security of a signature scheme is defined is almost the same way as for a conventional authentication scheme, but with some technical differences:

Before the oracle game with the adversary, we run G on input k to get sk, pk. We give pk to the adversary E, and keep sk inside the oracle. The adversary can now play the same game as described above, and is sucessful if he produces m_0, s_0, such that m_0 is not one of the messages the oracle was asked to authenticate (sign), and that $V_{pk}(s_0, m_0) = accept$. In this case, however, the probability that E has success is a function of the security parameter k, we call this $Adv_E(k)$

We say that a signature system is *secure* if for any polynomial time adversary E, $Adv_E(k)$ is negligible as a function of k. This is the strongest sense in which a signature system can be secure.

4 Existence of good MAC Schemes

One of the the most well-known scheme for MAC's is called CBC-MAC and is based on conventional cryptosystems: given such a system, say DES or AES, one simply encrypts the input message in CBC mode (using $IV = 0$ always) and define the MAC to be the final block of the ciphertext. This has the added advantage that the MAC has fixed length, no matter how long the message is.

If the cipher we start from is a good deterministic cryptosystem, i.e., it forms a good family of pseudorandom functions, the the resulting MAC we get this way is also good to a certain extent, as stated by the following result, from [1]. Notation: the block size of the given system is k bits, and the system is used on message of length n blocks.

Theorem

Suppose (G, E, D) is a (t', q', ϵ')-secure deterministic cryptosystem. Then, for any fixed n, CBC-MAC based on this system is a (t, q, ϵ)-secure MAC scheme, where

$$t = t' - O(nq'k), \quad q = q'/n - 1, \quad \epsilon = \epsilon' + \frac{3q'^2 + 1}{2^k}$$

The main conclusion from this result is similar to the one for modes of operation in encryption, namely that as long as we do not use the system for too large amounts of data before changing the key, the CBC-MAC inherits most of the strength of the original cipher.

It is not known if this result is tight, i.e. if there are attacks against CBC-MAC that show that the result above cannot be improved. But there are known attacks that show we cannot expect something better than $\epsilon' + \frac{q'^2}{n2^k}$, a factor of n off from the result we have.

Of course, in practice, we would like a MAC that works on messages of any length. To handle this, one may first use a padding scheme to make sure that all messages have length a multiple of k bits. For instance, allways append a 1, and then append enough 0's to make the string length a multiple of k. But even after doing this, how to we handle messages consisting of a variable number of blocks (the above theorem is only for a fixed number of blocks)? A straightforward method is to append the length in blocks of the message, encoded in binary, as an extra block. And then finally use CBC-MAC on the result. However, this does not work! Take any blocks b, b', c with $b \neq b'$. And ask the oracle for MAC's on messages b, b' and $b||1||c$, where $||$ means concatenation and 1 is the block containing just one 1-bit. Say this results in MAC's $t_b, t_{b'}, t_{b1c}$. One can then verify that t_{b1c} is also the MAC on the message $b'||1||t_b \oplus t_{b'} \oplus c$. Fortunately, there are other ways that do work: you may *pre*pend the length in blocks, instead of appending. Or you take the length n in blocks, encrypt n under your key, and use the result of this as a key when computing a MAC on the message itself.

Another construction of MAC-schemes is known as HMAC and can be based on any collision intractable hash function. We describe it here in the concrete standardized version that is based on SHA-1: the key is a random 512-bit string K. The scheme uses two 512 bit constants $ipad = 3636...36, opad = 5C5C...5C$, in HEX notation. Then we define

$$HMAC_K(m) = SHA1(\ (K \oplus opad)\ ||\ SHA1((K \oplus ipad)||x)\ )$$

where $||$ means concatenation of bit strings. This scheme generalizes naturally to be based on any hash function instead of SHA-1. It can be proved secure when the hash function is modelled as a random oracle, and no better result is known when we want to base HMAC on an *arbitrary* hash function. If security on the random oracle model was all we wanted, then a much simpler scheme where one just appends the key to the message before hashing, would be sufficient. However, in many cases the hash function is built by iterating a compression function with fixed size input, like SHA-1. Then the extra complexity of applying the key twice, xor'ed with different strings, allows to prove a result "in the real world". Namely, HMAC is secure if the hash function is collision resistant *and* if the basic compression function is secure in a weak sense when used as a MAC : part of the input to the function is used as key, the rest as message, and the adversary must forge MAC's even when the messages involved are partly unknown to him. No proof assuming only collision-resistance is known. Nevertheless, this is a popular scheme, due to its efficiency and because it does not rely on cryptosystems, which are sometimes under export restrictions.

