COL106 : 2021-22 (Semester I)
Project: Module 1 *

Satwik Banchhor Venkata Koppula Sarang Sunilchaudhari

August 23, 2021

Notations

Let α be any character, and ℓ a positive integer. Then α^ℓ is a string of ℓ characters, each character being α. For any function $f : \mathcal{D} \to \mathcal{R}$ such that $\mathcal{R} \subseteq \mathcal{D}$, and natural number ℓ, $f^{(\ell)}$ denotes the function f applied ℓ times (for instance, $f^{(3)}(x) = f(f(f(x)))$).

Important: The question marked with a ♠ is the *lab-submission* problem. It is to be submitted via Moodle by 11:59PM on the day that you have your lab.

1 Introduction

In this document, we will see the simple cryptographic tools used for building a cryptocurrency. First, let us briefly discuss what is *cryptography.* It started, many centuries ago, as an art for hiding secrets. Today, cryptography is much more than just hiding secrets. Over the last fifty years, cryptography has developed into a science that is integral for ensuring that various parties, communicating over the digital medium, behave honestly. We will see two very popular tools from cryptography — *collision resistant functions* (CRFs) and *digital signatures.* Both are very simple and natural tools, and we use them everyday!

1.1 Collision Resistant Function (CRF)

A collision resistant function (CRF) is a compressing function that maps the input file/string to a short string (say 64 characters long). The important property is that it is collision resistant — even though the function description is public and known to everyone, it is very hard to find two inputs x, y such that the function maps them to the same string. Such a pair of strings is referred to as a *collision* for the function. Note that since the function is compressing (the function maps strings of unbounded size to strings of a fixed output size), we know that there certainly exist many collisions, but still, it is hard to find them (and hence they are called *collision resistant functions*)!

An application of CRFs: Why is such a function useful? Consider the following scenario : there is an exciting new software that you would like to use. Luckily, your friend has this software, and he gives it to you (on a USB/DVD). How can you be sure that this is indeed the 'correct' software, and not some malicious file instead?

Here is an idea: the software company will apply the collision resistant function on the legitimate software, and publish the short output string on their website. You can compute the collision resistant function on the file that your friend gave, and check that the output matches the one on the website.

*Author names listed in alphabetical order. Many thanks to Chirag Bansal, Kritika Gupta, Pratik Kedia, Akshay Mattoo, Bikram Mondal and Suraj Patni for help with this module.

F.O.F.Y.

Why can you be sure that if the CRF output matches the string on the software website, then this is indeed the correct file?

How to build a CRF Building a collision resistant function is an extremely challenging task. Consider any *natural* function that maps a domain $\mathcal{D}$ to a smaller co-domain $\mathcal{R}$. Examples:

- Take any integer p and consider the mod_p function : this maps all integers to the set $\{0, 1, \ldots, p-1\}$ via the operation $\mathsf{mod}_p(x) = x \mod p$. This function maps a large domain (the set of all integers) to a smaller co-domain (the set of all integers in $\{0, 1, \ldots, p-1\}$). Given p and any integer z, can you find another integer z' such that (z, z') form a collision?
- Let us consider a more complicated function called ExpMod_p : this again maps all integers to the set $\{0, 1, \ldots, p-1\}$ via the operation $\mathsf{ExpMod}_p(x) = 2^x \mod p$. Given p and any integer z, can you find another integer z' such that (z, z') form a collision for ExpMod_p?

Moral of the story : building CRFs is hard!

Open Question

Construct a function mapping a large domain to a smaller co-domain that is potentially collision resistant.

From now on, let us fix the domain to be the set of all strings, and the co-domain to be all hexadecimal strings[1] of a fixed output length. Therefore, the size of the co-domain is $16^{\text{output-length}}$.

Luckily, we do have some functions that *seem* to be collision resistant. As part of this assignment, you are given an implementation of such a collision-resistant function. Fix any output length `outputsize`[2], and the function maps any input string x to a hexadecimal output of length `outputsize`. Moreover, this function is very efficient – the time complexity of this function is $O(|x|)$.

