Digital Signatures

KG

November 25, 2016

Contents

1 Introduction

In this note, we consider the following problem. Alice wants to send a message to Bob via some channel. Eve has access to the channel and she may tamper with anything sent over the channel, and even introduce her own messages. Alice wants her message to Bob to arrive without modification, or if it has been tampered with, Bob should notice.

Various solutions using public key encryption schemes are possible, but a different primitive is more convenient, namely digital signatures. The basic idea is explained and defined in Section 2. Section 3 discusses hash functions and how they can be used to make some signature schemes more convenient. Section 4 and Section 5 discuss digital signature schemes based on the RSA problem and the discrete logarithm problem, respectively. Finally, in Section 6 we use digital signatures to construct a secure version of the Diffie-Hellman protocol.

This text is intended for a reader that is familiar with mathematical language, basic number theory, basic algebra (groups, rings, fields and linear algebra) and elementary computer science (algorithms), as well as the Diffie-Hellman protocol, discrete logarithms and the RSA public key cryptosystem.

This text is informal, in particular with respect to computational complexity. Every informal claim in this text can be made precise, but the technical details are out of scope for this note.

This text uses colour to indicate who is supposed to know what. When discussing cryptography, red denotes secret information that is only known by its owner, Alice or Bob. Green denotes information that Alice and Bob want to protect, typically messages. Blue denotes information that the adversary Eve will see. Information that is assumed to be known by both Alice and Bob (as well as Eve) is not coloured.

2 Digital Signatures

Alice, Bob and a number of other people want to be able to send messages to each other, and they want to notice any tampering with those messages. Alice does not want to manage a long-term secret for each correspondent, so symmetric key techniques such as message authentication codes cannot be used. Alice is willing to manage public information for each correspondent.

In this situation, what is needed is digital signatures.

Definition 1. A *digital signature* scheme consists of three algorithms $(\mathcal{K}, \mathcal{S}, \mathcal{V})$.

- The *key generation* algorithm $\mathcal{K}$ takes no input and outputs a *signing key* sk and a *verification key* vk. To each key pair there is an associated message seg denoted by $\mathcal{M}_{sk}$ or $\mathcal{M}_{vk}$.
- The *signing* algorithm $\mathcal{S}$ takes as input a signing key sk and a message $m \in \mathcal{M}_{sk}$ and outputs a signature σ.
- The *verification* algorithm $\mathcal{V}$ takes as input a verification key vk, a message $m \in \mathcal{M}_{vk}$ and a signature σ, and outputs either 0 or 1.

We require that for any key pair (vk, sk) output by $\mathcal{K}$ and any message $m \in \mathcal{M}_{vk}$

$$\mathcal{V}(vk, m, \mathcal{S}(sk, m)) = 1.$$

We interpret a 1 from the verification algorithm as a *valid* signature, and a 0 as an *invalid* signature. A valid signature that was created without the signing key is a *forgery*.

Definition (informal) 2. A signature scheme is *secure* if it is hard to create a valid signature on a message without the signing key, even when you can see valid signatures on many different messages.

3 Hash Functions

A vital component for designing practical digital signature schemes is the hash function. The idea is that we can easily build signature schemes, but often we get a scheme with a very small message space. Moreover, many of the schemes we build suffer from a weakness where it is very easy to come up with signatures on random messages, even without the signing key.

If we combine our primitive signature schemes with a suitable hash function, we can extend the message space and protect against these designed-in weaknesses.

The idea for the construction is that instead of signing the message itself, we shall sign a hash of the message. Let $(\mathcal{K}, \mathcal{S}', \mathcal{V}')$ be a signature scheme and let $h : S \to T$ be a hash function such that T is a subset of the signature scheme's message space. We construct a new signature scheme $(\mathcal{K}, \mathcal{S}, \mathcal{V})$ with message space S as follows. The key generation algorithm is unchanged. The signing algorithm creates a signature of a message m under the signing key sk be computing $\mathcal{S}'(sk, h(m))$. On input of vk, m and σ, the verification algorithm outputs $\mathcal{V}'(vk, h(m), \sigma)$.

Signing a hash of the message could be a security problem if we could find two messages that have the same hash. A signature on one of the messages would also be a signature on the other message, which would be bad. Ideally, we would like the hash function to be injective, but that would not allow us to expand the message space. Instead, we shall settle for a hash function that merely "looks" injective.

Definition 3. Let $h : S \to T$ be a function. A *preimage* of $t \in T$ is an element $s \in S$ such that $h(s) = t$. A *second preimage* for $s_1 \in S$ is an element $s_2 \in S$ such that $s_1 \neq s_2$ while $h(s_1) = h(s_2)$. A *collision* for h is a pair of distinct elements $s_1, s_2 \in S$ such that $h(s_1) = h(s_2)$.

