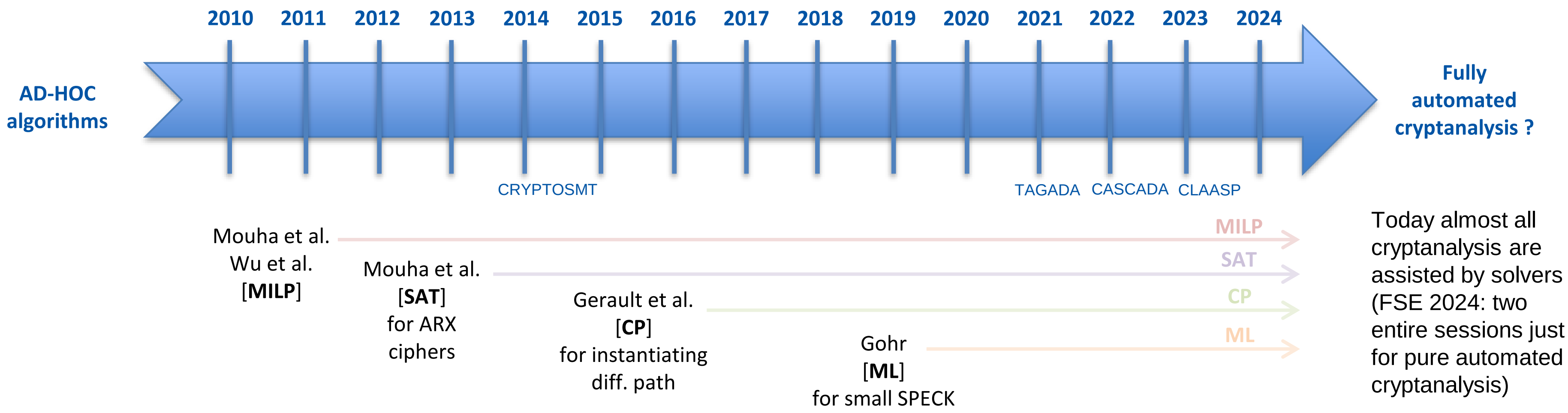
Open Cryptanalysis Platform (OCP)

A New Collaborative Effort for Automated Cryptanalysis

Thomas Peyrin – Chunning Zhou
NTU Singapore

SKCAM 2025 - Roma
15th March 2025

Timeline of Automated Cryptanalysis



Automated cryptanalysis using declarative frameworks (SAT/MILP/CP/etc.) is generally slower or at best same as ad-hoc tools, but so much **more convenient**

Mainly on **differential** and **linear cryptanalysis**, but now also on integral distinguishers, cube attacks, meet-in-the-middle attacks, etc.

Solving time is a crucial aspect and can be impacted by:

- the framework you use (SAT/MILP/CP/etc.)
- the strategy of modeling (many works on various modeling strategies)
- the solver (less contributions on that, different research field)
- the type of problem studied / scale

Open Cryptanalysis Platform

(Open-CP)

Open-CP: a new collaborative cryptanalysis platform

- Automatic generation of attacks / implementations
- In collaboration with all cryptanalysts willing to join !
- **Free** and **open source**
- **Easy** to use and contribute (define a common language we all understand)
- Very modular, allows custom operators, etc.
- Easy to install (no or very minimal use of external tools)
- Start simple (differential)
- **Goal:** become the go-to platform for creating / testing / benchmarking cryptanalysis

https://github.com/Open-CP/OCP

How to Define a Crypto Scheme ?

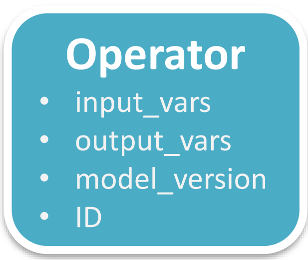
First step: define a common language we all agree with and understand. Once we have that, a LOT of things are possible.

How do we **describe** a cipher in a generic way and have this language usable for our global cryptanalysis platform ?

