TP Advanced Cryptography 2024

Léo Colisson Palais

Motivations

The goal of this TP is to code a full system to convert an arbitrary circuit C (made from NOT, AND, OR and XOR gates) into a non-interactive ZK proof, to prove for instance that we know x such that $C(x) = y$ for some publicly known y. The high-level picture we will follow is the one described in the course:

1. first, turn the circuit into an equivalent satisfiable SAT instance s,
2. then turn the satisfiable SAT s instance into a graph g with a Hamiltonian path p,
3. finally prove in ZK that there exists a Hamiltonian path p to g without revealing anything about g. We will implement here an improvement to represent sparse graphs more efficiently.

I highly recommend to use python (or sage) as a programming language since I'll provide some tests in python, provided in file `tp_01_tests.py` that you should place next to your code, started in the template `tp_01.py`.

Notations. In the following, unless otherwise specified, `n` represents the number of edges in the graph (clear from the context), `e` the number of edges, `N` the number of bytes needed to encode `n` (i.e. `N = math.ceil(math.log(N, 256))`), `E` the number of bytes needed to encode `e`, `R = math.ceil(options.secpar/8)` will represent the number of bytes of the randomness used for the commitment. We often omit the type of the functions inputs when the input name is clear (e.g. `g` represents typically a graph, `path` a list of integers, `i` is an integer etc).

Part 1: Hamiltonian paths and ZK

We will solve these tasks in reverse order. Note that in order to efficiently store and send the ZK proofs, we will encode them as byte string (`bytes` in python). As such, we provide in the template a few useful (but boring) functions:

- `bytes_str = serialize_list(N, int_list)` returns an object of type `bytes`, containing `N` $\times$ |`int_list`| bytes and encoding the list of integer `int_list` in big endian. Note that N should be large enough so that the binary representation of the elements in the list fit in N bytes.
- `int_list = deserialize_list(N, bytes_str)` is the reverse operation
- `[n1, ..., nj] = deserialize_tuple([i1, ..., ij], bytes_str)` will decode the byte string assuming it was encoded as the concatenation `serialize_list(i1, [n1]) + ... + serialize_list(ija, [nj])` (so contrary to the previous function, elements can have different size, but their number is known in advanced). If some of these elements are byte string instead of numbers (e.g. if the string was obtained as `bytes_str = serialize_list(i1, [n1]) + s2` with `s2` a byte string of length `i2`), you should replace the corresponding size with something like:
  `[n1, s2] = deserialize_tuple([i1, ("bytes", i2)])`
- `b = get_bit(bytes_str, i)` returns the i-th bit of `bytes_str` of type `bytes`.

You need now to implement the other functions:

1. First, define a python class `Options` (used to keep track of the various security parameters used along the protocol) that can be initialized with two optional arguments: `secpar` will default to 80 and `rounds` defaults to secpar unless otherwise specified. Make sure to test your code thanks to the tests we provide (and do the same for all other questions).