5 Combining Signatures and Hashing

This section assumes that you have seen plain RSA and El Gamal signatures – we looked at RSA above, and El Gamal is described in Stinson, sec. 7.3).

Exercise 2 One may wonder how El Gamal came up with the apparently strange looking equation one checks to verify an El Gamal signature. Most likely, it was done simply by trying some possibilities until one that seemed to work was found. But in fact, there are many possible variants of this equation, and it is not even clear that the original one was the best. Here is one variant: Let the public key be $G = Z_p^*, \alpha, \beta$ where α is a generator of Z_p^*, and $\beta = \alpha^a \bmod p$ and a is the secret key. The signature is a pair (γ, δ) that satisfies:

$$\gamma = \alpha^\delta \beta^{\gamma' m} \bmod p$$

where $\gamma' = \gamma \bmod p - 1$. Describe how one would compute a signature from message m and the secret key a (and the information in the public key). Compare your method to the way one computes an nomal El Gamal signature: do you see any advantages or disadvantages of this variant? [1]. Finally: In Stinson, it is proved that standard El Gamal is insecure if one uses the same γ in signatures on two different messages. Does this variant have the same problem, and if so, why?

In practice, schemes like El Gamal an RSA are never directly used in practice. There are several reasons for this: first, as messages get longer, it quickly becomes too inefficient to apply RSA or El Gamal to the entire message. Moreover, neither system satisfy the security definition. For RSA, this trivially follows from the multiplicative property: given public key n, e and signatures $s_1 = m_1^d \bmod n, s_2 = m_2^d \bmod n$ on messages m_1, m_2, one can easily compute the signature $s_1 s_2 \bmod n$ on a new message $m_1 m_2 \bmod n$, since $s_1 s_2 = (m_1 m_2)^d \bmod n$. For El Gamal a more complicated computation also allows to make new signatures from old ones.

Using a fast and collision resistant hash function h, we can hope to solve both problems at the same time: we simply hash the input message m to get $h(m)$. We then map this to plaintext space on the signature scheme. For SHA-1 and RSA, this means creating a full size RSA block from the 160 bits output from SHA-1. This is usually done using some fixed injective

[1] There is no single correct answer here, you should not be looking for some huge difference between the methods

function from bit strings to plaintext space. Such a mapping is often called a *padding scheme*. Concrete methods for this are specified in several international standards, for instance the ISO 9796, or the PKCS series. The result is called $pad(h(m))$. We finally apply the original signature scheme to sign $pad(h(m))$. This way, we only have to apply the slow signature scheme to a single block. The collision intractability implies that an adversary cannot replace the message by another one with same image under h. Moreover, in the case of RSA, if he tries to exploit the multiplicative property as above, he will be faced with the problem of trying to find a preimage under h of $pad^{-1}(pad(h(m_1))pad(h(m_2)) \bmod n)$. If the hash function is SHA-1, or something else that has "nothing to do" with modular arithmetic, this seems to be just as difficult as inverting h on a random element in the image – something we know is hard by the collision resistant property. Unfortunately, no proof of this is known for any concrete hash function. The best we know how to do is to prove security in the random oracle model, where we assume that sending m to $pad(h(m))$ from the adversary's point of view is not better than sending it to a random value. This is known in the litterature as the security proof for Full Domain Hash (FDH).

The best result we know how to prove in the "real world" is from [2], and gives in some sense the next-best thing: if both hash function and signature scheme *were already secure*, then the combination of them is also secure. Given signature scheme Σ and hash function generator $\mathcal{H}$, we can define a new scheme Σ': To generate keys, we first make keys for Σ by running G, then we run $\mathcal{H}$ to get h where we assume we can make h such that it maps into the plaintext space defined by Σ. h is included in the public key of Σ'. You can sign any bit string m in the combined scheme Σ', by first computing $h(m)$, and then the signature $A_{sk}(h(m))$. It should be obvious how to verify a signature in the new scheme.