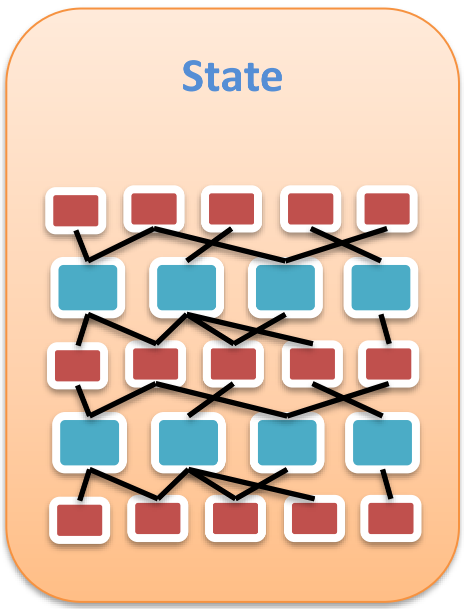
Current Architecture of Open-CP

- Coded in **Python**: everything is an object and can have its own modeling
- Internal representation: a **multipartite directed acyclic graph** – similar to TAGADA and CLAASP

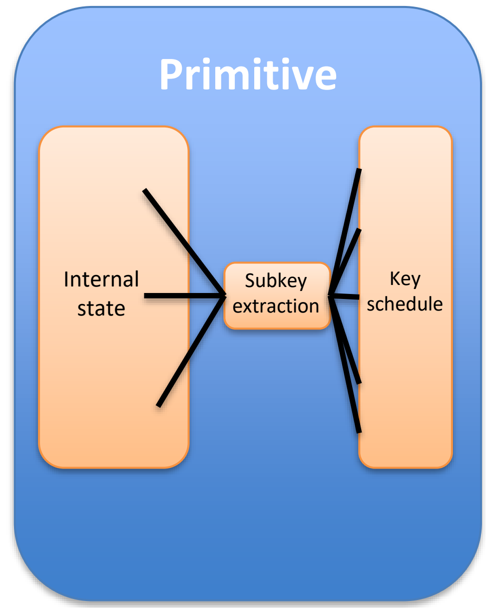
Ex: 1 bit,
8/32/64-bit words



Ex: equal, Sbox,
XOR/OR/AND,
matrix multiplication,
modular addition, …



Ex: Simon internal permutation
Simon key schedule



Ex: Simon block cipher (1 state for internal state, 1 state for key schedule state, 1 state for subkey extraction)

State Architecture of Open-CP

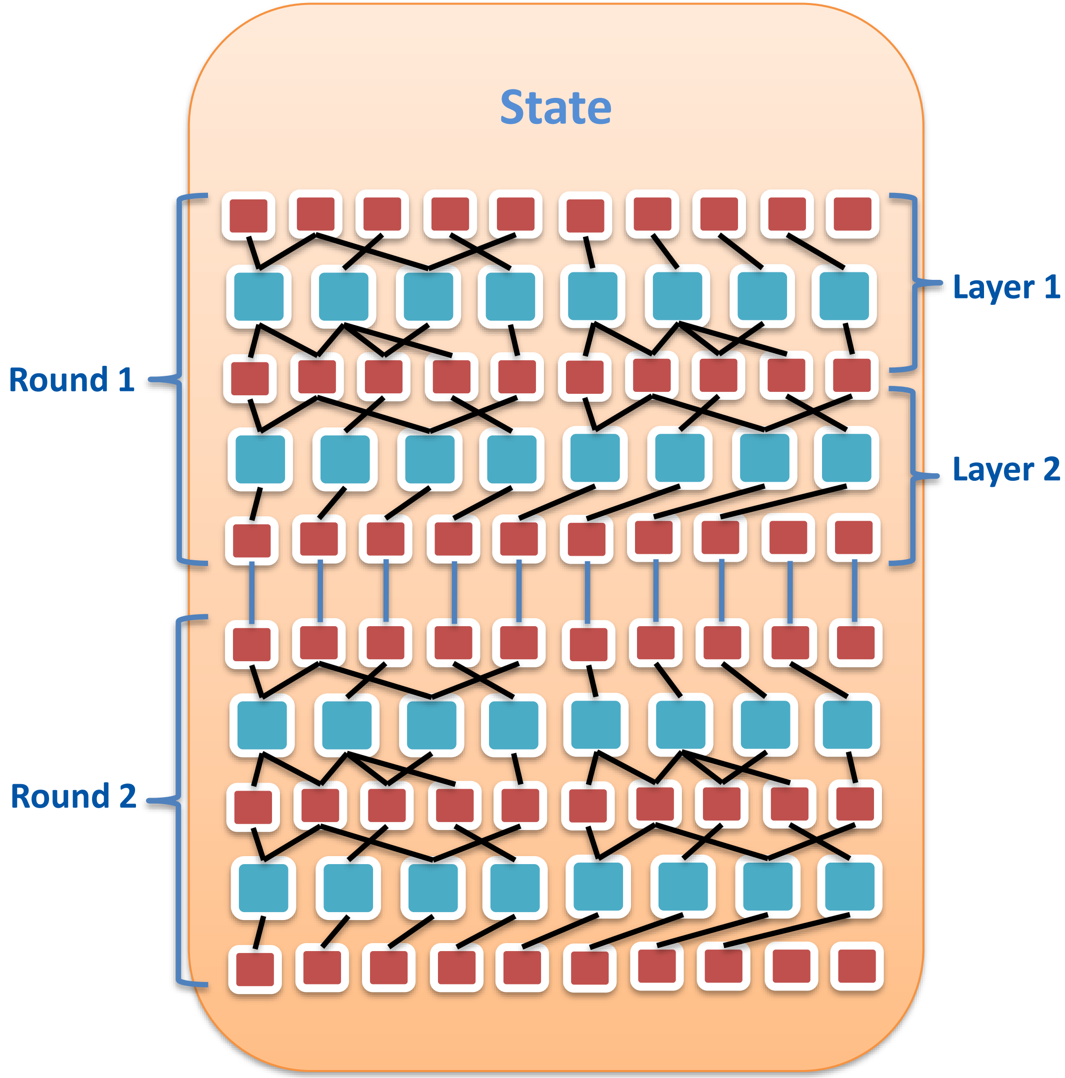
A **state** is represented by a **collection of rounds**, each round is composed of a **collection of layers**.