2. Then, create a class `Graph()` to represent a graph `g = Graph()`. We choose to represent any graph g thanks to its number n of nodes (each node is labeled in $[n] \coloneqq \{0, \dots, n-1\}$), and by its list of directed edges of the form $(a, b) \in [n]^2$. Create the methods:
   - `v = g.add_node()` that adds a node and returns the id of this node ($v \in [n]$),
   - `g.add_edge(a,b)` that adds an edge (returns nothing),
   - `b = g.edge_exists(a,b)` that outputs `True` if the edge `(a,b)` exists and `False` otherwise (**WARNING:** this operation should be done in constant/logarithmic time over the number of edges),
   - `g.add_double_edge(a,b)` that adds both an edge from `a` to `b` and from `b` to `a`,
   - `n = g.len()` that returns the number of nodes,
   - `l = g.edges()` that returns the **sorted** list of edges of `g` (just use `sorded(...)` to sort it alphabetically),
   - You may also benefit from coding a helper function `s = g.get_graphviz()` that outputs a string representing the graph like `digraph G {n0 -> n1; n1 -> n2;}` that you can for instance visualize in https://dreampuf.github.io/GraphvizOnline (it may be easier to see the graph with the `fdp` engine).
3. Write a function `b = is_hamiltonian_path(g, path)` that outputs a boolean, true iff the path `path` is Hamiltonian. The path is coded as the list of vertices to follow (starting from the first element in the list), like `p = [0, 1, 2]`.
4. Write a function `generate_permutation(n)` (calling only the external, crypto-secure library `randbelow(i)` to sample an element in [`i`]) that returns a uniformly distributed random permutation of [`n`]. This algorithm should follow the Fisher-Yates algorithms, that starts from the list `[0,1,...,n-1]`, and exchanges the first element of the list with a random element to its right (possibly exchanged with itself), then exchanges the second element of the list with a random element to its right etc (so the first element of the list is never changed anymore).
5. Implement 2 functions to implement commitments:
   - A function `(r, c) = commit(options, message)` to commit a `message` of type `bytes`, where `options` an instance of `Options` (the length of the randomness r is `math.ceil(options.secpar/8)`), and `c` is of type `bytes`. The commitment is done by sampling r using `token_bytes`, and the hash is computed via hashing with SHA3-224 the randomness `r` concatenated with the message (use the imported `c = sha3_224(...).digest()`).
   - A function `check_commit(options, c, r, message)` that outputs a boolean, true iff the opening (`r`, `message`) is valid.
6. Implement the functions corresponding to the ZK protocol that checks if a graph admits a Hamiltonian path:
   - First, the function `(info_open, commitments) = commit_phase(g, path, options=Options())` that implements the first (commit) phase of the protocol. `info_open` represents an arbitrary structure kept by the prover for the second phase, while `commitments` has type `bytes`, and length $28 \times (1 + e)$ bytes (28 is the output size of SHA3-224). The first block of 28 bytes is the commitment of the permutation π (serialized via the above functions, where $[i_1, \dots, i_n]$ represents the permutation mapping the node 0 to i_1, the node 1 to $i_2 \dots$) used in the ZK protocol, and the remaining e commitments are the commitments of the edges of g_π (permutation of the graph g). Note that the position of the edges in this list must be randomized for security reasons!
   - Then, the function `openings = open_phase(info_open, b)` runs the opening phase based on the challenge `b` $\in \{$`True`, `False`$\}$. `openings` has type `bytes`, whose format is described in the template.
   - Finally, the function `(ok, reason) = verify(g, commitments, b, opening, options=Options())` perform the verification done by the verifier assuming the challenge was `b`. `ok` is true iff the verifier accepts, and `reason` is an arbitrary (regular) string that gives a reason of rejecting (useful for debug mostly).

7. Implement the functions to make this verification non-interactive based on the Fiat-Shamir transform:

   - `proof = fiat_shamir_proof(g, path, message=b'', options=Options())` that implements the Fiat-Shamir transform of the above protocol, returning a unique, non-interactive, proof. We use SHA3-224 to implement the random oracle (you might benefit from the helper function `get_bit` described above). Since the number of rounds may be larger than 224 (number of output bis of SHA3-224), you should get more enough bits by concatenating: `SHA3-224`($0\|\dots$)$\|$`SHA3-224`($1\|\dots$)$\|$`SHA3-224`($2\|\dots$) until you have enough bits for the challenge. You should also concatenate at the end of each message to be hashed the bytes `message` to obtain a signature following the principle of Schnorr's signature. The format of the `proof` of type `bytes` is the concatenation of all the commitment phases, followed by the concatenation of all the openings.
   - `verify_fiat_shamir_proof(g, proof, message=b'', options=Options())` is the verification procedure.

Part 2: SAT to Hamiltonian graph

Now that we can prove that a graph is Hamiltonian, we want to prove statements about generic circuits and not just graphs. The first step is to turn a SAT instance into a Hamiltonian path as seen in the course.

1. Write the `graph_from_sat(sat, evaluation=None)` function. `sat` is a list of clause, for instance `[[1], [-1, 2]]` represents the formula $(a) \wedge (\neg a \vee b)$ (see the template for details). This functions either returns a single graph `g` corresponding to the `sat` formula if `evaluation` is `None` (see the template that documents the convention to follow on the naming the nodes to keep compatibility with the testing procedure and other implementations). When `evaluation` is a dictionary such that `evaluation[v]` contains the boolean value that the variable `v` (represented by an integer ≥ 1) must take to satisfy `sat`, the function returns (`g`, `path`) where `path` is a Hamiltonian path in g.

Part 3: Circuit to SAT

Finally, we can now write the functions to convert arbitrary circuits C to a SAT instance (and therefore a Hamiltonian graph) in order to prove that there exists x such that $C(x) = y$ for some public y.