We now have

THEOREM 3 *If $\mathcal{H}$ is collision intractable and Σ is secure, then Σ' is secure.*

To prove the theorem, we assume that there is an enemy E' that breaks Σ' under a chosen message attack.

Consider the following algorithm E: it gets as input a hash function h, a public key pk from Σ, and an oracle O that on input a message x returns the Σ-signature $A_{sk}(x)$, where sk is the secret key corresponding to pk, i.e. E gets to do a chosen message attack on Σ.

Now E does the following:

- It starts E' on input pk, h (this is a public key in the combined scheme Σ').
- When E' makes an an oracle call, i.e. it wants the Σ' signature on a message m, this is handled by computing $h(m)$, calling O on input $h(m)$ and returning to E' the result $A_{sk}(h(m))$.
- When E' outputs a message m_0 and a signature s_0, E also outputs these values, plus the set of messages M for which E' made oracle calls.

The fact that E' by assumption is successful with large (non-negligible) probability means that there is a good chance that $m_0 \notin M$, and that s_0 is a valid Σ-signature on $h(m_0)$. Given that this happens, either $h(m_0) \in h(M_0)$ or $h(m_0) \notin h(M_0)$ so at least one of these cases must occur with non-negligible probability.

1. If $h(m_0) \notin h(M_0)$, we have forged the Σ-signature on a new message $h(m_0)$: we never asked O for a signature on $h(m_0)$.
2. If $h(m_0) \in h(M_0)$, we have found a collision for h.

It follows that the assumption that E' break Σ' leads to a contradiction with at least one of the assumptions in the theorem: if case 1 occurs with large probability, we can take a public key P as input, choose h ourselves and run E on h, pk to break Σ under a chosen message attack, i.e. if we are given an oracle O. If case 2 occurs with large probability, we can take a hash function h as input, run G ourselves to get sk, pk, and run E on input h, pk to find a collision for h (note that in this case, we chose sk ourselves, so the oracle O required by E is trivial to implement).

Note that it is essential for the proof of the result that h and (sk, pk) are chosen independently when we generate keys for the combined scheme. One can make artificial examples where h depends on sk, pk such that even though h and sk, pk are secure by themselves, the combination is not.

The results says that we can combine a signature scheme and a hash function that are already maximally secure, without loosing security. But it does not apply to signature schemes with less security. For instance, plain RSA is not secure against a chosen message attack because of the multicative property. Thus, combining it with a strong hash function may certainly result in a secure scheme, but we cannot use the above theorem to establish this fact.

6 Existence of good Signature Schemes

It is easy to see that signature systems cannot exist unless one-way functions exist: the generator algorithm G can be seen as a mapping from the random choices it makes to the public key pk. If this mapping is not one-way, an adversary could reconstruct a good set of random choices, and run G himself using these choices (and the right value of pk, which is public). This will lead the right public key pk being generated, but will also produce the matching secret key, and so the signature scheme could not be secure.

On the other hand, it can be shown [3] that secure signatures can be constructed from any one-way function, and so we have

THEOREM 4 *One-way functions exist iff secure signature schemes exist.*

The proof of this is very long and complicated and will not be given here. It is interesting to note, however, that apparently this puts public key signature schemes on a different status than public key *encryption* schemes: those schemes seem to require something stronger, either trapdoor functions such as RSA or one-way functions with other extra properties, such discrete log based functions.

To give a flavor of how one can prove the general result, we look briefly at a weaker result which says that collision resistant hash functions are sufficient. We begin by a scheme that may look rather silly at first sight, but nevertheless is the main building block. This is known as the Lamport-Diffie (LD) one-time signature scheme:

To generate keys, we run $\mathcal{H}$ on input k to get hash function f (we use the notation f in order to reserve h for another function we need later). Then we choose t pairs of inputs

$$(x_0^1, x_1^1), (x_0^2, x_1^2), ..., (x_0^t, x_1^t)$$

at random, say, from $\{0,1\}^{k+1}$. The public key is now

$$f, (y_0^1 = f(x_0^1), y_1^1 = f(x_1^1)), ..., (y_0^t = f(x_0^t), y_1^t = f(x_1^t)),$$

while the secret key is the pairs of x-values. The message space is $\{0,1\}^t$. The signature on a bit string $b_1, ..., b_t$ is $x_{b_1}^1, ..., x_{b_t}^t$., so signature verification is obvious: one just evaluates f on the x-values in the signature and verifies if the result matches the correpsonding y-values in the public key.