Advantage: this allows very **easy indexing** and **usage**

vars[r][l][i] for variable at round r, layer l, position i

Drawback: some redundant variables, but not a problem, we can clean after if needed

Ex: AES internal state has 10 rounds of 4 layers each (SB, SR, MC/ID, AC) with 16 variables of 8 bits each.



Easy and Fast cipher definition

Many functions already present to help you define a state very easily

SPECK

```
# The Speck internal permutation
class Speck_permutation(Permutation):
    def __init__(self, name, version, s_input, s_output, nbr_rounds=None, represent_mode=0):

        # define the parameters
        p_bitsize = version
        if nbr_rounds==None: nbr_rounds=22 if version==32 else 22 if version==48 else 26 if version==64 else 28 if version==96 else 32 if version==128 else None
        if represent_mode==0: nbr_layers, nbr_words, nbr_temp_words, word_bitsize = 4, 2, 0, p_bitsize>>1
        super().__init__(name, s_input, s_output, nbr_rounds, [nbr_layers, nbr_words, nbr_temp_words, word_bitsize])
        S = self.states["STATE"]
        rotr, rotl = (7, 2) if version == 32 else (8, 3)

        # create constraints
        if represent_mode==0:
            for i in range(1,nbr_rounds+1):
                S.RotationLayer("ROT1", i, 0, ['r', rotr, 0]) # Rotation layer
                S.SingleOperatorLayer("ADD", i, 1, op.ModAdd, [[0,1]], [0]) # Modular addition layer
                S.RotationLayer("ROT2", i, 2, ['l', rotl, 1]) # Rotation layer
                S.SingleOperatorLayer("XOR", i, 3, op.bitwiseXOR, [[0,1]], [1]) # XOR layer
```