The following two definitions are informal. It is possible to give precise definitions, but this is out of scope for this note.

Definition (informal) 4. Let $h : S \to T$ be a function. We say that it is *one-way* if it is an infeasible computation to find a preimage of a random $t \in T$ and to find a second preimage for a random $s \in S$.

Definition (informal) 5. Let $h : S \to T$ be a function. We say that it is *collision resistant* if it is an infeasible computation to find collisions for h and to find a second preimage for a random $s \in S$.

We quickly note that if you can find second preimages, you can also find collisions. The converse does not have to be true. It follows that if finding a collision is an infeasible computation, the hash function will be collision resistant and it will behave like an injective function in practice.

It would be natural if the ability to find preimages implied the ability to find second preimages. This is not true, as implied by the following exercise.

Exercise 1. Let $h : S \to T$ be a hash function, and suppose that $T \subseteq S$, but $|T| < |S|$.

Let $h' : S \to \{0,1\} \times T$ be the hash function defined by

$$h'(s) = \begin{cases} (0, s) & s \in T, \\ (1, h(s)) & \text{otherwise.} \end{cases}$$

Show that for half of all elements of $\{0,1\} \times T$, it is easy to find preimages, but for none of those preimages there exists a second preimage.

We see that if our hash function is collision resistant, the hash function behaves as an injective function and the above signature scheme $(\mathcal{K}, \mathcal{S}, \mathcal{V})$ is no less secure than the original scheme $(\mathcal{K}, \mathcal{S}', \mathcal{V}')$.

If our hash function is one-way, the above scheme may actually be more secure than the original scheme.

We note that there is a different security notion for hash functions, *random-looking*, which is that the hash function should in some sense look like a typical "random" function. We do not discuss this notion further, except to note that this is different from the above notions.

3.1 Attacks

The main generic attack on hash functions is to hash random messages until a collision is found. Let $h : S \to T$ be a hash function. Choose a large number of random messages $s_1, s_2, \dots, s_l$ and compute their hashes. We store the messages and their hashes in a list sorted by their hash values. By the birthday paradox, as soon as l is roughly $\sqrt{|T|}$, we should have a reasonable likelyhood of finding a collision.

This result gives us a minimal size for the set T, namely that $\sqrt{|T|}$ should be an infeasible computation. However, the attack described above requires a lot of memory.

Exercise 2. Let $l = 10\lfloor\sqrt{|T|}\rfloor$. Let $g : T \to S$ be an injective function, and let $f : T \to T$ be the function defined by $f(t) = h(g(t))$. Let t_0 and t'_0 be two distinct elements of T. Define two sequences by the equations

$$t_i = f(t_{i-1}) \qquad \text{and} \qquad t'_i = f(t'_{i-1}).$$

a. Imagine that the two sequences are really sequences of random elements. Argue that with reasonable probability, $t'_j = t_i$ for some i, j smaller than l.

b. Suppose h is "random-looking". Argue that the above result should apply even when the sequences are determined solely by the random choice of t_0 and t'_0, respectively.

c. Suppose $t'_j = t_l$ for some $j < 2l$, and $t'_j \neq t_0$ for any $j < l$. Show how you can find, given j, a collision in h using at most $3l$ evaluations of g and h.

d. Suppose h is "random-looking". Argue that with reasonable probablity, you can find a collision in h using about $6l$ hash evaluations.

3.2 Compression Functions

Typically, the domain of a hash function is much larger than the range. For example, $\log_2 |T|$ will typically be between one hundred and a few thousand, while $\log_2 |S|$ is from 2^{64} and upwards. A hash function where the domain is larger than the range, but not by much, is called a *compression function.*

We are interested in compression functions for two reasons. First of all, it is probably easier to construct compression functions than large-domain hash functions. And second, we have efficient constructions that turn secure compression functions into secure hash functions.

We begin by discussing the latter construction. So let $f : S' \to T$ be a compression function. Suppose further that there is a set $\mathcal{A}$ such that $\{0, 1\} \times \mathcal{A} \times T$ is either a subset of S' or trivially injects into S'. Then we can consider the restriction of f to $\{0, 1\} \times \mathcal{A} \times T$ instead.

We shall now construct a hash function $h : S \to T$. The domain S is the set of all finite sequences of elements from $\mathcal{A}$, denoted by $\mathcal{A}^*$. Let $t_0 \in T$ be a fixed element of T.

The value s we want to hash is a sequence $s_1 s_2 \dots s_L$ of elements from $\mathcal{A}$. The function h is computed using the formula

$$t_1 = f(1, s_1, t_0), \qquad\qquad t_i = f(0, s_i, t_{i-1}), \qquad 2 \le i \le L.$$

Then $h(s) = t_L$.

