Welcome to DojoCrypt!

The DojoCrypt platform is available at https://dojocrypt.tii.ae.

If you are reading this file, you are in the `getting_started` folder. This is a good place to start getting familiar with the **DojoCrypt** platform!

Examples and tutorials

Here you can find some **examples** and **tutorials** on how to use some of the main tools available in the **DojoCrypt images**. Examples and tutorials are provided in the form of python files, jupyter notebooks or shell scripts, depending on the tool.

For each tool, you will find a folder with the name of the tool. Inside this folder, you can find a `README.md` file, with instructions on how to quickly get started with the tool, and some description of the content of the folder.

Your workspace

The `getting_started` folder is read-only, so you will not be able to modify the files in this folder. The content of these files is meant to be copy-pasted into your **workspace**.

You should try to keep your files in the folder `/home/<your_username>/my-files`. This is the only folder for which persistence of your files is guaranteed, even across different **deployments** (see next section) and after your current deployment is deleted. Any file outside the `my-files` folder will be deleted whenever the deployment is deleted. Note, that some tools require to generate files in the folder of the tool, e.g. `/home/tagada/tools`. Remember to bring those files in the `my-files` folder if you need them after the deployment is deleted.

Images and deployments

If you are reading this, you have probably already created a **deployment** from the DojoCrypt dashboard. You might have noticed that, during the process of creating this deployment, you were asked to select an image.

IMPORTANT NOTE: the examples for one tool will only work in the image related to that tool.

In the free version, you can only create one deployment at a time. If you need to *create* a new deployment, you first need to *delete* the current deployment.

SSH connection

You can connect to your deployments also using **SSH**. You only need to

1. open a terminal in a device with internet connection;
2. copy-paste the command displayed in the deployment card into the terminal;
3. when prompted, copy-paste the password displayed in the deployment card into the terminal;

Your terminal will enter into your deployment home folder `/home/<username>/`, inside which you can find the `my-files` folder.

How to contribute to the project

There is a number of ways in which you could contribute to the project:

- Share a docker image you would like to push on the DojoCrypt platform: please share a link to the dockerfile and some example to test the image.
- Provide suggestions on how to improve the web-app.
- Fund the project: this can be done with
  - direct payments,
  - by participating to research grants together,

    - pay your own deployments
    - ask us to deploy DojoCrypt in your infrastructure
- Use Dojocrypt:
    - in your research
    - in your lessons
- Talk about DojoCrypt!

For all the above, you can reach us out from the **Contact us** section at dojocrypt.tii.ae.

Questions?

For any question, feel free to reach us out through the **Contact us** section at dojocrypt.tii.ae.

A collection of scripts to learn CLAASP

This collection contains short lessons, scripts and tutorials, usually in the form of Jupyter Notebooks, to become familiar with the CLAASP library.

Introduction to CLAASP

It is advised to start from the `introduction_to_claasp` folder, for example:

- read the `claasp_overview.md` file to have a quick overview of CLAASP;
- read the `claasp_basic_introcution.ipynb` notebook to have an overview of CLAASP;
- read the notebooks in the `cipher_evaluation` folder to learn how to evaluate a cipher in CLAASP;
- read the files in the `intro_to_differential_cryptanalysis_from_block_cipher_companion` folder to have a basic introduction to differential cryptanalysis;
- check the `toy_ciphers` folder to see some example of toy ciphers implemented in CLAASP.

Did you know that?

The folder `did_you_know_that` contains a collection of very short scripts to quickly learn how to perform basic cryptanalysis tasks, such as:

- find a linear or differential trail
- run avalanche tests
- find a neural distinguisher
- print your test results in a latex table
- and many more...

Tutorials

The folder `tutorials` contains a collection of more in depth tutorials on specific CLAASP modules.

CLAASP: a Cryptographic Library for the Automated Analysis of Symmetric Primitives

Time of writing: Pi-day 2025.

CLAASP is a library whose *goal* is to provide an extensive toolbox gathering state-of-the-art techniques aimed at simplifying the manual tasks of symmetric cipher designers and analysts.

CLAASP is an **opensource** library built on top of Sagemath.

It is designed to be

- modular
- extendable
- easy-to-use
- generic (allowing the implementation and analysis of a wide range of cipher designs)
- automated (input a cipher design and output an analysis of the cipher design with respect to some desired property).

Resources

- CLAASP repository: https://github.com/Crypto-TII/claasp
- CLAASP readthedocs: https://claasp.readthedocs.io/en/stable/
- CLAASP white paper (not updated, better to not use it as a reference): https://eprint.iacr.org/2023/622.pdf (use readthedocs or the `getting_started` tutorials)

Who made CLAASP?

The CLAASP project started around 2019, and, up to know, it has been mostly funded by the Technology Innovation Institute (TII) of Abu Dhabi, in the UAE.