SKINNY

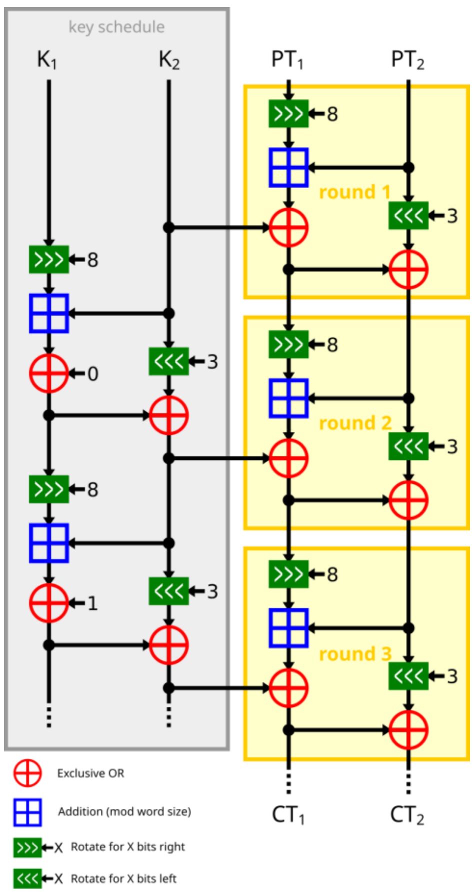
```
# The Skinny internal permutation
class Skinny_permutation(Permutation):
    def __init__(self, name, version, s_input, s_output, nbr_rounds=None, represent_mode=0):

        # define the parameters
        p_bitsize = version
        if nbr_rounds==None: nbr_rounds=32 if version==64 else 40 if version==128 else None
        if represent_mode==0: nbr_layers, nbr_words, nbr_temp_words, word_bitsize = 4, 16, 0, int(p_bitsize/16)
        super().__init__(name, s_input, s_output, nbr_rounds, [nbr_layers, nbr_words, nbr_temp_words, word_bitsize])
        round_constants = self.gen_rounds_constant_table()
        sbox = op.Skinny_4bit_Sbox if word_bitsize==4 else op.Skinny_8bit_Sbox

        S = self.states["STATE"]

        # create constraints
        if represent_mode==0:
            for i in range(1,nbr_rounds+1):
                S.SboxLayer("SB", i, 0, sbox)
                S.AddConstantLayer("C", i, 1, "xor", [True,None,None,None, True,None,None,None, True], round_constants)  # Constant layer
                S.PermutationLayer("SR", i, 2, [0,1,2,3, 7,4,5,6, 10,11,8,9, 13,14,15,12]) # Shiftrows layer
                S.MatrixLayer("MC", i, 3, [[1,0,1,1], [1,0,0,0], [0,1,1,0], [1,0,1,0]], [[0,4,8,12], [1,5,9,13], [2,6,10,14], [3,7,11,15]])  #Mixcolumns layer
```

Modeling Example of Open-CP

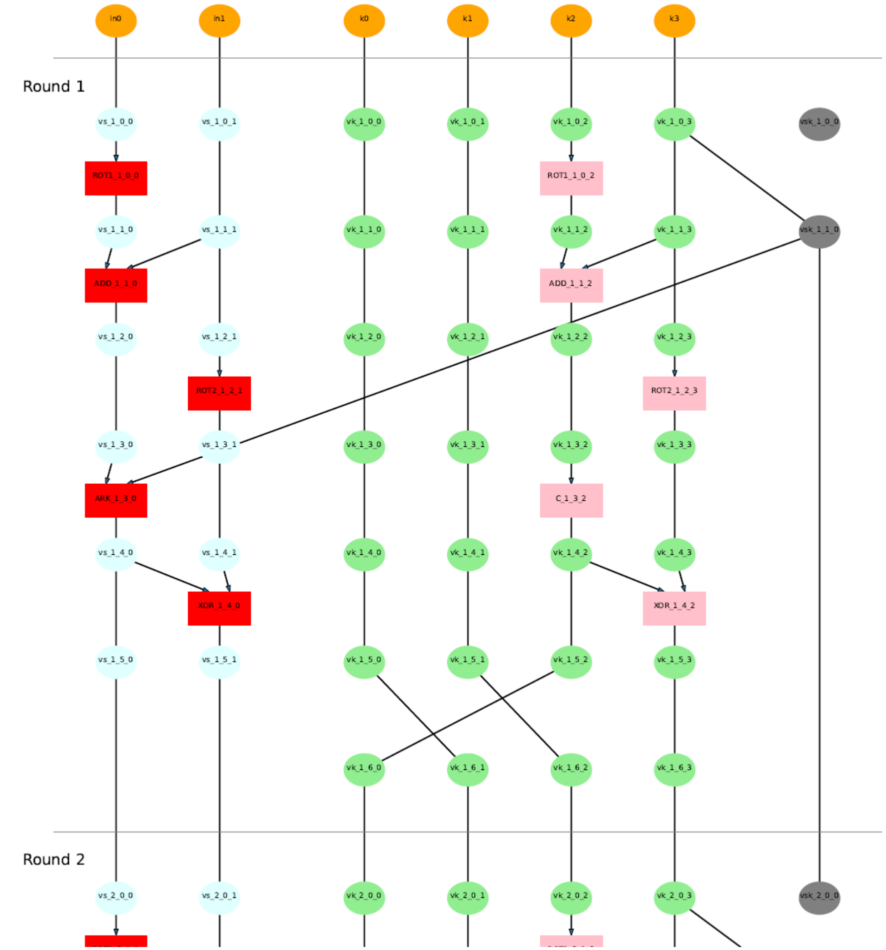
variables

operators

Example: SPECK-32 block cipher



(image from Wikipedia)