The cost (in terms of compression function evaluations) of computing h is linear in the length of the message to be hashed. If it is easy to compute f, then computing h is quite efficient.

If the compression function is one-way and collision resistant, we would like the above defined hash function to be both one-way and collision resistant.

Theorem 1. *Given a collision (s, s') in the above constructed hash function, we can find a collision in the compression function f using at most $2L$ evaluations of f, where the length of s and s' is at most L.*

Proof. Let $s = s_1 s_2 \dots s_L$ and $s' = s'_1 s'_2 \dots s_{L'}$, with $L \le L'$.

We know that

$$f(0, s_L, t_{L-1}) = f(0, s'_{L'}, t'_{L'-1}).$$

Either we have found our collision, or $s_L = s_{L'}$ and $t_{L-1} = t'_{L'-1}$. The latter means that

$$f(0, s_{L-1}, t_{L-2}) = f(0, s'_{L'-1}, t'_{L'-2}).$$

We continue in this way until either we find a collision or we reach the beginning of s. Then if $L = L'$, we must have that $s_1 \ne s'_1$ since $s \ne s'$, which gives us a collision since

$$f(1, s_1, t_0) = f(1, s'_1, t_0).$$

If $L < L'$, we must have that

$$f(1, s_1, t_0) = f(0, s'_{L'-L+1}, t'_{L'-L}),$$

which will also be a collision.

We can find this collision by first computing $t'_1, t'_2, \dots, t'_{L'-L}$, then computing the pairs $(t_1, t'_{L'-L+1}), (t_2, t'_{L'-L+2}), \dots, (t_L, t'_{L'})$. One of these pairs will be our collision. The claim follows. □

The theorem says that from any collision in the constructed hash function h, it is easy to find a collision in the compression function f. Which means that if our compression function is collision-resistant, the constructed hash function is also collision-resistant.

Exercise 3. Consider the above construction. Suppose you have an "oracle" that for any $t \in T$ will provide you with a reasonable-length preimage of t under h. Show that you can use this oracle to find preimages for any $t \in T$ under f.

As for collision resistance, the consequence of the above exercise is that if we can construct a compression function where finding preimages is an infeasible computation, we can construct a hash function where finding preimages is an infeasible computation.

To conclude that the hash function is one-way, we must also consider second preimages. Unlike for preimages and collision, being able to find second preimages for h does not seem to imply the ability to find second preimages of f. But the ability to find second preimages for h implies the ability to find collisions for h, which implies the ability to find collisions for f. That is, if we can find second preimages for h, then we can find collisions in f.

This means that if f is one-way and collision-resistant, then h is one-way and collision-resistant hash function.

Note that the construction uses the 0 and the 1 to differentiate the start of the iteration. This is important in the proof, since without this differentiation, we could have run into problems when messages of different length collided.

There are other ways to construct hash functions from compression functions.

Exercise 4. Suppose we have a compression function $f : \mathcal{A} \times T \to T$, and that $\{0, 1, \dots, 2^{64} - 1\}$ is a subset of $\mathcal{A}$. Let t_0 be a fixed element of T.

We define two hash functions for messages that are sequences of elements from $\mathcal{A}$ of length less than 2^{64}, using the two recursive formulas

$$t_1 = f(L, t_0), \qquad t_{i+1} = f(s_i, t_i), \quad 1 \le i \le L,$$

and

$$t_i = f(s_i, t_{i-1}), \quad 1 \le i \le L, \qquad t_{L+1} = f(L, t_L).$$

In either case, the hash of the message is t_{L+1}.

For each hash function, state and prove a result similar to that of Theorem 1 for that hash function.

Note that to use the first construction in Exercise 4, you must know the length of the message before you begin hashing it. There are reasonable cases where you want to begin hashing a message before you know the entire message, and in particular before you know the length of the entire message.

Hash functions are used for many things in cryptography, and frequently the security requirements are different from what we need for digital signatures.

One example is to use a hash function as a message authentication code, simply by computing $\mu(k, m) = h(k||m)$, where $k||m$ denotes the concatenation of the key and the message. As the following exercise shows, our construction cannot be used like this.

Exercise 5. Let $h : \mathcal{A}^* \to T$ be the hash function constructed above. Suppose you are given a value t such that $t = h(m)$. Show that you can easily compute $h(m||m')$ for any m', even when you do not know m, only t. (Here, $m||m'$ denotes the concatenation of m and m'.)

3.3 Constructing a Compression Function

Let G be a cyclic group of prime order n, and let x and y be non-zero elements. Then we can construct a compression function $f_{x,y} : \{0, 1, 2, \ldots, n-1\} \times \{0, 1, 2, \ldots, n-1\} \to G$ as

$$f_{x,y}(u, v) = x^u y^v.$$

When x and y are clear from context, we shall write simply f for $f_{x,y}$.