Up to this date, it counts more than 50 contributors and collaborators, including bachelor/master/Ph.D. students, post-docs, professors, software developers, devops engineers.

Among the main institutions involved in the projects:

- Technology Innovation Institute, UAE
- University of Milan, Italy
- Politecnico di Torino, Italy
- Radboud University, Netherland

- Nanyang Technological University, Singapore
- RomaTre University, Italy
- LeanMind, Spain

How does CLAASP compare with other similar tools?

WARNING: The following tables are **work-in-progress**.

	CASCADA	CLAASP	CryptoSMT	TAGADA
Paper published in	2022	2023	2016?	2021
Goal	SMT bit-vector	General Purpose	SMT bit-vector	Minizinc/aligned-SPN focused
Programming Language	Python	Python/SsageMath	Python/CommandLine	Rust
Cipher representation	DAG	DAG	SMT API	DAG
#Ciphers	20+	80+	20+	6
Solvers	SMT	SAT, SMT, MILP, CP	SMT	CP
Models	Differential, Rotational-XOR, Impossible-Differential, Impossible-rotational-XOR, Linear, Zero-correlation	Differential, Linear, Deterministic/Probabilistic Truncated-Differential, Impossible-Differential, Zero-correlation, Boomerang, Division Property, Hash Pre-image, Key-recovery	Differential, Linear, hash Pre-image, Key-recovery	Differential, Truncated-Differential

Available Models

	CLAASP	CASCADA	CryptoSMT	TAGADA
single-key differential cryptanalysis	✅	✅	✅	✅
related-key differential cryptanalysis	✅	✅	✅	✅

	CLAASP	CASCADA	CryptoSMT	TAGADA
single-key rotational-XOR cryptanalysis	❌	✅	?	❌
related-key rotational-XOR cryptanalysis	❌	?	?	❌
single-key truncated-differential cryptanalysis	✅	❌	❌	✅
related-key truncated-differential cryptanalysis	✅	❌	❌	✅
single-key impossible-differential cryptanalysis	✅	✅	?	❌
related-key impossible-differential cryptanalysis	✅	✅	?	❌
single-key impossible-rotational-XOR cryptanalysis	❌	✅	?	❌
related-key impossible-rotational-XOR cryptanalysis	❌	?	?	❌
linear cryptanalysis	✅	✅	?	❌
single-key zero-correlation cryptanalysis	✅	✅	?	❌
differential-linear cryptanalysis	✅	❌	❌	❌
(three-subset) division property	✅	❌	❌	❌
boomerang distinguisher	✅	❌	❌	❌
hash pre-image	✅	❌	✅	❌
key-recovery	✅	❌	✅	❌

What can you do with CLAASP exactly?

Here is a list of task you can perform with CLAASP (as a user):

1. **Cipher management**
    - **80+** already implemented ciphers
    - Create your **own cipher** using the following components:
        a. OR
        b. AND
        c. XOR
        d. NOT
        e. Linear/Nonlinear Feedback Shift Register (over any field)
        f. S-box

        g. Modular Addition
        h. Modular Subtraction
        i. Constant
        j. LinearLayer
        k. Shift
        l. Rotate
        m. Variable Shift
        n. Variable Rotate
    - **Manipulate** the cipher:
        - select subcomponents by id or by round and position in the round
        - remove parts of the cipher, e.g. key-schedule
        - invert the cipher
        - ...
    - Analyze the **properties** of the cipher **components**
2. **Cipher evaluation**
    - Generate the cipher **evaluation code** or directly evaluate the cipher using:
        - Python Bitstring
        - Python Numpy
            - allows to evaluate **multiple inputs simultaneously**, especially useful with **GPU***
            - can be used to train **neural distinguishers** more efficiently
        - Standard C with BitString class
        - Standard C with WordString class
        - Cuda (Coming soon...)
3. **Avalanche and Statistical tests**
    - Statistical tests
        - NIST
        - DIEHARDER
    - Avalanche tests
        - avalanche probability vectors
        - avalanche dependence
        - avalanche dependence uniform
        - avalanche weight
        - avalanche entropy
    - Continuous diffusion tests
        - continuous avalanche factor
        - continuous diffusion factor
        - continuous neutrality measure
4. **Cipher algebraic models**
    - Model the cipher as a **system of Booleab equations**, and
    - try to solve it using Groebner basis calculators from SageMath
    - Fix values of some variables and find the others, e.g.
        - compute **pre-images**,
        - Perform a **partial key-recovery** given ciphertext, plaintext and some bits of the key

5. **Cipher models for trail search**
    - Models that search for trails, usually implement some or all the following methods (or a variant):
        - find one trail
        - find one trail with fixed weight
        - find all trails with fixed weight
        - find all trails with weight at most
        - find the lowest weight trail
    - The following models are included in CLAASP:

Model	**SAT**	**SMT**	**MILP**	**CP**
Cipher Model	✓	✓	✓	✓
Cipher Model ARX Optimized				✓
XOR Differential	✓	✓	✓	✓
XOR Differential ARX Optimized				✓
XOR Differential Number of Active Sboxes				✓
XOR Differential Trail Search Fixing Active Sboxes				✓
XOR Linear	✓	✓	✓	✓
Bitwise Deterministic Truncated XOR Differential	✓	✓	✓	✓
Bitwise Deterministic Truncated XOR Differential ARX Optimized				✓
Wordwise Deterministic Truncated XOR Differential			✓	✓
Probabilistic XOR Truncated Differential	✓		✓	
Bitwise Impossible XOR Differential			✓	✓
Wordwise Impossible XOR Differential			✓	✓ (in progress)
Hybrid Impossible XOR Differential				✓
Boomerang ARX Optimized				✓ (in progress)

6. **Neural distinguishers**
    - build a neural network blackbox distinguisher
    - run AutoND pipeline
    - find good input difference for neural distinguisher
    - build a neural network differential distinguisher (a-la-Gohr, given an input difference)
7. **Print reports**

- **Latex** tables
- **json** dictionaries
- Python **pandas** dataframes
- Python **plotly** plots and graphs

What can you NOT do with CLAASP?

At the moment, CLAASP does not support at least the following:

- Bit-vector SMT modelling
- ModAdd Differential trails, or trails based on other operations
- Rotational-XOR
- Zero-correlation
- Non-trivial and generic automatic **key-recovery** (still possible to write a script for a specific cipher or to use the cipher models)
- Represent ciphers over **non-binary fields** (not ideal for algebraic ciphers)
- Compile the CLAASP representation to **other tools representations**
- ...

CLAASP basic idea

The basic block of CLAASP is the description of a cryptographic primitives in the form of a **list of connected components** (S-Box, LinearLayer, Constants, Input/Output, etc.).

From this representation, the library can:

- generate the Python or C code of the encryption function,
- execute a wide range of statistical and avalanche tests on the primitive,
- automatically generate SAT, SMT, CP and MILP models to search, for example, differential and linear trails,
- measure algebraic properties of the cipher,
- test neural-based distinguishers.

A cipher as an acyclic directed graph

Informally, in CLAASP, a symmetric cipher is represented as a list of "connected components".

By the term **cipher component** (or simply **component**) we refer to the building blocks of symmetric ciphers (S-Boxes, linear layers, word operations, etc.).

We say that two components are **connected** when the output bits of the first component become the input bits of the second component.

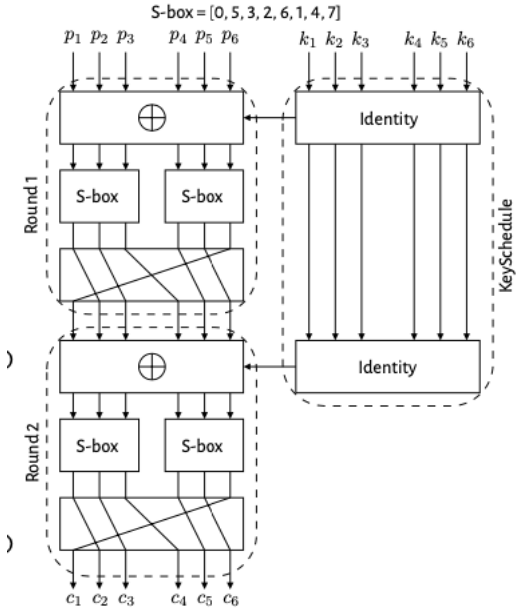
claasp_how_to_implement_a_cipher_object

March 17, 2025

1 How to implement a block cipher

Consider the following ToySPN block cipher: - 6-bit block SPN - 6-bit key - non-linear layer with 2 3-bit S-Boxes - linear layer with a 1-bit right rotation - no key schedule - 2 rounds only

You can check the diagram in the figure below:



[1]:
```
from claasp.cipher import Cipher

class ToySPN(Cipher):
    def __init__(self):
        super().__init__(
            family_name="toyspn",
            cipher_type="block_cipher",
            cipher_inputs=["plaintext", "key"],
            cipher_inputs_bit_size=[6, 6],
            cipher_output_bit_size=6
            )
        sbox = [0, 5, 3, 2, 6, 1, 4, 7]
```

```
        self.add_round()
        xor = self.add_XOR_component(["plaintext",␣
↪"key"],[[0,1,2,3,4,5],[0,1,2,3,4,5]],6)
        sbox1 = self.add_SBOX_component([xor.id], [[0, 1, 2]], 3, sbox)
        sbox2 = self.add_SBOX_component([xor.id], [[3, 4, 5]], 3, sbox)
        rotate = self.add_rotate_component([sbox1.id, sbox2.id],[[0, 1, 2], [0,␣
↪1, 2]], 6, 1)

        self.add_round_output_component([rotate.id], [[0, 1, 2, 3, 4, 5]], 6)

        self.add_round()
        xor = self.add_XOR_component([rotate.id,␣
↪"key"],[[0,1,2,3,4,5],[0,1,2,3,4,5]],6)
        sbox1 = self.add_SBOX_component([xor.id], [[0, 1, 2]], 3, sbox)
        sbox2 = self.add_SBOX_component([xor.id], [[3, 4, 5]], 3, sbox)
        rotate = self.add_rotate_component([sbox1.id, sbox2.id],[[0, 1, 2], [0,␣
↪1, 2]], 6, 1)

        self.add_cipher_output_component([rotate.id], [[0, 1, 2, 3, 4, 5]], 6)
```

Let us know instantiate the cipher and evaluate it over a plaintext and a key

[8]:
```
toyspn = ToySPN()
plaintext = 0x3F
key = 0x31
print(f'ciphertext = {hex(toyspn.evaluate([plaintext, key]))}')
```

```
ciphertext = 0x3b
```

[ ]:
```

```

claasp_basic_introduction

March 19, 2025

1 CLAASP: a basic introduction

1.1 Components, Ciphers and Solvers

First, let us see which components, ciphers and solvers are available in CLAASP.

First we need to import all classes from the CLAASP package.

[2]:
```
import os
import importlib
import inspect

def import_all_classes_from_package(package_name, package_dir):
    all_classes = []

    for root, _, files in os.walk(package_dir):
        for file in files:
            if file.endswith('.py') and file != '__init__.py':
                module_rel_path = os.path.relpath(os.path.join(root, file),␣
 ↪package_dir)
                module_name, _ = os.path.splitext(module_rel_path.replace(os.
 ↪path.sep, '.'))

                full_module_name = f"{package_name}.{module_name}"
                try:
                    imported_module = importlib.import_module(full_module_name)
                    module_classes = inspect.getmembers(imported_module,␣
 ↪inspect.isclass)
                    all_classes.extend(module_classes)
                except Exception as e:
                    print(f"Error importing module '{full_module_name}': {e}")

    return all_classes

package_name = 'claasp'  # name of the package
package_dir = '/home/sage/tii-claasp/claasp'  # directory path of the package
all_classes = import_all_classes_from_package(package_name, package_dir)
```

```
2025-03-14 23:50:47.413146: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not
```

```
find cuda drivers on your machine, GPU will not be used.
2025-03-14 23:50:47.459920: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not
find cuda drivers on your machine, GPU will not be used.
2025-03-14 23:50:48.256404: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not
find TensorRT
```

Now we can print all CLAASP components:

[24]:
```
from claasp.component import Component
from claasp.components.modular_component import Modular
from claasp.components.multi_input_non_linear_logical_operator_component import␣
 ↪MultiInputNonlinearLogicalOperator
from claasp.components.linear_layer_component import LinearLayer

component_subclasses = Component.__subclasses__()
modular_subclasses = Modular.__subclasses__() # parent class of ModAdd and␣
 ↪ModSub
minllo_subclasses = MultiInputNonlinearLogicalOperator.__subclasses__() #␣
 ↪parent class of AND, OR
linearlayer_subclasses = LinearLayer.__subclasses__() # parent class of Sigma,␣
 ↪Reverse, MixColumn, Permutation, ThetaXoodoo, ThetaKeccak

all_subclasses = minllo_subclasses + component_subclasses +␣
 ↪linearlayer_subclasses + modular_subclasses

for component in all_subclasses:
    if not component.__name__ in ['CipherOutput', 'Concatenate', 'Modular',␣
 ↪'MultiInputNonlinearLogicalOperator']: # concatenate is a legacy component
        print(component.__name__)
print()
```

```
OR
AND
XOR
NOT
FSR
SBOX
SHIFT
LinearLayer
Rotate
Constant
VariableShift
VariableRotate
Sigma
Reverse
MixColumn
Permutation
```

```
ThetaXoodoo
ThetaKeccak
MODADD
MODSUB
```

And then all CLAASP ciphers:

[4]:
```
from claasp.cipher import Cipher

counter = 1
for cipher in Cipher.__subclasses__():
    print(f'{counter:>3} Cipher name: {cipher.__name__}')
    counter += 1
print()
```

```
  1 Cipher name: ThreefishBlockCipher
  2 Cipher name: AESBlockCipher
  3 Cipher name: AradiBlockCipher
  4 Cipher name: AradiBlockCipherSBox
  5 Cipher name: AradiBlockCipherSBoxAndCompactLinearMap
  6 Cipher name: BalletBlockCipher
  7 Cipher name: BEA1BlockCipher
  8 Cipher name: DESBlockCipher
  9 Cipher name: DESExactKeyLengthBlockCipher
 10 Cipher name: HightBlockCipher
 11 Cipher name: KasumiBlockCipher
 12 Cipher name: LBlockBlockCipher
 13 Cipher name: LeaBlockCipher
 14 Cipher name: LowMCBlockCipher
 15 Cipher name: MidoriBlockCipher
 16 Cipher name: PresentBlockCipher
 17 Cipher name: PrinceBlockCipher
 18 Cipher name: PrinceV2BlockCipher
 19 Cipher name: QARMAv2BlockCipher
 20 Cipher name: QARMAv2MixColumnBlockCipher
 21 Cipher name: RaidenBlockCipher
 22 Cipher name: RC5BlockCipher
 23 Cipher name: SCARFBlockCipher
 24 Cipher name: SimeckBlockCipher
 25 Cipher name: SimeckSboxBlockCipher
 26 Cipher name: SimonBlockCipher
 27 Cipher name: SimonSboxBlockCipher
 28 Cipher name: SkinnyBlockCipher
 29 Cipher name: SparxBlockCipher
 30 Cipher name: SpeckBlockCipher
 31 Cipher name: SpeedyBlockCipher
 32 Cipher name: TeaBlockCipher
 33 Cipher name: TwineBlockCipher
```

```
34 Cipher name: TwofishBlockCipher
35 Cipher name: UblockBlockCipher
36 Cipher name: XTeaBlockCipher
37 Cipher name: Blake2HashFunction
38 Cipher name: BlakeHashFunction
39 Cipher name: MD5HashFunction
40 Cipher name: SHA1HashFunction
41 Cipher name: SHA2HashFunction
42 Cipher name: WhirlpoolHashFunction
43 Cipher name: AsconPermutation
44 Cipher name: AsconSboxSigmaNoMatrixPermutation
45 Cipher name: AsconSboxSigmaPermutation
46 Cipher name: ChachaPermutation
47 Cipher name: GastonPermutation
48 Cipher name: GastonSboxPermutation
49 Cipher name: GiftPermutation
50 Cipher name: GiftSboxPermutation
51 Cipher name: GimliPermutation
52 Cipher name: GimliSboxPermutation
53 Cipher name: GrainCorePermutation
54 Cipher name: KeccakInvertiblePermutation
55 Cipher name: KeccakPermutation
56 Cipher name: KeccakSboxPermutation
57 Cipher name: PhotonPermutation
58 Cipher name: SalsaPermutation
59 Cipher name: SparklePermutation
60 Cipher name: SpongentPiFSRPermutation
61 Cipher name: SpongentPiPermutation
62 Cipher name: SpongentPiPrecomputationPermutation
63 Cipher name: TinyJambuWordBasedPermutation
64 Cipher name: TinyJambuFSRWordBasedPermutation
65 Cipher name: TinyJambuPermutation
66 Cipher name: XoodooInvertiblePermutation
67 Cipher name: XoodooPermutation
68 Cipher name: XoodooSboxPermutation
69 Cipher name: A51StreamCipher
70 Cipher name: A52StreamCipher
71 Cipher name: BiviumStreamCipher
72 Cipher name: BluetoothStreamCipherE0
73 Cipher name: Snow3GStreamCipher
74 Cipher name: TriviumStreamCipher
75 Cipher name: ZucStreamCipher
76 Cipher name: ConstantBlockCipher
77 Cipher name: FancyBlockCipher
78 Cipher name: IdentityBlockCipher
79 Cipher name: ToyCipherFour
80 Cipher name: ToyFeistel
81 Cipher name: ToySPN1
```

```
82 Cipher name: ToySPN2
```

To print the available solvers, we do as follows:

NOTE: internal solvers are the ones that are called through the SageMath interface.