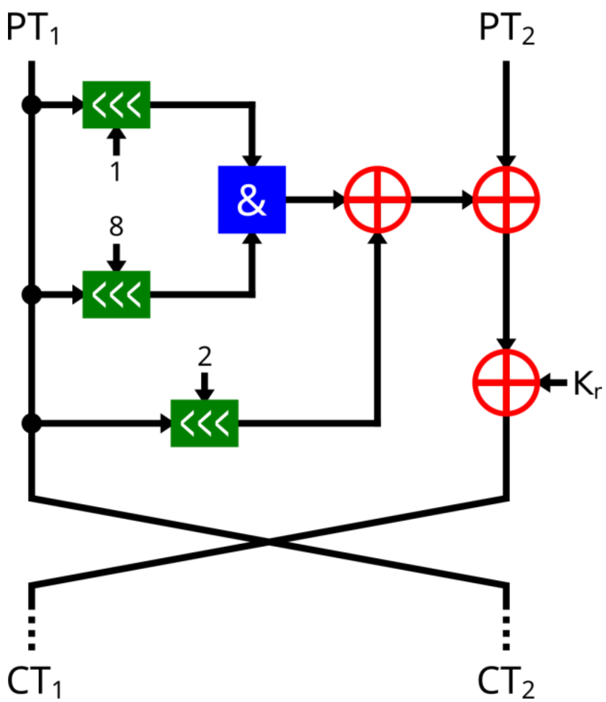


Modeling Example of Open-CP

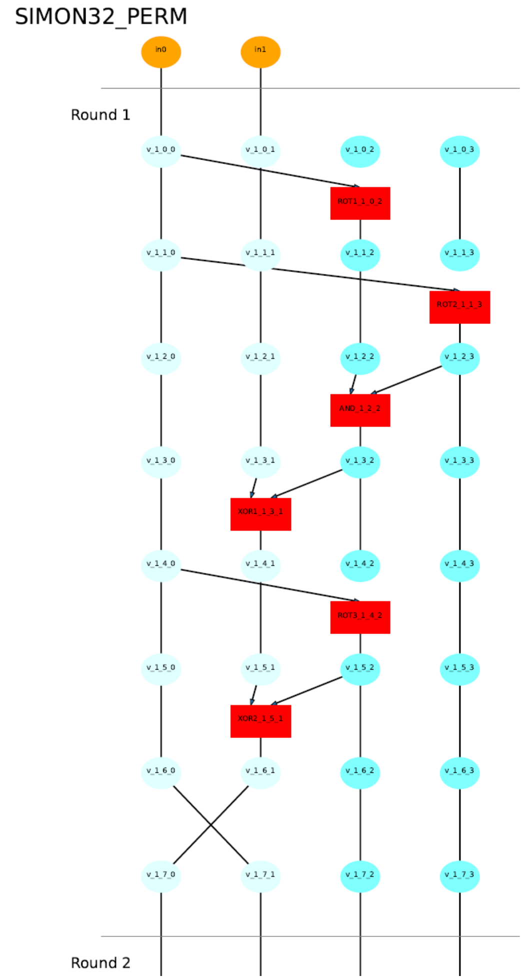
variables

operators

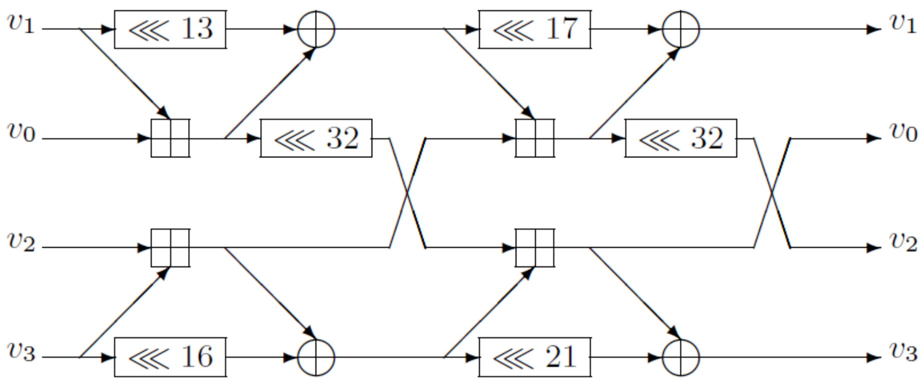
Example: SIMON-32 permutation



(image from Wikipedia)