If it is hard to compute discrete logarithms in G, then it is hard to find both preimages and collisions for this compression function, provided x and y have been chosen at random from G.

Theorem 2. *Suppose we know a collision* $((u, v), (u', v'))$ *for* f*. Then we can compute* $\log_x y$ *using* 3 *arithmetic operations.*

Proof. If we have a collision, we know that

$$x^u y^v = x^{u'} y^{v'}.$$

Since $(u, v) \neq (u', v')$ and the above equation holds, we have that $u \neq u'$ and $v \neq v'$. This means that

$$y = x^{-(u-u')(v-v')^{-1}}.$$

The claim follows. ☐

Exercise 6. Suppose you have an "oracle" that for any x, y, z of your choice will find one preimage of z under the hash function $f_{x,y}$. Explain how you can use this oracle to compute discrete logarithms in G.

Using this compression function and the construction from the previous section, we have a one-way and collision-resistant hash function. However, this hash function is of theoretical interest only, since we have much faster constructions that also have other interesting properties.

4 RSA Signatures

We briefly recall the textbook RSA public key cryptosystem. The key generation algorithm chooses two primes p and q and finds e and d such that $ed \equiv 1 \pmod{\operatorname{lcm}(p-1, q-1)}$. The encryption key is (n, e), where $n = pq$ and the decryption key is (n, d).

To encrypt a message $m \in \{0, 1, \ldots, n-1\}$, we compute $c = m^e \bmod n$. To decrypt a ciphertext c, we compute $m = c^d \bmod n$.

It turns out that we can construct a very simple signature scheme based on this. The *textbook RSA* signature scheme $(\mathcal{K}, \mathcal{S}, \mathcal{V})$ works as follows.

- The *key generation* algorithm $\mathcal{K}$ chooses two large primes p and q. It computes $n = pq$, chooses e and finds d such that $ed \equiv 1 \pmod{\operatorname{lcm}(p-1, q-1)}$. Finally it outputs $vk = (n, e)$ and $sk = (n, d)$. The message set associated to vk is $\{0, 1, 2, \ldots, n-1\}$.

- The *signing* algorithm $\mathcal{S}$ takes as input a signing key (n, d) and a message $m \in \{0, 1, 2, \ldots, n-1\}$. It computes $\sigma = m^d \bmod n$ and outputs the signature σ.

- The *verification* algorithm $\mathcal{V}$ takes as input a verification key (n, e), a message $m \in \{0, 1, 2, \ldots, n-1\}$ and a signature $\sigma \in \{0, 1, 2, \ldots, n-1\}$. It outputs 1 if $\sigma^e \equiv m \pmod{n}$, otherwise 0.

Exercise 7. The above is an informal description of a signature scheme. Write down carefully what the three algorithms $\mathcal{K}$, $\mathcal{S}$ and $\mathcal{V}$ are. Show that the triple $(\mathcal{K}, \mathcal{S}, \mathcal{V})$ is a signature scheme.

4.1 Attacks

As for textbook RSA public key encryption, as long as the RSA modulus n chosen by the key generation algorithm is hard to factor, the above digital signature scheme is useful. But it is not entirely secure.

Exercise 8. Suppose $(n, 3)$ is a verification key. Forge a signature on the message 8.

Exercise 9. Suppose (n, e) is a verification key. Explain how to create a random message with a forged signature.

Malleability Just like the RSA encryption scheme, the RSA signature scheme is malleable, and this can be used to create forgeries.

Exercise 10. Suppose you have two messages m, m' and signatures σ, σ' on those messages under the verification key (n, e). Show how to construct a signature on the product $mm' \bmod n$.

Exercise 11. Suppose Eve wants to have Alice' signature on a message m. Suppose also that she is capable of getting Alice to sign any other message. Show how Eve can use this to forge a signature on m.

Short Messages We want to use the idea from Section 3 with a hash function $h : S \to T$ that is both collision resistant and one-way, and where T is a set of integers all smaller than any RSA modulus we choose. This *hashed RSA* signature scheme works as follows.

- The *key generation* algorithm is exactly the same as the RSA key generation algorithm.
- The signing algorithm $\mathcal{S}$ takes a signing key $sk = (n, d)$ and a message $m \in S$ as input. It computes $\sigma = (h(m))^d \bmod n$ and outputs the signature σ.
- The verification algorithm $\mathcal{V}$ takes as input a verification key (n, e), a message $m \in S$ and a signature $\sigma \in \{0, 1, 2, \dots, n-1\}$. It outputs 1 if $\sigma^e \equiv h(m) \pmod{n}$, otherwise 0.