1. Here, we write a single function that allows both the verifier to obtain the wanted SAT formula from a given circuit `circ`, but also the prover to obtain both the SAT formula and an evaluation `evaluation = circ.get_evaluation()` of the variables that ensure the SAT formula is true. More precisely, write a class `circ = Circuit()` with the following methods:

   - `sat = circ.get_sat()` returns the sat clauses to satisfy the circuit.
   - `v = circ.add_var(val=None)` creates a new input variable (start from 1 and increment). When `val` is a boolean, it should be understood as if we are evaluating the circuit with input value `val` (note that this should not add any clause, this is just helpful to derive the `evaluation` dictionary).
   - `evaluation = circ.get_evaluation()` returns the table such that `evaluation[v]` is True iff the variable `v` must be true in order to satisfy the SAT formula (if the input of `add_var` were not provided, this can be an empty dictionary).
   - `circ.is_true(v)` (resp. `circ.is_false(v)`) adds a constraint (clause) that forces the variable `v` to be true (resp. false): not that this variable is typically an output variable that we want to force to be true, e.g. to force $f(x) = y$.
   - `c = circ.add_and(a, b)` creates a new variable identifier `c` that should be equal to the AND of the variables `a` and `b` (by adding new clauses in the SAT formula, and by updating the `evaluation` map if possible). For better compatibility with the tests and other implementations, add the clauses in the order specified by this table (cf. Tseytin transformation in the course):

Name	Operation	CNF
AND	$c = a \wedge b$	$(a \vee -c) \wedge (b \vee -c) \wedge (-a \vee -b \vee c)$
OR	$c = a \vee b$	$(-a \vee c) \wedge (-b \vee c) \wedge (a \vee b \vee -c)$
XOR	$c = a \oplus b$	$(a \vee b \vee -c) \wedge (a \vee -b \vee c) \wedge (-a \vee b \vee c) \wedge (-a \vee -b \vee -c)$
NOT	$b = \neg a$	$(a \vee b) \wedge (-a \vee -b)$

We can assume that both `a` and `b` are positive variable identifier. Similarly, define `c = circ.add_or(a, b)`, `c = circ.add_xor(a, b)` and `b = circ.add_not(a)`.

Part 4: Application; playing with a (simplified) Game of Life

We now have all the tools to prove arbitrary statements on the result of a circuit. We exemplify it on the "Rule 110"[1] automaton game (1D equivalent of the famous Game of Life[2]). In the Rule 110 game, a boolean array A of size $w \in \mathbb{N}$ (sometimes considered infinite) is initialized to an arbitrary starting position: then, at every iteration, the array is updated into A' following a simple rule based on the neighboring cells of each cell: for any $i \in [w]$, $A'[i] = ((\neg A[(i-1) \mod w]) \wedge A[i \mod w]) \vee (A[i \mod w] \oplus A[(i+1) \mod w])$. For instance, here are the first 20 iterations (one iteration per line, `X = True`) of a board initialized with a single cell:

```
|                                       X                                         |
|                                      XX                                         |
|                                     XXX                                         |
|                                    XX X                                         |
|                                   XXXXX                                         |
|                                  XX   X                                         |
|                                 XXX  XX                                         |
|                                XX X XXX                                         |
|                               XXXXXXX X                                         |
|                              XX     XXX                                         |
|                             XXX    XX X                                         |
|                            XX X   XXXXX                                         |
|                           XXXXX  XX   X                                         |
|                          XX   X XXX  XX                                         |
|                         XXX  XXXX X XXX                                         |
|                        XX X XX  XXXXX X                                         |
|                       XXXXXXXX XX   XXX                                         |
|                      XX      XXXX  XX X                                         |
|                     XXX     XX  X XXXXX                                         |
|                    XX X    XXX XXXX   X                                         |
|                   XXXXX   XX XXX  X  XX                                         |
```

In this part, we want to obtain a ZK proof proving that we know a secret initial disposition that maps to a publicly known final disposition.

1. Write a function `(sat, evaluation, last_position) = game_110_sat(position, n, is_starting=True)`, such that:
   - if `is_starting=True`, `position` corresponds to the boolean array of the initial position, `last_position` corresponds to the final boolean array after n iterations, SAT corresponds to a SAT formula that is satisfied by the evaluation `evaluation`, and that represents the circuit that runs n iterations and checks if the final disposition is `last_position`.
   - if `is_starting=True`, then `position` corresponds to the final disposition to obtain (run by the verifier since they don't know the initial position). In this case, `evaluation` is not used, `last_position` equals `position`, and `sat` is the SAT formula that is satisfiable only if there exists an initial position reaching to the final position `position` after n steps.

[1] https://en.wikipedia.org/wiki/Rule_110
[2] https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

2. Write a function `(proof, last_position) = game_110_zk_proof(starting_position, n, options=Options())` run by the prover, that outputs a non-interactive ZK proof that there exists a starting position leading to `last_position` after n runs.

3. Finally, write the corresponding verification function `(ok, reason) = game_110_zk_verify(last_position, n, proof, options=Options())` that returns `ok = true` iff `proof` is a valid proof that there exists an initial position leading to `last_position` after n iterations. Reason is as before an arbitrary string explaining the reason of the rejection of the proof if needed.