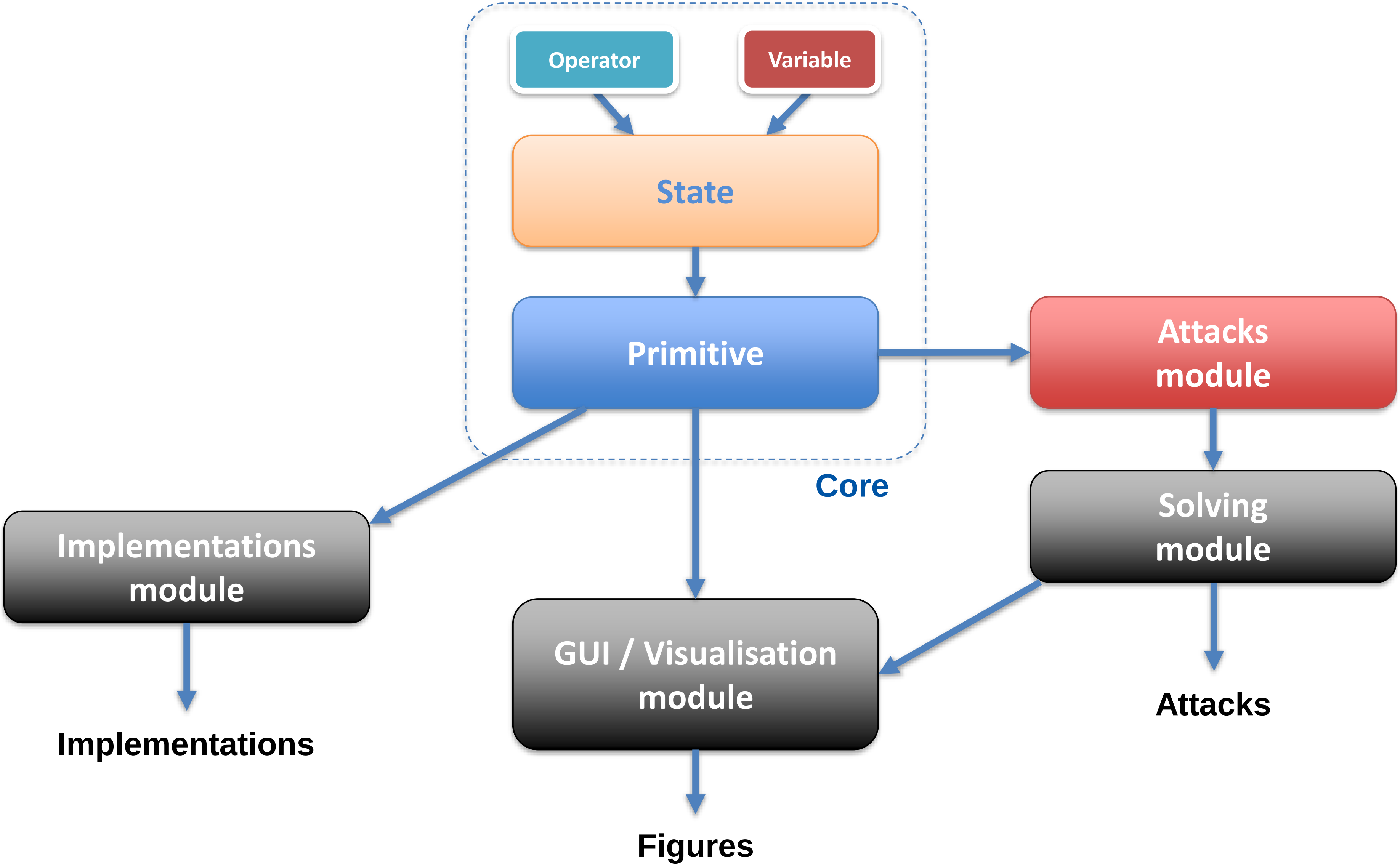


Ex: SipHash




```python
# The SipHash internal permutation
class SipHash_permutation(Permutation):
    def __init__(self, name, s_input, s_output, nbr_rounds=None, represent_mode=0):
        nbr_layers = 10
        nbr_words = 4
        nbr_temp_words = 0
        word_bitsize = 64
        super().__init__(name, s_input, s_output, nbr_rounds, [nbr_layers, nbr_words, nbr_temp_words, word_bitsize])
        S = self.states["STATE"]

        # create constraints
        if represent_mode==0:
            for i in range(1,nbr_rounds+1):
                S.SingleOperatorLayer("ADD1", i, 0, op.ModAdd, [[0,1], [2,3]], [0, 2]) # Modular addition layer
                S.RotationLayer("ROT1", i, 1, [['l', 13, 1], ['l', 16, 3]]) # Rotation layer
                S.SingleOperatorLayer("XOR1", i, 2, op.bitwiseXOR, [[0,1], [2,3]], [1, 3]) # XOR layer
                S.RotationLayer("ROT2", i, 3, [['l', 32, 0]]) # Rotation layer
                S.PermutationLayer("PERM1", i, 4, [2,1,0,3]) # Permutation layer
                S.SingleOperatorLayer("ADD2", i, 5, op.ModAdd, [[0,1], [2,3]], [0, 2]) # Modular addition layer
                S.RotationLayer("ROT3", i, 6, [['l', 17, 1], ['l', 21, 3]]) # Rotation layer
                S.SingleOperatorLayer("XOR2", i, 7, op.bitwiseXOR, [[0,1], [2,3]], [1, 3]) # XOR layer
                S.RotationLayer("ROT4", i, 8, [['l', 32, 0]]) # Rotation layer
                S.PermutationLayer("PERM2", i, 9, [2,1,0,3]) # Permutation layer
```

