Formal Methods for Security and Privacy (SS25) Prof. Matteo Maffei

Project 1 – Foundations: Z3 and Lean

Due | *02.04.2024* TU Wien

1 General Remarks

This project is composed of two main parts.

Z3 We will initially focus on the Z3 theorem prover using both the SMT and the Horn clause solvers. First, you will be asked to solve a simple cryptography problem using Z3 as a constraint solver. Then, referring to the encoding of the "example 2" program presented during the lecture you will be asked to apply a small transformation to the horn clauses and discuss the new encoding.

Lean We will then use the Lean4 proof assistant to prove correctness of a simple xor-based symmetric cypher that uses a specific definition of the xor operation.

2 Setting up Your System

This homework requires verifying your answers using a recent version of Z3 and Lean4.

- You can install Z3 and its python bindings using `pip` with the following command:

  ```
  pip install --user z3-solver
  ```

- To use Lean4 locally it is recommended to use the VSCode IDE. Although other editors may support Lean, the official documentation only applies to VSCode.
  - To install VSCode, follow the instructions on the official website.
  - To install the Lean VSCode extension, search for "Lean 4" in the search bar of the *Extensions* sidebar and click on the install button.
    * Access the Lean 4 setup guide: (i) Create a new text file by selecting *File > New Text File*; (ii) click on the ∀ symbol at the top-right corner of the window and select *Documentation > Docs: Show Setup Guide* From the dropdown menu; (iii) read and follow the instructions provided to complete the installation.

- The online version of Lean4 is available at https://live.lean-lang.org. You can copy and paste the `encrypt.lean` file in the editor to verify it.

3 Submission Instructions

Please submit your solution by *02.04.2024* on TUWEL. This project will account for 6 out of the 60 project points.

Your submission must be a single archive file `name-matriculation.zip` containing

1. The `submission.pdf` file containing all your answers and explanations, a LaTeX template can be found on TUWEL.
2. The Lean and Python files containing your solutions, as described in the submission box of each task:

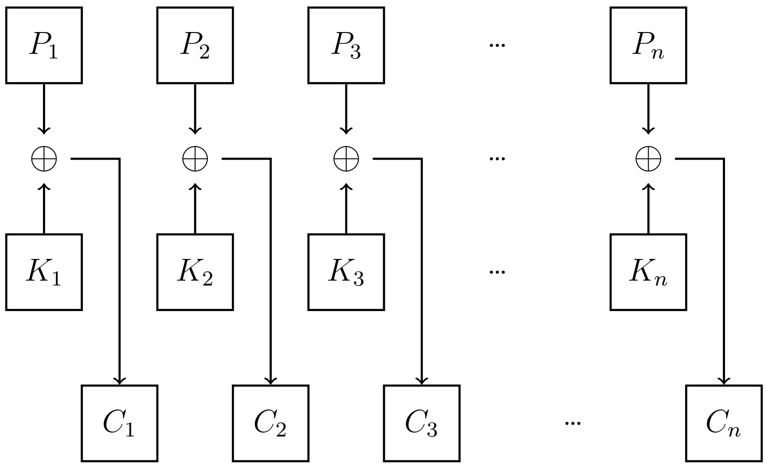
Required files
`encrypt.lean`
`cypher.py`
`initial_values.py`

4 Z3: SMT and Horn-Clause-Based Abstraction

4.1 Stream Ciphers

A stream cipher is a symmetric key cipher in which each byte of the plaintext message is combined (usually using the XOR bitwise operator) with a key stream. This key stream is generated based on a secret key and is typically as long as the message.