This scheme will be secure, if it is used to sign at most ONE message. Already if we sign 2 messages with the same public key, we may be in trouble:

Exercise 3: Assume the LD scheme has been used to sign 2 different messages $b_1, ..., b_t$, $b'_1, ..., b'_t$. Let ℓ be the number of indices i for which $b_i \neq b'_i$. Show that given the two signatures, an adversary can now easily sign $2^\ell - 2$ new messages.

To prove security when only one message is signed, the point is that a chosen message attack on this scheme allows the adversary to get the signature on one message $b_1, ..., b_t$, and so he learns the values $x^1_{b_1}, ..., x^t_{b_t}$. But if he can sign a new message $b'_1, ..., b'_t$, note that for at least one i, we have $b'_i \neq b_i$, and so he needs to come up with $x^i_{b'_i} = x^i_{1-b_i}$, which was not given in the signature. This means inverting f in a random point $h(x_{1-b_i})$ which we know is hard by Lemma 1. The fact that you know other x-values does not help, since the x's were chosen independently. More formally, here's a reduction showing how to use a sucessful forger F to build a machine that inverts f in a given point:

1. We are given f and y (where $y = f(x)$ for random x). Now run a normal key generation for the LD scheme using f as the one-way function, with one exception: we choose indices $i \in \{1, ..., t\}, b \in \{0, 1\}$ at random, and set $y^i_b = y$. Give the resulting public key as input to F. So we now know a preimage of all the y's in the public key, except for one.

2. Get a message $b_1, ..., b_t$ to sign from F. If $b_j = b$, return ? and stop. Otherwise sign the message as usual and give the signature to F.

3. Get a signature on a new message $b'_1, ..., b'_t$ from F, this consists of t values $x_1, ..., x_t$. If the signature is valid, and $b'_j = b$, then return x_j, otherwise return ?.

Exercise 4: Assume that F succeeds in making a valid signature on a new message with probability ϵ. Show that the probability of returning ? in Step 2 above is $1/2$. Use this to show that we succeed in finding a preimage of y with probability at least $\epsilon/2t$.

This scheme could in fact be based on just a one-way function, but is of course quite useless in the form we gave it here. But since we assume we have collision-intractable functions, we can do better: first note that by the Theorem 3 above, we can combine the LD scheme with a hash function, and we will still have a secure scheme. This will of course still be a one-time scheme, but since we now hash messages and then sign the hash value in the LD scheme, we can sign messages of *any* length. This allows us to build a scheme where we can sign any number of messages securely. The idea is that whenever we sign a message, we will also authenticate a new LD public key. We can then use the corresponding secret key to sign the next message, this again involves generating a new key pair, etc..

Key generation Construct a key pair pk_0, sk_0 for the LD scheme allowing to sign t-bit messages, for even t and choose a hash function $h : \{0,1\}^* \rightarrow \{0,1\}^{t/2}$. The public key is pk_0, h, while the secret key is sk_0. Initialize a list L containing for each previously signed message: an LD key pair, an LD signature, and a hash value (on a previously signed message). Initally, L has only one element, namely $((pk_0, sk_0), -, -)$, where the last two entries are empty.

Signing a message Assume we are about to sign message m_i, where this is the i'th message we sign. Thus the last entry in L is a key pair pk_{i-1}, sk_{i-1} a signature s_{i-1} and a hash value $h(m_{i-1})$. Generate an LD key pair pk_i, sk_i with parameters as above. Compute $h(pk_i), h(m_i)$ and compute the signature s_i on the concatenation of these two strings under the key pair pk_{i-1}, sk_{i-1}. Return as signature on m_i the sequence

$$(s_i, pk_i, h(m_i)), (s_{i-1}, pk_{i-1}, h(m_{i-1})), ..., (s_1, pk_1, h(m_1)).$$

Append $((pk_i, sk_i), s_i, h(m_i))$ to L.