Exercise 12. Explain why the attacks from Exercises 8 and 9 fail against this scheme.
Hint: The hash function is one-way.

Exercise 13. Suppose that the hash function used is also random-looking. Explain why the attacks from Exercises 10 and 11 become much more difficult.

However, most practical hash functions have outputs that are very short relative to an RSA modulus, and this allows us to develop an attack.

Exercise 14. Let (n, e) be a verification key with corresponding signing key (n, d). Suppose you have messages $m_1, m_2, \dots, m_l$, and integers $\ell_1, \ell_2, \dots, \ell_l$, $\sigma_1, \sigma_2, \dots, \sigma_l$ and s_{ij}, $1 \le i, j \le l$, such that

$$h(m_i) = \prod_{j=1}^{l} \ell_j^{s_{ij}}, \qquad 1 \le i \le l \text{ and}$$
$$h(m_i) \equiv \sigma_i^e \pmod{n}.$$

We shall assume that the matrix $S = (s_{ij})$ is invertible modulo e, and that $R = (r_{ki})$ is an inverse modulo e.

a. Show that

$$\sum_{i=1}^{l} r_{ki} s_{ij} = \delta_{kj} + t_{kj} e$$

where $\delta_{kj} = 1$ if $k = j$, otherwise $\delta_{kj} = 0$.

b. Show that

$$\prod_{i=1}^{l} \sigma_i^{r_{ki}} \equiv \ell_k^d \prod_{j=1}^{l} \ell_j^{t_{kj}} \pmod{n}.$$

c. Explain how we can easily compute $\ell_k^d \bmod n$ from the above.

d. Suppose you are given a message m such that

$$h(m) = \prod_{i=1}^{l} \ell_i^{u_i}.$$

Explain how you, given the above, can easily compute $h(m)^d \bmod n$.

If we let $\ell_1, \ell_2, \dots, \ell_l$ be small primes, then if our hash function h has small integers as output, we can quickly find messages $m, m_1, m_2, \dots, m_l$ such that their hashes are products of powers of our small primes. Which means that if we get signatures on $m_1, m_2, \dots, m_l$, we can construct a forgery on m.

4.2 Secure Variants

It turns out that it is quite easy to fix the hashed RSA signature scheme discussed above. All we need is a random-looking, one-way, collision-resistant hash function whose domain is almost all of $\{0, 1, 2, \dots, n-1\}$. In which case the hashed RSA scheme is secure, and is known as the *full domain hashed RSA* signature scheme, or *RSA-FDH*.

Exercise 15. Explain why the attack from Exercise 14 fails against RSA-FDH.

5 Schnorr Signatures

In this section, we shall develop the well-known Schnorr signature scheme. While the Schnorr scheme is not used much as a signature scheme, its development is interesting and many other signature schemes are very similar to the Schnorr system.

For the remainder of this section, let G be a group of prime order n, and let g be a generator.

5.1 How to Prove That You Know a Secret

We begin with a very different question. Suppose Alice knows a secret number $a \in \{0, 1, 2, \dots, n-1\}$. Bob does not know the secret number, but he knows $x \in G$ such that $x = g^a$. Alice wants to convince Bob that she really knows a.

Of course there is an adversary. Eve may want to cheat Bob by pretending to know a. Or Eve may want to cheat Alice by somehow learning a.

The latter point explains why Alice cannot convince Bob simply by revealing a to Bob. While Bob is honest, Alice may at some point in time also want to convince Eve that she knows a, after which Eve could cheat Bob by pretending to know a.

One thing Alice could do was to choose a random number r, compute $\alpha = g^r$ and $\gamma = r + a \bmod n$, and then show Bob α and γ. Bob accepts that Alice knows a if

$$g^\gamma \stackrel{?}{=} \alpha x.$$

The idea is that the above protocol does not reveal a to Bob, because Alice just as well could choose a random γ and compute α as $g^\gamma x^{-1}$.

Exercise 16. Use the above security argument to show how Eve can cheat Bob and pretend that she knows a, even though she only knows x.

We can improve on this procedure as follows. Instead of showing Bob both α and γ at once, Alice first shows Bob α. Then Bob is allowed to choose if he wants to see $\gamma = r$ or $\gamma = r + a \bmod n$. He accepts that Alice knows a if

$$g^{\gamma} \overset{?}{=} \alpha \qquad \text{or} \qquad g^{\gamma} \overset{?}{=} \alpha x, \text{ respectively.}$$

Note that we can encode Bob's choice β as a 0 or a 1, in which case the above formulas can be reduced to

$$\gamma = r + \beta a \bmod n \qquad \text{and} \qquad g^{\gamma} \overset{?}{=} \alpha x^{\beta}. \tag{1}$$

Exercise 17. a. Suppose Bob tells Eve what his choice β will be before Eve chooses α. Explain how Eve can choose α and then respond such that Bob will accept that she knows the secret, even though she only knows x.