An example is depicted in the following image, where P_n are the bytes of the plain text message, K_n are the bytes of the key, $\oplus$ is the XOR bitwise operator, and C_n are the resulting ciphertext bytes

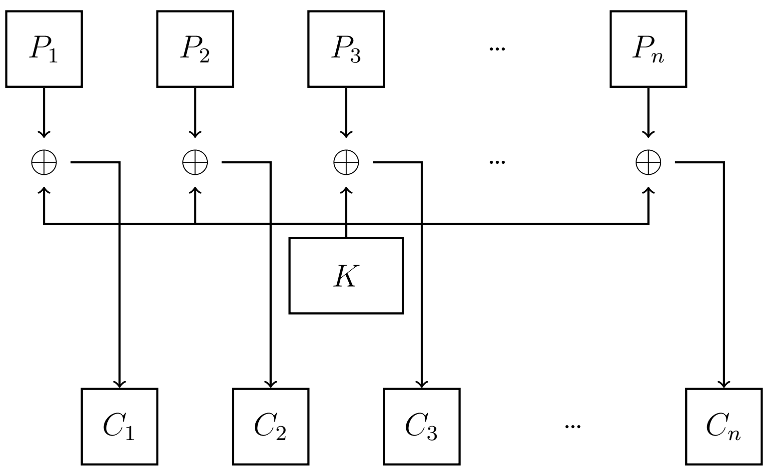


An ideal stream cipher, where the key stream is truly random, behaves like a one-time pad (OTP). As long as the key is never reused, a one-time pad provides perfect secrecy, meaning that the ciphertext reveals no information about the plaintext.

In practice, however, the key stream is often generated dynamically using a pseudo-random number generator (PRNG) seeded with a fixed-length key.

4.1.1 Project Task: The Many-Times Pad

Let us now consider the cipher depicted in the following image:



A *single key value* is used instead of a key stream, so each plain text value P_n is XOR-ed with the **same key** to generate the ciphertext C_n.

Assuming that the encrypted ciphertext C_n is the following byte array, your task is to use Z3 to recover both the original message and the key that was use to encrypt it.

```
[53, 38, 49, 58, 45, 42, 32, 38, 58, 44, 54, 39, 38, 32, 49, 58, 51, 55, 38, 39,
 55, 43, 38, 32, 58, 51, 43, 38, 49, 55, 38, 59, 55, 48, 54, 32, 32, 38, 48, 48,
 37, 54, 47, 47, 58]
```

The `cypher.py` file contains a template for the code required to solve the task. Pay close attention to the comments in the file: they include (i) a description of the BitVector theory necessary to reason about bytes, and (ii) some hints on the constraints required to solve the task.

Submission (1pt). Please add to your submission the modified `cypher.py` file that prints the correct key and message on stdout. You can specify additional notes in the `submission.pdf` in the relevant section.

4.2 Project Task: Horn-Clause-Based Abstraction - Initial Values

Consider the examples for Horn-clause-based abstraction in `example_2.py` and `example_2_loop.py`, both of which can be found in the git repository (in the `01 - Semantics Abstraction SMT` directory) at:

https://gitlab.secpriv.tuwien.ac.at/teaching/fmsp-s25

Recall that we extended all predicates signatures with the initial values of `x` and `y` (stored in the variables `x0` and `y0`) to be able to reason over the relation of initial and final values.

Your task is now to investigate whether this is necessary or if a different method (described below) would also be sufficient to prove the program correct.

We suggest to apply the following steps, starting from the `example_2.py` file.

- step 0: delete all superfluous predicate and rule declarations
- step 1: delete the definition of `query` (you will redefine a query later)
- step 2: introduce a predicate `Init` that has two integer parameters
- step 3: introduce a rule that says "`Init(x,y)` is true for all `x` and `y` greater than 0"
- step 4: change the `initial` rule to say "`S0(x,y,z)` is true for all `x` and `y` such that `Init(x,y)` holds"
- step 5: introduce a new query to the effect of "Given that `Init` is true for `x0` and `y0` and `E0` is true for `x`, `y`, and `z`, can it be the case that `z` is not equal to `x0+z0`" (`example_2_loop.py` already contains a similar query)

Run the query to see if you get the expected result.

Discuss the approach, and in particular if the query gives the result you expect: if it does, describe how it works; if it does not, discuss why.

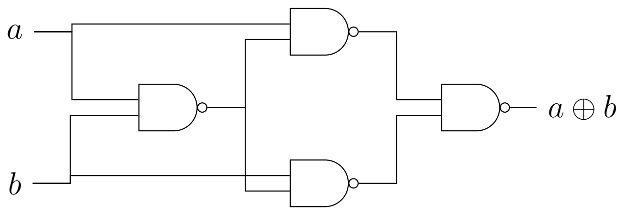
Submission (2pts). Please add to your submission the Python script that you created following steps 0-5, naming it `initial_values.py`. Discuss your implementation and the query results in section 4.2 of the `submission.pdf` report.