Verifying a signature We are given a message m, public key h, pk_0 and a signature $(s_i, pk_i, h(m_i)), (s_{i-1}, pk_{i-1}, h(m_{i-1}), ..., (s_1, pk_1, h(m_1))$. First verify that $h(m)$ matches the hash value in the first triple in the signature. Then, for $j = i$ down to 1, verify that s_j is a valid LD signature on $h(pk_j)||h(m_j)$ under public key pk_{j-1}. Note that for $j = 1$, this involves verifying a signature under pk_0, which is part of the public key.

It is quite straight forward to show that this scheme is secure, based only on collision resistance of h and security of the LD scheme (which was again based on one-wayness of the function f). A problem with the scheme is that the length of a signature grows linearly with the number of messages signed. But this is only because we organize data on previous signatures as a linear list (for simplicity). If instead we build a tree structure, where each public key authenticates a message and 2 new public keys, we immediately get a scheme where the signature length only grows logarithmically.

The more general result that any 1-way function is enough for signatures, was obtained first by observing that collision-resistance for h in the scheme above is overkill: a slightly weaker type of functions known as One-way Universal Hash Functions (OWUF's) will suffice, and finally by a construction of OWUF's based on arbitrary one-way functions.

7 Authentication in Practice

Let us finally return to the scenario where A sends messages to B over an insecure channel. Depending on which precise security goals we have, the authentication schemes we have seen may or may not be enough to solve the problems by themselves.

The reason why there may be problems is that if I receive for instance a message m with a digital signature from A, this only proves that *at some point*, A produced this message. It leaves open the possibility for an adversary to take a copy of the signed message and send it to me as many times as he wants. This is known as a *replay attack*. If A is a bank customer, the reciever is a bank, and the message is a request to transfer money from A's account to someone else, the security problems with replay should be evident.

So what we really want in practice is often not just to protect the integrity of messages, but to have a real authentic channel, that is, we want B to receive the exact same sequence of messages that A sends, and if this not possible, we want to come as close to this as we can. If we do not have physical control over the communication line, we cannot prevent an adversary from physically blocking some messages or from reordering them before they are received, but we should at least be able to make sure that replayed messages are filtered out.

One trivial way to ensure this is have the sender make sure that he never sends exactly the same message twice, for instance by appending a sequence number, and also add an authenticator computed over both message and sequence number. Then we can have the receiver store every message he ever receives (or at least the sequence numbers). This will allow the receiver to filter out every replayed message, and also to correctly place all messages he gets in the order they were sent.

Of course, this is hardly a practical solution. Even if we do use sequence numbers, we cannot expect the receiver to store more than the sequence number of a small number of recently accepted message. Suppose, for instance, that he stores only the last accepted message. Then if the receiver has stored number n and the next message does not have number $n+1$, we need a rule for the receiver to decide what to do. The problem is that in some cases, messages may get lost or be reordered even if no attacker is present. One possibility is to require that the next message must have number $\geq n$ to be accepted. This will prevent replay, but may also mean that good messages get rejected if they arrive too late.

A different approach appends a timestamp to messages. To be of any use, this of course requires that the receiver checks the time stamp, to ensure that it is not too far from his own time. Some compromise has to be made here, so that on one hand good messages are accepted, but on the other hand replayed messages are rejected. This requires that there is some synchronization between sender and receiver, and also that messages are delivered quickly enough. On the other hand, there is no need here to remember any information about earlier messages.

Finally, one may use interaction: we can have the receiver first send a number R to the sender. This number can be chosen at random, or be a sequence number, the only real requirement is that it has never been used before. Then the sender sends the message plus a MAC computed over the message and R. This will prevent replay, and there is no need for sychronization, but one does need to at least remember some state in order to ensure that R-values are not reused. In addition this is not trivial to implement securely if the receiver has to handle several connections in parallel.

References

[1] Bellare, Kilian and Rogaway: *The security of CBC*, Proc. of Crypto 94.

[2] Damgård: *Collision-Free hash functions and public-key sugnature schemes*, Proc. of EuroCrypt 87.

[3] Rompel: *One-way functions are necessary and sufficient for digital signatures*, proc. of STOC 90.