If Bob does not tell Eve his choice early, explain why Eve can guess his choice, proceed as above and successfully cheat Bob, all with probability 1/2.

b. Suppose that Eve has chosen α and that she knows the correct response to make Bob accept, regardless of Bob's choice. That is, Eve knows γ_0 and γ_1 such that $g^{\gamma_\beta} = \alpha x^{\beta}$. Show that Eve can easily compute a from γ_0 and γ_1.

We wanted to ensure that if Alice runs this protocol with Eve, then Eve learns nothing about Alice's secret. We shall argue that if Eve can learn something from Alice, she can learn the same thing without Alice.

So suppose Eve gets α from Alice, chooses β, receives γ and from that exchange learns something about a.

Now Eve decides to do without Alice. Instead, she makes a guess β' at what challenge she will choose upon seeing α. Then she proceeds according to the first part of Exercise 17. With probability 1/2, she will guess her choice of challenge correctly, in which case her conversation would proceed exactly as if she were talking to Alice, which means that she would learn something about a. Of course, with probability 1/2, Eve will not guess correctly, so she may not learn anything, but in this case she can just try again.

We also wanted to ensure that Eve cannot cheat Bob. From the above exercise we know that Eve can successfully pretend to know a with probability 1/2, but unless she knows a, she cannot succeed with any greater probability.

It follows that Alice can convince Bob that she almost certainly knows a by repeating the above protocol many times. For each repetition, Eve would have probability 1/2 of cheating successfully, but for k repetitions her success probability sinks to 2^{-k}.

Doing k repetitions is quite inefficient, of course. Instead, we can do something slightly different. The problem is that Eve can guess Bob's choice and choose α based on that. But note that in (1), there is nothing that forces β to be just 0 or 1.

The protocol can then work as follows.

1. Alice chooses r uniformly at random from $\{0, 1, 2, \dots, n-1\}$, computes $\alpha = g^r$, and sends α to Bob.

2. Bob chooses β uniformly at random from $\{0, 1, 2, \dots, 2^k - 1\}$ and sends β to Alice.

3. Alice computes $\gamma = r + \beta a \bmod n$ and sends γ to Bob.

Bob accepts that Alice knows a if

$$g^\gamma \overset{?}{=} \alpha x^\beta.$$

By the above arguments, the probability that Eve cheats Bob should not be much larger than 2^{-k}. One problem is that our argument for why Eve does not learn anything from Alice was exactly the argument that proved that Eve could cheat Bob. Which means that strictly speaking, we no longer have an argument for why Eve will not learn anything by talking to Alice.

However, if Eve chooses her challenge without looking at Alice's α, then the argument from the first part of Exercise 17 applies, and we can use it to show that in this case Eve cannot learn anything about Alice's message.

Unfortunately, for the same reason, Bob cannot reveal his challenge before Alice reveals her α. The question is, how can we force Bob to choose his challenge before Alice reveals her α, but without Bob revealing his challenge?

Exercise 18. Let $h : S \to T$ be a hash function such that $\{0, 1, 2, \dots, 2^{2k} - 1\} \times \{0, 1, 2, \dots, 2^k - 1\}$ is a subset of the domain.

Suppose Bob chooses t and β and computes $\omega = h(t, \beta)$. He sends ω to Alice. The protocol then proceeds by Alice sending α to Bob, who responds with his already chosen β and the random number t. Alice verifies that ω equals $h(t, \beta)$ and responds with the correct γ, which Bob verifies as usual.

Under reasonable assumptions on the hash function h, it can be shown that ω does not reveal anything about β, and that Bob cannot find different t', β' such that $h(t', \beta') = \omega$.

Argue why Eve cannot cheat Alice (to learn something about a) or Bob (to convince him that Eve knows a) using the above protocol.

A different approach can be used if we have a "random-looking" hash function $h : S \to T$ where $G \times G$ is a subset of S and $T = \{0, 1, \dots, 2^k - 1\}$. Instead of choosing a random challenge, Bob can instead compute the challenge as $\beta = h(x, \alpha)$.

Since Eve no longer chooses her challenge when trying to cheat Alice, Eve will not be looking at α before deciding on her challenge, so as we have argued above, she should not learn anything new.

When Eve is trying to cheat Bob, however, she can know what challenge Bob will choose without actually sending α to Bob. She cannot know β until after she has chosen α, but she will still have the ability to look at many αs with corresponding βs, before she sends one α to Bob. While this does give her increased power, it can easily be neutralized by increasing k.

