Applied Information Security and Cryptography

Lab 04: Hash functions and digital signatures

In the first part of this lab, you will familiarize with the concept of cryptographic hash functions and see in what they are different from conventional hash functions. Then, you will employ different hash functions to implement a digital signature algorithm based on Schnorr scheme. This can be used to sign your messages, so that the receiving end can verify their provenance, or to authenticate a message you receive from some other party. However, you will verify that a careless implementation of the digital signature algorithm may allow an adversary to forge a signature.

You will need the Python “secrets” module to generate secure random numbers, which is available in the standard library from version 3.6. Alternatively, you can emulate secrets using calls to os.urandom(), as done in Lab 03.

Hash functions

Some cryptographic hash functions are already available in the Python standard library using the “hashlib” module. For example, you can obtain the SHA-256 digest of a message as:

```
m = hashlib.sha256()
m.update(message)
hash = m.digest()
```

In all the activities in this lab, message should be encoded as a bytes-like object (e.g., message = b’my message’) and the corresponding hash value is returned as a bytes-like object (in this case, 32 bytes, since SHA-256 produces a 256 bit digest). It is suggested that you wrap this code in a simple function that takes as input the message, the (optional) output length in bytes, and returns the message digest. To return a hash digest of nbytes, simply return the first nbytes of hash, i.e., hash[:nbytes]. For example, the full 32 bytes SHA-256 digest of a message should look like this:

```
message: b'SHA–256 is a cryptographic hash function'
SHA–256 digest:
b'\xd8\x17\xfax\x9b\xe6b9\xac\x88\x13B?\x05\xeb\xf7\xee\xe1iE\xe6\x8df\
xe3\xdc\x99\xd8\xb8\x1c\xbcA\x85'
```

We will verify that the complexity of finding random collisions in the output of a hash function is proportional to $2^{n/2}$, where n is the bit length of the hash digest. A naïve implementation of a collision search algorithm could be to generate $2^{n/2}$ hash values of random messages and check all possible pairs for equality. Obviously, this would require $O(2^{n/2})$ memory and $2^{\frac{n}{2}}(2^{\frac{n}{2}} - 1)/2 \approx 2^{n-1}$ comparisons, so it is not very efficient. A more efficient algorithm is described by the following pseudo code (Algorithm 5.9 in J. Katz, Y. Lindell, Introduction to Modern Cryptography):

```
Input: hash function H, hash value size in bits n
     # in the implementation you can also generate x0 to be one byte
more than the output of H, i.e., n+8 bits. The important thing is
that x0 should be longer than the output of H
    x0 = n+1 random bits
    x1 = H(x0)
    x2 = H(x1)
    while x1 != x2:
        x1 = H(x1)
        x2 = H(H(x2))
    x1 = x0
    while H(x1) != H(x2):
        x1 = H(x1)
        x2 = H(x2)
Output: x1, x2
```

The rationale of the above algorithm is that, if you model the output of the hash function as a random value, then the series $s[1], s[2], s[3], \ldots = H(x0), H(H(x0)), H(H(H(x0))), \ldots$, is a series of random n-bit values until you find a repetition. After that point, the sequence will periodically repeat the same values, i.e., if the first repetition is $s[i] = s[i+D]$, then $s[j] = s[j+kD]$ for every $k \geq 0$ and $j \geq i$. Let's find $x = yD$ such that $i \leq x \leq i+D$. It is immediate to verify $s[x] = s[x+yD] = s[2x]$. Hence, the first loop will terminate after at most $i+D$ steps. On average, you expect a repetition after $O(2^{n/2})$ values, so that $i+D = O(2^{n/2})$. The second loop simply starts from $s[1] = H(x0)$ and $s[yD+1] = H(s[yD])$ and will exit at step i when $s[i] = s[i+yD]$. At that point $s[i-1]$ and $s[i+yD-1]$ are a collision for the hash function H. So, on average, you expect the above algorithm to execute $O(2^{n/2})$ steps before finding a collision.

Implement the algorithm in Python. For simplicity, you can consider hash functions that takes as input an integer number of bytes and produce as output an integer number of bytes. The seed x0 should be a bytearray of random bytes strictly longer that the output size of the hash function, i.e., if the hash digest is 32 bytes you should generate at least 33 random bytes (this is to ensure that in the worst case the first loop will exit after 2^n steps).

In your implementation, count the number of times that the two while loops are executed, let's call these values C1 and C2. Then, test the algorithm with hash functions of different output length. For this, you can simply take the message digest of SHA-256 and consider only the first m bytes. Record the value of C1 and C2 when the output length of the hash function is 1 byte (8 bits), 2 bytes (16 bits), 3 bytes (24 bits), and 4 bytes (32 bits). You can repeat this experiment 10 times by randomly choosing x0. You should see that C1 and C2, on average, are on the order of $2^{n/2}$, where n is the output length of the hash function in bits.

Now, let's consider a universal hash function. In short, a universal hash function has the property that the output of the function is uniformly distributed over the possible 2^n values. A simple way of implementing a universal hash function is to represent the message as a big integer number m, then compute the value

$$h = am + b \bmod q$$

where q is a prime number and a and b are two constants smaller than q. The hash is obtained by representing h as a bytearray and taking the first bytes, according the desired length of the hash output.