Architecture of Open-CP Tool



Automatic Generation of C / Python code

```python
#Rotation Macros
def ROTL(n, d, bitsize): return ((n << d) | (n >> (bitsize - d))) & (2**bitsize - 1)
def ROTR(n, d, bitsize): return ((n >> d) | (n << (bitsize - d))) & (2**bitsize - 1)

# Function implementing the SIMON32_PERM function
# Input:
#   IN: a list of 2 words of 16 bits
# Output:
#   OUT: a list of 2 words of 16 bits
def SIMON32_PERM(IN, OUT):

    # Input
    v_0_0 = IN[0]
    v_0_1 = IN[1]
    v_0_2 = v_0_3 = 0

    # Round function
    for i in range(10):
        v_1_0 = v_0_0
        v_1_1 = v_0_1
        v_1_2 = ROTL(v_0_0, 1, 16)
        v_1_3 = v_0_3
        v_2_0 = v_1_0
        v_2_1 = v_1_1
        v_2_2 = v_1_2
        v_2_3 = ROTL(v_1_0, 8, 16)
        v_3_0 = v_2_0
        v_3_1 = v_2_1
        v_3_2 = v_2_2 & v_2_3
        v_3_3 = v_2_3
        v_4_0 = v_3_0
        v_4_1 = v_3_1 ^ v_3_2
        v_4_2 = v_3_2
        v_4_3 = v_3_3
        v_5_0 = v_4_0
        v_5_1 = v_4_1
        v_5_2 = ROTL(v_4_0, 2, 16)
        v_5_3 = v_4_3
        v_6_0 = v_5_0
        v_6_1 = v_5_1 ^ v_5_2
        v_6_2 = v_5_2
        v_6_3 = v_5_3
        v_7_0 = v_6_1
        v_7_1 = v_6_0
        v_7_2 = v_6_2
        v_7_3 = v_6_3
        v_0_0 = v_7_0
        v_0_1 = v_7_1
        v_0_2 = v_7_2
        v_0_3 = v_7_3

    # Output
    OUT[0] = v_7_0
    OUT[1] = v_7_1

# test implementation
IN = [0x0, 0x0]
OUT = [0x0, 0x0]
SIMON32_PERM(IN, OUT)
print('IN', str([hex(i) for i in IN]))
print('OUT', str([hex(i) for i in OUT]))
```