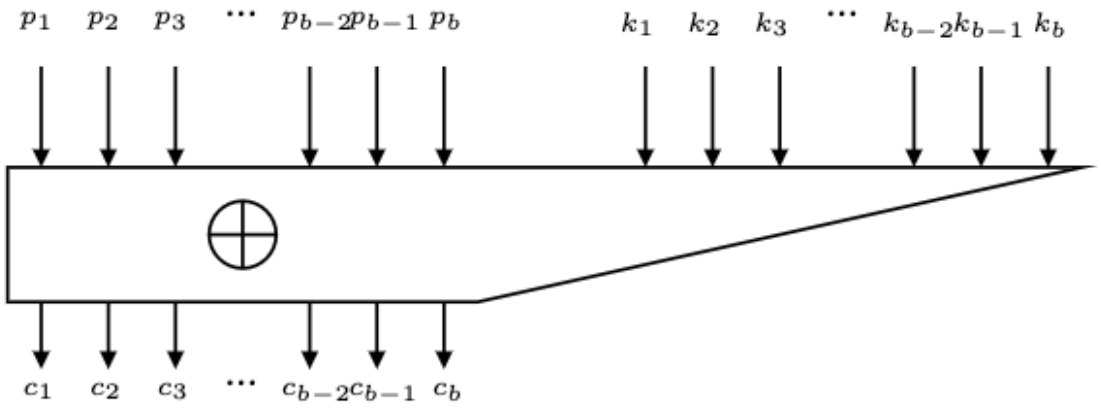
[14]:
```
from claasp.cipher_modules.models.sat.solvers import SAT_SOLVERS_INTERNAL,␣
 ↪SAT_SOLVERS_EXTERNAL
from claasp.cipher_modules.models.smt.solvers import SMT_SOLVERS_INTERNAL,␣
 ↪SMT_SOLVERS_EXTERNAL
from claasp.cipher_modules.models.milp.solvers import MILP_SOLVERS_INTERNAL,␣
 ↪MILP_SOLVERS_EXTERNAL
from claasp.cipher_modules.models.cp.solvers import CP_SOLVERS_INTERNAL,␣
 ↪CP_SOLVERS_EXTERNAL

print("SAT solvers")
print(f"internal: {[solver['solver_name'] for solver in SAT_SOLVERS_INTERNAL]}")
print(f"external: {[solver['solver_name'] for solver in SAT_SOLVERS_EXTERNAL]}")

print("SMT solvers")
print(f"internal: {[solver['solver_name'] for solver in SMT_SOLVERS_INTERNAL]}")
print(f"external: {[solver['solver_name'] for solver in SMT_SOLVERS_EXTERNAL]}")

print("MILP solvers")
print(f"internal: {[solver['solver_name'] for solver in␣
 ↪MILP_SOLVERS_INTERNAL]}")
print(f"external: {[solver['solver_name'] for solver in␣
 ↪MILP_SOLVERS_EXTERNAL]}")

print("CP solvers")
print(f"internal: {[solver['solver_name'] for solver in CP_SOLVERS_INTERNAL]}")
print(f"external: {[solver['solver_name'] for solver in CP_SOLVERS_EXTERNAL]}")
```

```
SAT solvers
internal: ['cryptominisat', 'picosat', 'glucose', 'glucose-syrup']
external: ['CADICAL_EXT', 'CRYPTOMINISAT_EXT', 'GLUCOSE_EXT',
'GLUCOSE_SYRUP_EXT', 'KISSAT_EXT', 'PARKISSAT_EXT', 'MATHSAT_EXT',
'MINISAT_EXT', 'YICES_SAT_EXT']
SMT solvers
internal: []
external: ['MATHSAT_EXT', 'YICES_EXT', 'Z3_EXT']
MILP solvers
internal: ['GLPK', 'GLPK/exact', 'Coin', 'CVXOPT', 'Gurobi', 'PPL',
'InteractiveLP']
external: ['GUROBI_EXT', 'GLPK_EXT', 'SCIP_EXT', 'CPLEX_EXT']
CP solvers
internal: ['choco', 'chuffed', 'coin-bc', 'cplex', 'findmus', 'gecode',
'globalizer', 'gurobi', 'scip', 'Xor', 'xpress']
```

```
external: ['Chuffed', 'gecode', 'Xor', 'coin-bc', 'choco']
```

1.2 How to implement the XOR Block Cipher

We implement a class representing the XOR Block Cipher, taking one key and one plaintext of the same bit size as input, and returning their bitwise sum modulo 2 (XOR).



[1]:
```
from claasp.cipher import Cipher

class XorCipher(Cipher):
    def __init__(self, block_size=128):
        super().__init__(
                    family_name="xorcipher",
                    cipher_type="block_cipher",
                    cipher_inputs=["plaintext", "key"],
                    cipher_inputs_bit_size=[block_size, block_size],
                    cipher_output_bit_size=block_size
                    )
        self.add_round()
        xor = self.add_XOR_component(
            ["plaintext", "key"],
            [
                [i for i in range(block_size)],
                [i for i in range(block_size)]
            ],
            block_size)
        self.add_cipher_output_component(
            [xor.id],
            [
                [i for i in range(block_size)]
            ],
            block_size)
```

The Cipher class contains methods that allow us to evaluate the cipher over a set of integers representing the input bitstring. For example - the integer 1 represents the bitstring 0b1. - the

integer 15 represents the bitstring 0b1111. - the integer 0xAB represents the bitstring 0b10101011. and so on… Let us recall that, in Python, one can represent integers using base 10, 16 or 2. For example, the number 15, can be represented as 15, 0xF, 0b1111, respectively in base 10, 16 and 2. One can also print the representation in base 16 and 2 by simply writing

```
print(hex(15))
print(bin(15))
```