Implement the above universal hash function in Python. For converting a message, expressed as a bytes-like object, to a big integer use:

```
m = int.from_bytes(message, byteorder='big')
```

For the constants q, a, b, use the values of qDSA, aU, bU that you find in the script AISC_04.py. For converting the resulting h to a bytes-like object, use:

```
hash = h.to_bytes(20, byteorder='big')
```

This is because qDSA, hence the value h, can be represented over 20 bytes. If you want a nbytes hash digest, simply return hash[:nbytes]. To test your function, the following is a 20-bytes digest obtained with the universal hash function:

```
message: b'SHA-256 is a cryptographic hash function'
universal hash digest:
b'\x03r\x86\xab?e\xa8\xe3\xa8\xdb|+T\xfeH\t\xd8\x80\x02\x84'
```

Test the collision finding algorithm with the universal hash function, using hash digest length of 1,2,3, and 4 bytes, and record the values of C1 and C2 that you obtain in this case. The values should be similar to those observed using SHA-256.

However, a universal hash function in general is not a cryptographic hash function. To prove this, write a simple script that, given a 20 bytes hash digest produced by the previous universal hash function, finds a preimage of that digest. (Hint: given h, it should not be too difficult to invert $h = am + b \bmod q$ for fixed values of a, b, and q. If you need a division, remember the eGCD algorithm of Lab 03.) To verify your function, simply compute the hash digest of the preimage and very that it is equal to the hash digest that you provided as input.

Digital signatures

In this second part, you will implement a Schnorr signature scheme based on the hash functions developed in the first part. The public domain parameters of the signature scheme will be p=pDSA, q=qDSA, and g=gDSA, as reported in the script AISC_04.py.

To generate your key pair, simply generate a random number x between 1 and $q-1$. This can be done securely using secrets.randbelow in the secrets module. This is your private key. The public key will be $y = g^x \bmod p$. You can publish y so that everybody can verify your signatures.

Given the private key x, the Schnorr signature of a message can be computed as follows:

- Generate a random k between 1 and $q - 1$
- Compute $I = g^k \bmod p$
- Compute $r = H(I \mid message) \bmod q$
- Compute $s = k - rx \bmod q$
- Output (r, s)

Given the public key y, the Schnorr signature (r, s) of a message can be verified as follows:

- Compute $I = g^s\, y^r \bmod p$
- Compute $r' = H(I\, |message)$
- Output $r' == r$

Write two Python functions computing the Schnorr signature and signature verification, where H is SHA-256. If you need to convert an integer I to a bytearray use:

```
I.to_bytes(nbytes, byteorder='big')
```

To ensure that I is encoded on a fixed number of bytes, use the length of the public domain parameter p (I is computed modulo p, so it never exceeds this value):

```
nbytes = (p.bit_length()+7)//8
```

If you need to convert a bytearray rbytes to an integer use:

```
r = int.from_bytes(rbytes, byteorder='big')
```

To ensure compatibility, compute the full output of SHA-256 on 32 bytes, convert the bytes to an integer, and compute the reduction modulo q. The concatenation of I and message should be obtained by appending message to the byte representation of I (in Python, this is simply I_bytes + message).

Check that you are able to sign a message and verify the signature with the provided functions. Check that even a slight modification of the message invalidates the signature.

Publish a message on the public chat with your signature. If you published your public key (and if everybody followed the specifications), everybody should be able to verify that the message is authentic. Moreover, if you keep your private key secret, since the public parameters are safe, discrete logarithm is hard related to them, and SHA-256 is collision resistant, only you will be able to generate a valid signature on any message.

However, even in this case Schnorr signatures can be forged if not properly implemented. Let us assume that an implementation of the signature algorithm generates the random k value by simply incrementing by one the previously used value. That is, if a signature uses k, the next signature uses $k + 1$. This guarantees that k is always different. However, this does not guarantee that k is unpredictable!

The instructor will publish two signatures obtained in this way, together with his public key. You do not know if they are consecutive signatures, however you can assume that they have been computed in a short time interval, let's say no more than 100 signatures apart. You should derive the instructor's private key, sign a message of your choice, and publish it on the chat to demonstrate that you broke the signature scheme.

Bonus task: Another vulnerability of a signature algorithm can be a cryptographically weak hash function. Modify the signature and verification functions so as to use the universal hash function. To ensure compatibility, compute the full output on 20 bytes. The functionality should be the same as before. However, since the hash function is not collision resistant, now an adversary can forge a signature without the private key, if another valid signature is available. Namely, the universal hash function that we implemented has an interesting property: it is possible to compute a preimage in which all the bytes, except the last 20, are arbitrarily chosen. Use this property to forge a signature for a message chosen by you (of course, you will not have control over the last 20 bytes, so you will have to leave room for them). The forgery should be based only on the public key and an existing valid signature.

Questions (Answer these in your report)

Report the values of C1 and C2 that you obtained for the different lengths of the hash digest, for both SHA-256 and the universal hash function. Are these values consistent with the length of the hash digest? Please motivate your answer.

Consider the following statements: 1) The output of a cryptographic hash function is uniformly distributed over the possible values; 2) If the output of a hash function is uniformly distributed over the possible values then it is a cryptographic hash function. Discuss whether they are true or false, possibly referring to the results of this lab.

Produce a collision for the universal hash function, 20-bytes output.

If the instructor asked you to publish a collision for SHA-256, for a hash digest of 32 bytes, would you consider this a feasible assignment? Please motivate your answer.

Report your public key for the Schnorr signature scheme and a signed message that can be verified with that key.

Describe how did you derive the instructor's private key from the two signed messages.

Bonus question: in an implementation of the digital signature algorithm, the value k is obtained by incrementing a counter and encrypting it with AES. The key for AES is derived from the private key (e.g., by hashing it). Is this implementation secure? Please motivate your answer.

Bonus task: produce an example of a forged signature based on the universal hash function (a valid signature and the corresponding forged signature). Describe how did you forge the signature.