5 Lean: Interactive Theorem Proving

5.1 XOR-Based Symmetric Encryption

Let us now focus on the *exclusive or* (XOR) bitwise operator. The XOR bitwise operator applies XOR to pairs of bits in two fixed-width integers (i.e., bit strings or bit vectors). For each pair of bits, XOR returns 1 if the bits are different and 0 if they are equal.

A logical circuit implementing XOR can be constructed using only 4 NAND (NOT AND) gates, as shown in the picture below.



Let us now encode the above circuit as a Lean expression. We start by introducing a new abbreviation to represent unsigned integers of 256 bits:

```
abbrev BV256 : Type := BitVec 256
```

The `BitVec n` type represents all bit vectors (i.e., machine integers) of length `n`. The `BitVec` type supports the bitwise operators you may be familiar with: in this project we will only use three of them, AND, written as `&&&`, OR, written as `|||`, and NOT, written `~~~`.

We can define the NAND gate (that works on `BV256`) using the NOT and AND bitwise operator:

```
def BV256.nand (a b : BV256) : BV256 :=
  ~~~ (a &&& b)
```

With a implementation of NAND, we can implement the above circuit for our `BV256` type.

```
def BV256.xor (a b: BV256) : BV256 :=
  (a.nand (a.nand b)).nand (b.nand (a.nand b))
```

Note that if we define a function in the `BV256` namespace (i.e., prefixed by `BV256.`), we can use it as a *member function* (similarly to Java methods) of every value of the `BV256` type, e.g., `a.nand b` is equivalent to `BV256.nand a b`.

5.1.1 Encryption and Decryption using XOR

We can define encryption and decryption of a message using a key in term of XOR with the following definitions.

```
def encrypt (message key : BV256) : BV256 :=
  message.xor key

def decrypt (cyphertext key: BV256) : BV256 :=
  cyphertext.xor key
```

These definitions represent a valid symmetric cypher if the following *correctness property* is valid.

$$\forall (m\,k\ : \texttt{BV256}),\quad \texttt{decrypt}(\texttt{encrypt}(m, k), k) = m$$

5.1.2 Project Task: Correctness of the XOR cypher

The `encrypt.lean` file includes the above definitions of the `BV256` type, the XOR function and some useful theorems. Your task is to complete the proofs of the theorems marked with the `sorry` placeholder, specifically:

1. (0.5 pts) Following the example of the provided proof of `BV256.demorgan1`, prove

   ```
   theorem BV256.demorgan2 {a b : BV256} : ~~~(a ||| b) = ~~~a &&& ~~~b
   ```

2. (0.5 pts) Prove that 0 is the right identity of XOR.

   ```
   theorem xor_identity {a : BV256} : a.xor (0#256) = a
   ```

   Where the notation `i#n` is the integer `i` represented as a `BitVec` of `n` bits.

3. (0.5 pts) Prove that the XOR of a number with itself is 0.

   ```
   theorem xor_self {a : BV256} : (a.xor a) = (0#256)
   ```

4. (0.75 pts) Prove that XOR is associative.

   ```
   theorem xor_assoc {a b c : BV256} : (a.xor b).xor c = a.xor (b.xor c)
   ```

5. (0.75 pts) Prove that XOR-based encryption is correct.

   ```
   theorem encrypt_correct message key : decrypt (encrypt message key) key = message
   ```

   Note: avoid using the any `simp` tactic to prove this specific theorem. The proof can be easily written in terms of `unfold`, `intro`, `apply`, and `rw` (hint: theorems 2, 3, 4 are all equalities).

Submission (3pts). Please add to your submission the modified Lean script called `encrypt.lean`. The file should not contain any unproven theorem (or `sorry`). The `simp` tactics are allowed in any proof except in the proof of `encrypt_correct`. You can specify additional notes in `submission.pdf` in the relevant section.