Let us know test the cipher:

[2]:
```
cipher = XorCipher(block_size=1)
key = 0b1
plaintext = 0b1
ciphertext = 0b0
print(f'{bin(cipher.evaluate([key, plaintext])) = }')
print(cipher.evaluate([key, plaintext]) == ciphertext)

cipher = XorCipher(block_size=128)
key = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
plaintext = 0x55555555555555555555555555555555
ciphertext = 0xAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
print(f'{hex(cipher.evaluate([key, plaintext])) = }')
print(cipher.evaluate([key, plaintext]) == ciphertext)
```

```
bin(cipher.evaluate([key, plaintext])) = '0b0'
True
hex(cipher.evaluate([key, plaintext])) = '0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
True
```

1.3 Check cipher properties

It is possible to check the cipher properties by calling the available cipher methods

[3]:
```
cipher = XorCipher(block_size=128)
cipher.as_python_dictionary()
```

[3]:
```
{'cipher_id': 'xorcipher_p128_k128_o128_r1',
 'cipher_type': 'block_cipher',
 'cipher_inputs': ['plaintext', 'key'],
 'cipher_inputs_bit_size': [128, 128],
 'cipher_output_bit_size': 128,
 'cipher_number_of_rounds': 1,
 'cipher_rounds': [[{'id': 'xor_0_0',
    'type': 'word_operation',
    'input_bit_size': 256,
    'input_id_link': ['plaintext', 'key'],
    'input_bit_positions': [[0,
      1,
      2,
```

```
3,
4,
5,
6,
7,
8,
9,
10,
11,
12,
13,
14,
15,
16,
17,
18,
19,
20,
21,
22,
23,
24,
25,
26,
27,
28,
29,
30,
31,
32,
33,
34,
35,
36,
37,
38,
39,
40,
41,
42,
43,
44,
45,
46,
47,
48,
49,
```

```
50,
51,
52,
53,
54,
55,
56,
57,
58,
59,
60,
61,
62,
63,
64,
65,
66,
67,
68,
69,
70,
71,
72,
73,
74,
75,
76,
77,
78,
79,
80,
81,
82,
83,
84,
85,
86,
87,
88,
89,
90,
91,
92,
93,
94,
95,
96,
```

```
 97,
 98,
 99,
 100,
 101,
 102,
 103,
 104,
 105,
 106,
 107,
 108,
 109,
 110,
 111,
 112,
 113,
 114,
 115,
 116,
 117,
 118,
 119,
 120,
 121,
 122,
 123,
 124,
 125,
 126,
 127],
[0,
 1,
 2,
 3,
 4,
 5,
 6,
 7,
 8,
 9,
 10,
 11,
 12,
 13,
 14,
 15,
```

16,
17,
18,
19,
20,
21,
22,
23,
24,
25,
26,
27,
28,
29,
30,
31,
32,
33,
34,
35,
36,
37,
38,
39,
40,
41,
42,
43,
44,
45,
46,
47,
48,
49,
50,
51,
52,
53,
54,
55,
56,
57,
58,
59,
60,
61,
62,

63,
64,
65,
66,
67,
68,
69,
70,
71,
72,
73,
74,
75,
76,
77,
78,
79,
80,
81,
82,
83,
84,
85,
86,
87,
88,
89,
90,
91,
92,
93,
94,
95,
96,
97,
98,
99,
100,
101,
102,
103,
104,
105,
106,
107,
108,
109,

```
    110,
    111,
    112,
    113,
    114,
    115,
    116,
    117,
    118,
    119,
    120,
    121,
    122,
    123,
    124,
    125,
    126,
    127]],
  'output_bit_size': 128,
  'description': ['XOR', 2]},
 {'id': 'cipher_output_0_1',
  'type': 'cipher_output',
  'input_bit_size': 128,
  'input_id_link': ['xor_0_0'],
  'input_bit_positions': [[0,
    1,
    2,
    3,
    4,
    5,
    6,
    7,
    8,
    9,
    10,
    11,
    12,
    13,
    14,
    15,
    16,
    17,
    18,
    19,
    20,
    21,
    22,
```

```
23,
24,
25,
26,
27,
28,
29,
30,
31,
32,
33,
34,
35,
36,
37,
38,
39,
40,
41,
42,
43,
44,
45,
46,
47,
48,
49,
50,
51,
52,
53,
54,
55,
56,
57,
58,
59,
60,
61,
62,
63,
64,
65,
66,
67,
68,
69,
```

```
70,
71,
72,
73,
74,
75,
76,
77,
78,
79,
80,
81,
82,
83,
84,
85,
86,
87,
88,
89,
90,
91,
92,
93,
94,
95,
96,
97,
98,
99,
100,
101,
102,
103,
104,
105,
106,
107,
108,
109,
110,
111,
112,
113,
114,
115,
116,
```

```
      117,
      118,
      119,
      120,
      121,
      122,
      123,
      124,
      125,
      126,
      127]],
    'output_bit_size': 128,
    'description': ['cipher_output']}]],
 'cipher_reference_code': None}
```