We will explore several properties of such functions.

- First, we will see a *generic inefficient* algorithm for finding collisions. By generic, we mean that this algorithm will work for *any* function that maps a domain to a smaller co-domain. By inefficient, we mean that the running time of the algorithm will be exponential in the size of output length. In particular, you can use this algorithm to find a collision if `outputsize` is small, but if `outputsize` is large, then the algorithm will take a long time.
  The algorithm takes no inputs, and it must output a collision (z, z') for the function `Fn`. Let us start with the simplest algorithm. Start with any fixed string, say $y_0 = 0^{\texttt{outputsize}+1}$ (this denotes a string of `outputsize` + 1 zeroes). For each i, set $y_{i+1} = \texttt{Fn}(y_i)$, and keep doing this until you find a collision. You can check that this process will output a collision in at most $16^{\texttt{outputsize}} + 1$ steps. [3]
- Actually, there exists a simple *randomized* algorithm that is faster than the one described above. This randomized algorithm runs in time $O(16^{0.5*\texttt{outputsize}})$. See Exercise 2.1 for more details.

[1] A hexadecimal string is a string whose characters are either digits $0, 1, \ldots, 9$, or alphabets $A, B, \ldots, F$.

[2] Our implementation supports `outputsize` in the range $[1, 64]$.

[3] Let $\mathcal{R}$ be the co-domain of the function. Why can you be sure that evaluating the function for $|\mathcal{R}| + 1$ distinct inputs will always produce a collision?
Additional reading: Pigeonhole Principle

F.O.F.Y.

In a real world CRF with `outputsize`=64, how much time would the simplest algorithm take to find any collision? What about the randomized algorithm? You can assume that your laptop can perform 10^{10} computations per second.

1.2 Digital Signatures

Next, we move on to digital signatures. A digital signature is just the digital analogue of a real-world signature. We sign documents, and if our signature is sufficiently complicated, then an adversary cannot forge the signature, even if the adversary has seen your signature in the past.

A digital signature scheme achieves this goal for digital documents. Such a scheme consists of three algorithms.

- First, there is a key generation algorithm. When you run this algorithm, it produces a *signing key*, and a *verification key*. The signing key is what you will use to sign documents, and therefore must be kept secret. The verification key can be used by anyone to verify your signature.
- Next, we have a signing algorithm that can be used to sign a document using a signing key.
- Finally, we have a verification algorithm that uses the verification key to verify a signature on a document.

The key generation algorithm needs to be run only once (to generate the signing/verification keys). After the keys are generated, you can use the same signing key to sign multiple documents. As you might expect, if you sign a document with a signing key, and then verify this signature with the corresponding verification key, then this verification must pass. This is referred to as the *correctness* of the scheme.[4]

For security, we have the requirement that no adversary can produce a valid signature with respect to a verification key `vk` [5] on a document without having the corresponding signing key `sk`.

How to build a signature scheme There are two steps in building a signature scheme :

- First, build a signature scheme that can sign bounded-length documents.
- Next, use the bounded-message signature scheme, together with a collision-resistant function, to build a signature scheme that can handle arbitrary documents.

The first task (building a secure signature scheme for bounded-length messages) is beyond the scope of this course (it will require a few lectures to describe even the simplest signature scheme). However, in this project, you are given an implementation of the following methods:

- `Keygen`: This function takes no inputs, and outputs a signing key and a verification key. [6]
- `BoundedMsgSign`: This function takes as input a signing key (which is a string) and a bounded-length message (which is a 64-character hexadecimal string). It outputs a signature (which is a string).
- `BoundedMsgVerify`: This function takes as input a verification key (which is a string), a bounded-length message (which is a 64-character hexadecimal string) and a signature (which is a string). It outputs a boolean value.

[4] Note: the signing algorithm could be randomized, and therefore correctness does not imply that the signing algorithm will output the same signature when signing a document m using signing key `sk`. If you compute multiple signatures on m using `sk`, these signatures could be different, but all of them will be accepted by the verification algorithm when using the corresponding verification key `vk`.