Of course, if Eve can compute β before sending α to Bob, so can Alice. We can therefore greatly simplify the process. Alice chooses r, computes $\alpha = g^r$, $\beta = h(x, \alpha)$, and $\gamma = r + \beta a \bmod n$. She then sends α, β and γ to Bob. Bob verifies that

$$\beta \overset{?}{=} h(x, \alpha) \qquad \text{and} \qquad g^{\gamma} \overset{?}{=} \alpha x^{\beta}.$$

An equivalent verification equation can be

$$\alpha \overset{?}{=} g^{\gamma} x^{-\beta}.$$

If the hash function is collision resistant (and a "random-looking" hash function should be), this equation will hold in practice if and only if

$$h(x, g^{\gamma} x^{-\beta}) \overset{?}{=} \beta.$$

This gives us further scope for simplification and making the formulas tidier. The process now works as follows. Alice chooses r, computes $\alpha = g^r$, $\beta = h(x, \alpha)$ and $\gamma = r - \beta a \bmod n$. She then sends β and γ to Bob. Bob verifies that

$$h(x, g^{\gamma} x^{\beta}) \overset{?}{=} \beta.$$

5.2 Schnorr Signatures

The Schnorr signature scheme is based on the ideas on how to prove that you know something, but the proofs are augmented by including something extra in the hash that generates the challenge.

Suppose $\mathcal{P}$ is a set of messages and we have a "random-looking" hash function $h : S \to T$, where $G \times G \times \mathcal{P}$ is a subset of S and $T = \{0, 1, \dots, 2^k - 1\}$.

The Schnorr signature scheme $(\mathcal{K}, \mathcal{S}, \mathcal{V})$ works as follows.

- The *key generation* algorithm $\mathcal{K}$ samples a number a uniformly at random from the set $\{0, 1, 2, \dots, n-1\}$. It computes $x = g^a$ and outputs $vk = x$ and $sk = a$. The message set associated to vk is $\mathcal{P}$.

- The *signing* algorithm $\mathcal{S}$ takes as input a signing key a and a message $m \in \mathcal{P}$. It samples a number r uniformly at random from the set $\{0, 1, 2, \dots, n-1\}$. It computes $\alpha = g^r$, $\beta = h(x, \alpha, m)$ and $\gamma = r - \beta a \bmod n$. It outputs the signature (β, γ).

- The *verification* algorithm $\mathcal{V}$ takes as input a verification key x, a message $m \in \mathcal{P}$ and a signature (β, γ). It outputs 1 if

$$h(x, g^{\gamma} x^{\beta}, m) \overset{?}{=} \beta,$$

  otherwise 0.

Exercise 19. The above is an informal description of a signature scheme. Write down carefully what the three algorithms $\mathcal{K}$, $\mathcal{S}$ and $\mathcal{V}$ are. Show that the triple $(\mathcal{K}, \mathcal{S}, \mathcal{V})$ is a signature scheme.

The Schnorr signatures and related schemes are famously sensitive to random number generation, as the following exercise show.

Exercise 20. Let x be a verification key for the Schnorr signature scheme. Suppose that under this verification key, (β, γ) is a valid signature on m, and (β', γ') is a valid signature on m', $m \neq m'$. Suppose further that

$$g^{\gamma} x^{\beta} = g^{\gamma'} x^{\beta'}.$$

Explain why the existence of these two signatures suggest a malfunction in random number generation, and show how to recover the signing key corresponding to x using only a handful of arithmetic operations.

5.3 The Digital Signature Algorithm

The Digital Signature Algorithm is a variant of the Schnorr signature scheme. We shall describe two variants of DSA, to show again how a hash function can improve both the practicality and security of a signature scheme.

Let $f : G \to \{0, 1, \dots, n-1\}$ be a "random-looking" hash function. Our first signature scheme $(\mathcal{K}, \mathcal{S}, \mathcal{V})$ works as follows.

- The *key generation* algorithm $\mathcal{K}$ is the same as for the Schnorr signature scheme.
- The *signing algorithm* $\mathcal{S}$ takes as input a signing key a and a message $m \in \{0, 1, \dots, n-1\}$. It samples a number r uniformly at random from the set $\{0, 1, \dots, n-1\}$. It computes $\beta = f(g^r)$ and

  $$\gamma = r^{-1}(m + \beta a) \bmod n.$$

  It outputs the signature (β, γ).
- The *verification algorithm* $\mathcal{V}$ takes as input a verification key x, a message $m \in \{0, 1, \dots, n-1\}$ and a signature (β, γ). It outputs 1 if

  $$f(g^{m\gamma^{-1}} x^{\beta\gamma^{-1}}) \overset{?}{=} \beta,$$

  otherwise 0. (Note that the exponent arithmetic happens modulo n.)