We can also retrieve the components of the cipher, given their round number and the relative number of the component within the round (Remember to count from 0!):

[4]:
```
cipher.get_number_of_components_in_round(0)
```

[4]:
```
2
```

[5]:
```
cipher.component_from(0,0).as_python_dictionary()
cipher.component_from(0,1).as_python_dictionary()
```

[5]:
```
{'id': 'cipher_output_0_1',
 'type': 'cipher_output',
 'input_bit_size': 128,
 'input_id_link': ['xor_0_0'],
 'input_bit_positions': [[0,
   1,
   2,
   3,
   4,
   5,
   6,
   7,
   8,
   9,
   10,
   11,
   12,
   13,
   14,
   15,
   16,
   17,
   18,
```

19,
20,
21,
22,
23,
24,
25,
26,
27,
28,
29,
30,
31,
32,
33,
34,
35,
36,
37,
38,
39,
40,
41,
42,
43,
44,
45,
46,
47,
48,
49,
50,
51,
52,
53,
54,
55,
56,
57,
58,
59,
60,
61,
62,
63,
64,
65,

66,
67,
68,
69,
70,
71,
72,
73,
74,
75,
76,
77,
78,
79,
80,
81,
82,
83,
84,
85,
86,
87,
88,
89,
90,
91,
92,
93,
94,
95,
96,
97,
98,
99,
100,
101,
102,
103,
104,
105,
106,
107,
108,
109,
110,
111,
112,

```
    113,
    114,
    115,
    116,
    117,
    118,
    119,
    120,
    121,
    122,
    123,
    124,
    125,
    126,
    127]],
  'output_bit_size': 128,
  'description': ['cipher_output']}
```

or given their ID:

[19]:
```
cipher.get_component_from_id("xor_0_0").as_python_dictionary()
```

[19]:
```
{'id': 'xor_0_0',
 'type': 'word_operation',
 'input_bit_size': 16,
 'input_id_link': ['plaintext', 'key'],
 'input_bit_positions': [[0, 1, 2, 3, 4, 5, 6, 7], [0, 1, 2, 3, 4, 5, 6, 7]],
 'output_bit_size': 8,
 'description': ['XOR', 2]}
```

We can retrieve all the components of the cipher, or all their IDs:

[15]:
```
cipher = XorCipher(block_size=8)
cipher.get_all_components()
```

[15]:
```
[<claasp.components.xor_component.XOR object at 0x7fc73e9014e0>,
 <claasp.components.cipher_output_component.CipherOutput object at
0x7fc75c4478b0>]
```

[16]:
```
cipher.get_all_components_ids()
```

[16]:
```
['xor_0_0', 'cipher_output_0_1']
```

We can check if a cipher belongs to a specific family:

[23]:
```
print(f'{cipher.is_andrx() = }')
print(f'{cipher.is_arx() = }')
print(f'{cipher.is_spn() = }')
```

```
cipher.is_andrx() = True
cipher.is_arx() = True
cipher.is_spn() = False
```

1.4 Cipher Inverse

CLAASP can automatically compute the inverse of a cipher. This is done, under the hood, by inverting the underlying graph and its components.

Let us see it in action while inverting the XORCipher:

[24]:
```
xorcipher = XorCipher(block_size=8)

xorcipher_inverse = cipher.cipher_inverse()

plaintext = 0xFF
key = 0xFF
ciphertext = cipher.evaluate([plaintext, key])

xorcipher_inv = cipher.cipher_inverse()

xorcipher_inv.evaluate([ciphertext, key]) == plaintext
```

[24]:
```
True
```

1.5 A cipher as a system of Boolean polynomials

CLAASP allows to express the cipher as a system of Boolean polynomials. The variables represent the input (denoted by an "x" postfix) and the output (denoted by an "y" postfix) bits of each component, including the input and the output components of the cipher.

[25]:
```
cipher.polynomial_system()
```

[25]:
```
[key_y0 + plaintext_y0 + cipher_output_0_1_x0, key_y1 + plaintext_y1 +
cipher_output_0_1_x1, key_y2 + plaintext_y2 + cipher_output_0_1_x2, key_y3 +
plaintext_y3 + cipher_output_0_1_x3, key_y4 + plaintext_y4 +
cipher_output_0_1_x4, key_y5 + plaintext_y5 + cipher_output_0_1_x5, key_y6 +
plaintext_y6 + cipher_output_0_1_x6, key_y7 + plaintext_y7 +
cipher_output_0_1_x7]
```