Homework 2 for 02231, 2025 (10 points)

Due 3.11.2025, **08:00**

Notation. For vectors we write $\vec{x} \in \{0,1\}^\lambda$ while we use x for strings. By $\vec{x}[i]$ we denote the ith element of $\vec{x}$, where $i \in \{1, \ldots, \lambda\}$, and use the same notation for indexing bits of a string. We denote the substring of $x \in \{0,1\}^\lambda$ that starts with bit i and ends with bit $j \geq i$ as $x[i;j]$. As in the lecture, we write $k \leftarrow K$ if k is sampled from the finite set K such that it can be each element from K with equal probability $1/|K|$.

For two strings x, y we use $x \| y$ to denote the string obtained from concatenating x with y. For $x \in \{0,1\}^*$ we let $\textsc{len}(x)$ be the length of the string x in number of bits. For $x \in \mathbb{R}$ (i.e. a real number x) we let $\lceil x \rceil$ be the function that rounds x up to the next integer y. If x is already an integer then output x, otherwise output the next integer larger than x. For example, $\lceil 1 \rceil = 1, \lceil 0.4 \rceil = 1$, and $\lceil 3.000001 \rceil = 4$. Equally, we write $\lfloor x \rfloor$ for the function that rounds x down to the next integer. For example, $\lfloor 3 \rfloor = \lfloor 3.1 \rfloor = \lfloor 3.99999 \rfloor = 3$. We write $\textsc{zeros}(k)$ to denote a string of 0s of length k (which is empty if $k = 0$).

Exercise 2.1. (How to hash long inputs, revisited - 5 points)
In the lecture, you learned how the Merkle-Damgård construction can be used to hash inputs of arbitrary length given a collision-resistant compression function $H' : \{0,1\}^{t+\lambda} \to \{0,1\}^\lambda$. In this exercise, you will look at another very popular construction to hash long messages, namely so-called Merkle trees.

The construction works as follows: Let $H : \{0,1\}^{3\lambda} \to \{0,1\}^\lambda$ be a collision-resistant hash function[1]. Then, to hash a message $m \in \{0,1\}^*$ using salt $s \in \{0,1\}^\lambda$ we define $\textsc{MerkleTree}(m, s)$ recursively as follows:

1. If $\textsc{len}(m) \leq \lambda$ then output m.
2. Let $\ell = \lceil \log_2(\textsc{len}(m)/\lambda) \rceil$
3. Define $m' = m \| \textsc{zeros}(2^\ell \cdot \lambda - \textsc{len}(m))$
4. For $i \in \{0, \ldots, 2^{\ell-1} - 1\}$:
   (a) Set $h_i \leftarrow H(s \| m'[2 \cdot i \cdot \lambda + 1; 2 \cdot i \cdot \lambda + 2\lambda])$

[1] We note that the algorithm that we present below is not collision-resistant as we do not employ a suitable padding scheme.

5. Return $\textsc{MerkleTree}(h_0\| \cdots \|h_{2^{\ell-1}-1}, s)$

In this exercise, you are asked to answer the following questions:

1. Let $\lambda = 4$. Assume that m_1 has length 1 bit, m_2 has length 4 bits, m_3 has length 5 bits, m_4 has length 8 bits and m_5 has length 16 bits. Write down the output of MerkleTree for all of these 5 inputs, in terms of calls to H (for some fixed value s). That is, can you split up each m_i into substrings, so that H applied to these substrings (and outputs of H on these substrings) is equivalent to $\textsc{MerkleTree}(m_i, s)$?

2. In some applications of hash functions, a protocol participant A wants to prove to another protocol participant B that for a message $m = m_1|\ldots|m_t$ of length $t \cdot \lambda$ that hashes to h, the ith block $m_i \in \{0,1\}^\lambda$ equals x.[2] This can of course be done if A simply reveals m to B, who checks if it hashes to h and looks that the ith block of m has the right value. However, A then has to send $t \cdot \lambda$ bits of information.

   Describe a more efficient mechanism that reveals "intermediate" values generated when computing $\textsc{MerkleTree}(m, s)$ instead of m, and describe the total number of bits sent by your more efficient mechanism in terms of $\textsc{len}(m)$ and λ. To see how this shorter "proof" could work, imagine A has a string m of length 8λ bits, consisting of substrings $m_1, \ldots, m_4$ of length 2λ bits each. Which additional information (generated during $\textsc{MerkleTree}(m_1|\ldots|m_4, s)$) in addition to m_4 would A have to send to B to reveal the last λ bits of m_4? And how does B verify that this information was correct?

3. Implement the algorithm $\textsc{MerkleTree}(m, s)$ described above, allowing you to hash messages of arbitrary length. Use $SHAKE - 256$ as the function H for $\lambda = 256$. Describe in the handin what general approach your code follows.

You should upload a .zip-file containing the code which you wrote for this exercise (in addition to the handin), together with a README file. Your code must compile on a machine running Ubuntu 24.04.2 LTS. Please include the following two sections in the README:

Compilation and Installation: What are the prerequisites (installed libraries and versions etc.) to compile and run your code, which commands should be used to compile the code, how is the code installed?