[5] That is, a signature that will be accepted by the verification algorithm using `vk`

[6] Strictly speaking, the `Keygen` method outputs a `SignatureKeys` object, and each `SignatureKeys` object consists of two attributes – a signing key and a verification key.

In this course, we only need to use these algorithms (and not worry about how they are implemented). Note that the implementations of signing and verification algorithms above can only handle bounded length messages. The first exercise is to handle unbounded length messages, and this is the subject of Exercise 2.2. The idea is simple: to sign a long message `m`, we first compute a 64 character 'digest' `dgst` of `m`, using a collision-resistant function. Next, we compute a signature on `dgst` using the `BoundedMsgSign` algorithm. The signature on `m` is simply a signature on `dgst`. First, you must implement this algorithm. Next, we need to verify signatures on large messages. You must first design an appropriate verification algorithm, and then implement it.

2 Assignment Questions

You must solve all the problems below, preferably before your lab session next week. The lab evaluation questions will build upon these assignment questions, and therefore it is strongly recommended that you try these questions beforehand.

You are given the following classes:

- `public class CRF`: This class has the following attributes and functions.

  Attributes:

  - `public int outputsize`: denotes the output size of the function, following the constraint $1 \leq$ `outputsize` ≤ 64.

  Predefined functions:

  - `public String Fn(String` s`)`: takes as input a string s, and outputs a hexadecimal string s', which is `outputsize` characters long.

  To be implemented:

  - `public Pair<String,String> FindCollDeterministic()`: takes no input, and outputs a pair of strings (s, s') which is a collision to the function `Fn`.
  - `public void FindCollRandomized()`: takes no input, evaluates the function CRF `Fn` on $n = 1000 \cdot |\mathcal{R}|^{0.5}$ random strings and does the following:
    - in a file named *"FindCollRandomizedAttempts.txt"*, it outputs all the random strings (each string in a separate line)
    - In another file named *"FindCollRandomizedOutcome.txt"* the function outputs `"NOT FOUND"` in the first line if no collision was found; if a collision (s, s') is found, then it outputs `"FOUND"` in the first line followed by the string s in the second line, and s' in the third line.

- `public class SignatureKeys`: This class has the following attributes and functions.

  Attributes:

  - `public String sk`: String representing the signing key.
  - `public String vk`: String representing the verification key.

- `public class Signature`: This class has the following attributes and functions.

  Predefined functions:

  - `public static SignatureKeys Keygen()`: takes no input and outputs a `SignatureKeys` object.
  - `public static String BoundedMsgSign(String m, String sk)`: takes as input a 64 character long string `m` (which is the message to be signed), a string `sk` (which is the signing key), and outputs a string `sig` (which is the signature).

- public static boolean BoundedMsgVerify(String m, String vk, String sig): takes as input a 64 character long message string m, a string vk (which is the verification key) and a string sig (which is the signature). It either accepts this signature (in which case it outputs True), or it rejects the signature (in which case it outputs False).

To be implemented:

- public static String Sign(String m, String sk): this function is an extension of the function BoundedMsgSign, which can be used to sign an arbitrarily long message m with the signing key sk.
- public static boolean Verify(String m, String vk, String sig): This function is an extension of the function BoundedMsgVerify, which is used to verify the signature sig on an arbitrarily long message m using the verification key vk.

2.1 Exercises

In the following exercises you will be implementing the functions that are marked as "To be implemented" in the above classes.