Exercise 21. The above is an informal description of a signature scheme. Write down carefully what the three algorithms $\mathcal{K}$, $\mathcal{S}$ and $\mathcal{V}$ are. Show that the triple $(\mathcal{K}, \mathcal{S}, \mathcal{V})$ is a signature scheme.

Exercise 22. Show how you can create a forgery for a random message for this scheme.

Hint: Choose u and v. Compute $\beta = f(g^u x^v)$. Now solve $v \equiv \beta\gamma^{-1} \pmod{n}$ and $u \equiv m\gamma^{-1} \pmod{n}$.

Exercise 23. Modify the above scheme to accept messages from S, by using a hash function $h : S \to \{0, 1, \dots, n-1\}$. Suppose h is one-way and collision resistant. Explain how this stops the attack from Exercise 22.

6 Securing Diffie-Hellman

As we have seen, the Diffie-Hellman protocol is subject to a man-in-the-middle attack, where Eve essentially runs the Diffie-Hellman protocol separately with Alice and Bob. Since Alice and Bob cannot distinguish each other's bits from Eve's bits, they will be cheated.

Signatures are one tool Alice and Bob can use to protect their Diffie-Hellman key exchange. In this case, Alice has a signing key pair (sk_A, vk_A) and Bob has a signing key pair (sk_B, vk_B), and they both know the other's verification key.

1. Alice chooses a number a uniformly at random from the set $\{0, 1, 2, \dots, n-1\}$. She computes $x = g^a$ and sends x to Bob.

2. Bob receives x from Alice. He chooses a number b uniformly at random from the set $\{0, 1, 2, \dots, n-1\}$ and computes $y = g^b$ and $z_B = x^b$. He computes $\sigma_B = \mathcal{S}(sk_B, m)$, where m is a message containing Alice' and Bob's names, that Alice initiated the key exchange, and x and y. Bob then sends y and σ_B to Alice.

3. Alice receives y and σ_B from Bob. Alice verifies σ_B and computes $z_A = y^a$. She also computes $\sigma_A = \mathcal{S}(sk_A, m')$, where m' is a message containing Alice' and Bob's names, that Alice initiated the key exchange, and x, y and σ_B. Alice then sends σ_B to Bob.

4. Bob receives σ_A from Alice. Bob verifies σ_A.

If either party's signature verification fails, that party stops immediately.

We note, without giving further details, that digital signatures can also be used with public key encryption to solve the problem of who sent a given ciphertext.

7 The Public Key Infrastructure Problem Revisited

Before asymmetric encryption was invented, a shared secret was required for secure communication over insecure channels. As we have seen, the Diffie-Hellman key exchange, public key encryption and digital signatures have removed the need for a preexisting shared secret, but public keys (for encryption or signature verification) still need to be exchanged before communicating.

A *public key infrastructure* is an infrastructure set up to move public keys from Alice to Bob in such a way that Bob can be sure that the public key he receives really belongs to Alice and is her current key, even if Alice and Bob have never communicated before.

As is often said, nothing will come of nothing, so Alice and Bob cannot hope to solve this problem on their own. One possible solution is the so-called *web of trust*. In this scheme, Alice and all her friends sign each other's public keys together with their unique names. Alice' public key, her name and her friend's signature is often called a *certificate*. The certificate is interpreted as Alice' friend saying that the public key belongs to the named person, namely Alice.

Alice' friends in turn sign their friends' public keys, and so forth. If we consider people as vertices in a graph, with edges between friends who have signed each other's public keys, Alice and Bob need to find a path between themselves in this graph.

A more practical system relies on a trusted third party, usually called a *certificate authority*. Again, the trusted third party signs Alice' public key along with her unique name. If Alice and Bob both trust each other's certificate authorities, they can simply send their certificate to the other party, and then use an appropriate public key protocol.

In practice, private keys are sometimes compromised, which means that Eve learns the key. When Alice discovers that someone knows her private key (and can thus impersonate her), Alice would like her certificate to stop working. She notifies her certificate authority that the certificate has been compromised.

The certificate authority was not involved when Alice and Bob communicated, so somehow Bob must be told that Alice' certificate has been revoked. The traditional approach is for the certificate authority to maintain a list of revoked certificates (a *certificate revocation list*). Anyone who relies on the certificate authority will periodically fetch an updated list of revoked certificates.

Since certificate revocation lists are fetched only periodically, there will typically be some time between Alice notifies her certificate authority until Bob stops accepting the certificate. Another problem with certificate revocation lists is that if there are many certificate authorities, managing the revocation lists becomes impractical.

One popular solution is for a certificate authority to provide a *certificate status service*. Any user may ask for the status of a given certificate. The certificate authority will reply with a signed message. If the certificate is valid (that is, not revoked), the message contains a statement to that effect and the current time. If the certificate has been revoked, the message contains a statement to that effect and the time of revocation.