[2]Of course every hash value likely has many pre-images under a hash function mapping long inputs to short hashes, so this is more about proving to somebody that *you know* a preimage where the ith block is m_i

Running the tests: Describe which commands should be used to run tests that show that your functions work. How should the outputs of the tests be interpreted by a reader?

Please test if the instructions in the README work in the described environment before handing in. Non-functioning code will impact how many points you obtain.

Exercise 2.2. (RSA - 5 points)
Remember that the RSA cryptosystem consists of the following 3 algorithms:

RSA.Keygen Generate two random distinct, odd primes p, q of length at least λ bits. Let $N = p \cdot q$, choose $e \in \mathbb{Z}^*_{\varphi(N)}$ uniformly at random and compute $d = e^{-1} \bmod \varphi(N)$. Then output $sk = (N, d)$ and $pk = (N, e)$.

RSA.Enc On input $pk = (N, e)$ and $m \in \mathbb{Z}_N$ output $c = m^e \bmod N$,

RSA.Dec On input $sk = (N, d)$ and $c \in \mathbb{Z}_N$ output $m = c^d \bmod N$.

1. Consider the following function $f : \{0,1\}^\lambda \to \mathbb{Z}_N$ which computes

$$f : \vec{x} \mapsto \sum_{i=0}^{\lambda-1} \vec{x}[i] 2^i.$$

f essentially converts the bit decomposition of an element from a subset of $\mathbb{Z}_N$ into the element in $\mathbb{Z}_N$. We define the following modified encryption and decryption functions:

RSA.Enc′ On input $pk = (N, e)$ and $m \in \{0,1\}^\lambda$ output $c = f(m)^e \bmod N$,

RSA.Dec′ On input $sk = (N, d)$ and $c \in \mathbb{Z}_N$ compute $\overline{m} = c^d \bmod N$. If $0 \leq \overline{m} < 2^\lambda$ then output the bit decomposition of $\overline{m}$, i.e. f^{-1}. Otherwise output $\perp$.

Let c be a ciphertext generated by Enc′. Describe under which condition on m the algorithm Dec′$(sk, 2 \cdot c \bmod N)$ outputs $\perp$. Also describe why this does not impact the correctness of the public-key cryptosystem Keygen, Enc′, Dec′ if the plaintext space is $m \in \{0,1\}^\lambda$.

2. Let SKE = (Keygen, Enc, Dec) be the One-time Pad Symmetric Key encryption scheme with Plaintext-, Ciphertext- and Keyspace $\{0,1\}^\lambda$. We define the following *hybrid* cryptosystem HE:

HE.Keygen Run and output $(sk, pk) \leftarrow \text{RSA.Keygen}()$.

HE.Enc On input $m \in \{0,1\}^\lambda$ compute $k \leftarrow \text{SKE.Keygen}()$, $c_1 \leftarrow \text{RSA.Enc}'(pk, k)$ and $c_2 \leftarrow \text{SKE.Enc}(k, m)$ and output $c = (c_1, c_2)$.

HE.Dec On input $c = (c_1, c_2)$ compute $k \leftarrow \text{RSA.Dec}'(sk, c_1)$. If $k \neq \perp$ then output $m = \text{SKE.Dec}(k, c_2)$, otherwise output $\perp$.

Show that HE is a perfectly correct public-key encryption scheme.

3. Given the observations from part 1 of this exercise, can you construct an attacker that shows that HE is not IND-CCA secure? Your attacker should only make one query to the decryption oracle. What is the success probability of your attacker?

4. Assume we replace RSA in HE with any IND-CCA-secure public key encryption scheme with message space $\{0,1\}^\lambda$ and SKE with any IND-CCA-secure symmetric key encryption scheme with key space $\{0,1\}^\lambda$. Show that the resulting encryption scheme is IND-CPA secure. *Bonus: Is the resulting scheme IND-CCA secure? If so, why?*

What you should do

- Enroll into one of the homework submission groups. You are encouraged to work in groups of (up to) 3, so the groups have capacity 3.
- Write the solutions to the exercises in one document and upload it on Learn.
- **Please refrain from using Generative AI when solving the exercises.** You won't learn anything if you don't do the exercise yourself, will likely do worse at the exam and you might embarrass yourself and your group if you have the exact same solution down to examples as 6 other groups.
- The format of your document should be PDF, together with a separate ZIP file containing any program code that you created as part of the exercises. We recommend that you use the following naming scheme for your submission:
  1. HW_{number}_Group_{group number}_handin.pdf
  2. HW_{number}_Group_{group number}_code.zip

- Please do not submit ZIP files that contain other ZIP files.
- Please do not submit your virtual environments in the ZIP folder, only the instructions for how to run the code and the code itself.
- For any program code, please also describe your solution in the PDF so that it can be understood without looking at all the details of your code.
- The PDF document should contain your group number as part of the title. Each submitted code file should contain your group number in the beginning as a comment.