Exercise 2.1. ***Collision finding algorithm*** *In this exercise you need to find collisions to the CRF* Fn*.*

(a) Implementing FindCollDeterministic*: Find a collision by starting with a fixed string (hard-coded in your program), and then repeatedly evaluating the CRF until a collision is found.*

(b) Implementing FindCollRandomized*: Find a collision by evaluating the function* Fn *on enough random points. How many random points? Suppose the function's co-domain is* $\mathcal{R}$ *= set of all hexadecimal strings of length* outputsize*. Then evaluating the function on* $|\mathcal{R}| + 1$ *distinct inputs will produce a collision.*

Interestingly, if the function is evaluated on $O(|\mathcal{R}|^{0.5})$ *random points, even then we have a good chance of finding a collision.*[7]

What is the time and space complexity of your algorithm? You can assume that computing the CRF on input string x *takes* $O(|x|)$ *time and space.*

Note: We expect your implementation of FindCollDeterministic *and* FindCollRandomized *to run for values of* outputsize *till 5 or 6.*

Exercise 2.2. ♠ ***Sign long messages and Verify signatures***

For this exercise you need to use the predefined functions BoundedMsgSign *and* BoundedMsgVerify *to sign and verify arbitrarily long messages.*

(a) Implementing Sign*: Construct a signature for an arbitrarily long message. For constructing the signature, you need to first compress the long input message by applying the appropriate CRF i.e. using a CRF instance with* outputsize *= 64, and then sign the CRF output using the function* BoundedMsgSign*.*

(b) Implementing Verify*: Verify the signature constructed using your implementation of* Sign *for an arbitrarily long message.*

Exercise 2.3. ***Optional Questions*** *These problems will not be graded.*

- ***Optional Question 1****: Find a collision in* $O(|\mathcal{R}|)$ *time and* $O(1)$ *space (assuming* outputsize *is a constant).*

  Hint: Additional Reading

[7] Why does this work? You can read about it here.

- ***Optional Question 2:*** *Find a collision in $O(|\mathcal{R}|^{0.5})$ time and $O(1)$ space (assuming* `outputsize` *is a constant).*

  Hint: start with a random string `seed`*, let $y_i = \texttt{Fn}^{(i)}(\texttt{seed})$, and let $i_{\texttt{seed}}$ be the smallest i such that $\texttt{Fn}^{(i)}(\texttt{seed}) = \texttt{Fn}^{(2i)}(\texttt{seed})$. First, find this $i_{\texttt{seed}}$, then use $i_{\texttt{seed}}$ to find a collision. Show that with probability at least $1/2$, $i_{\texttt{seed}} \leq 1000 \cdot |\mathcal{R}|^{0.5}$. Use this to find an $O(|\mathcal{R}|^{0.5})$ time, $O(1)$ space algorithm.*

2.2 Instructions

- Please go through the lab sheet **before** your lab starts. The lab test problem will be related to one of the above questions (that is, Exercises 2.1 and 2.2), and therefore it would be helpful to familiarize yourself with CRFs and digital signatures before the lab starts.

- During the lab, the TA will briefly discuss Exercise 2.1. Exercise 2.2 is your lab-submission problem for this week.

- The supporting code has the following important files:

  - `DemoClass.java` - use this to run the demo code. This demo code contains example syntax for initializing/using the CRF function, as well the signing/verification methods.
  - `CRF.java` - modify this file to implement the CRF related methods.
  - `Signature.java` - modify this file to implement the signatures related method.

    You must submit this file on Moodle as part of your lab-submission by 11:59PM on your lab day.

- Grading methodology for lab-submission problem: You must implement the `Sign` and `Verify` methods of `Signature` class. We will run separate test cases for the signing and verification methods.

  - For testing the signing method, we will choose a signing key sk and the corresponding verification key vk. We will run your `Sign` method on multiple inputs, each input of the form (m, sk). The output of your method will be verified using the verification key vk. Note that we will be using our own implementation of `Verify` here.
  - For testing the verification method, we will choose a signing key sk and the corresponding verification key vk. We will run your `Verify` method on multiple inputs, each input will be of the form (m, vk, s) where s is either a random string (in which case your verification must output False), or s is a signature on m using sk (in which case your verification must output True). Note that we will be using our own implementation of `Sign` here.

- Sample input/output can be found in the supporting code. However, note that some of the methods (such as `FindCollRandomized` and `Sign`) are randomized algorithms, and therefore your output may not match the sample output.

- See the README file (included in supporting code) for more information regarding test cases.

Acknowledgements Thanks to Aniruddha Deb for finding a mistake in an earlier version of this document.