C (rolled/unrolled)

```c
#include <stdint.h>
#include <stdio.h>

//Rotation Macros
#define ROTL(n, d, bitsize) (((n << d) | (n >> (bitsize - d))) & ((1<<bitsize) - 1))
#define ROTR(n, d, bitsize) (((n >> d) | (n << (bitsize - d))) & ((1<<bitsize) - 1))

// Function implementing the SIMON32_PERM function
// Input:
//   IN: an array of 2 words of 16 bits
// Output:
//   OUT: an array of 2 words of 16 bits
void SIMON32_PERM(uint32_t* IN, uint32_t* OUT){
    uint32_t v_0_0, v_0_1, v_0_2, v_0_3, v_1_0, v_1_1, v_1_2, v_1_3, v_2_0, v_2_1, v_2_2, v_2_3,

    // Input
    v_0_0 = IN[0];
    v_0_1 = IN[1];

    // Round function
    for (int i=0; i<10; i++) {
        v_1_0 = v_0_0;
        v_1_1 = v_0_1;
        v_1_2 = ROTL(v_0_0, 1, 16);
        v_1_3 = v_0_3;
        v_2_0 = v_1_0;
        v_2_1 = v_1_1;
        v_2_2 = v_1_2;
        v_2_3 = ROTL(v_1_0, 8, 16);
        v_3_0 = v_2_0;
        v_3_1 = v_2_1;
        v_3_2 = v_2_2 & v_2_3;
        v_3_3 = v_2_3;
        v_4_0 = v_3_0;
        v_4_1 = v_3_1 ^ v_3_2;
        v_4_2 = v_3_2;
        v_4_3 = v_3_3;
        v_5_0 = v_4_0;
        v_5_1 = v_4_1;
        v_5_2 = ROTL(v_4_0, 2, 16);
        v_5_3 = v_4_3;
        v_6_0 = v_5_0;
        v_6_1 = v_5_1 ^ v_5_2;
        v_6_2 = v_5_2;
        v_6_3 = v_5_3;
        v_7_0 = v_6_1;
        v_7_1 = v_6_0;
        v_7_2 = v_6_2;
        v_7_3 = v_6_3;
        v_0_0 = v_7_0;
        v_0_1 = v_7_1;
        v_0_2 = v_7_2;
        v_0_3 = v_7_3;

    }

    // Output
    OUT[0] = v_7_0;
    OUT[1] = v_7_1;
}
```

Python (rolled/unrolled)

Automatic Generation of SAT / MILP models

```
-1802 1818 -1834 -1803 -1819 -1835 0
-1802 -1818 1834 -1803 -1819 -1835 0
1803 1819 -1835 1804 1820 1836 0
1803 -1819 1835 1804 1820 1836 0
-1803 1819 1835 1804 1820 1836 0
-1803 -1819 -1835 1804 1820 1836 0
1803 1819 1835 -1804 -1820 -1836 0
1803 -1819 -1835 -1804 -1820 -1836 0
-1803 1819 -1835 -1804 -1820 -1836 0
-1803 -1819 1835 -1804 -1820 -1836 0
1804 1820 -1836 1805 1821 1837 0
1804 -1820 1836 1805 1821 1837 0
-1804 1820 1836 1805 1821 1837 0
-1804 -1820 -1836 1805 1821 1837 0
1804 1820 1836 -1805 -1821 -1837 0
1804 -1820 -1836 -1805 -1821 -1837 0
-1804 1820 -1836 -1805 -1821 -1837 0
-1804 -1820 1836 -1805 -1821 -1837 0
1805 1821 -1837 1806 1822 1838 0
1805 -1821 1837 1806 1822 1838 0
-1805 1821 1837 1806 1822 1838 0
-1805 -1821 -1837 1806 1822 1838 0
1805 1821 1837 -1806 -1822 -1838 0
1805 -1821 -1837 -1806 -1822 -1838 0
-1805 1821 -1837 -1806 -1822 -1838 0
-1805 -1821 1837 -1806 -1822 -1838 0
1806 1822 -1838 1807 1823 1839 0
1806 -1822 1838 1807 1823 1839 0
-1806 1822 1838 1807 1823 1839 0
-1806 -1822 -1838 1807 1823 1839 0
1806 1822 1838 -1807 -1823 -1839 0
1806 -1822 -1838 -1807 -1823 -1839 0
```

SAT

```
vs_3_1_0_10 - vs_3_1_1_10 + ADD_3_1_0_p_9 >= 0
vs_3_2_0_10 - vs_3_1_0_10 + ADD_3_1_0_p_9 >= 0
vs_3_1_0_10 + vs_3_1_1_10 + vs_3_2_0_10 + ADD_3_1_0_p_9 <= 3
vs_3_1_0_10 + vs_3_1_1_10 + vs_3_2_0_10 - ADD_3_1_0_p_9 >= 0
vs_3_1_0_9 + vs_3_1_1_9 + vs_3_2_0_9 + ADD_3_1_0_p_9 - vs_3_1_1_10 >= 0
vs_3_1_1_10 + vs_3_1_0_9 - vs_3_1_1_9 + vs_3_2_0_9 + ADD_3_1_0_p_9 >= 0
vs_3_1_1_10 - vs_3_1_0_9 + vs_3_1_1_9 + vs_3_2_0_9 + ADD_3_1_0_p_9 >= 0
vs_3_1_0_10 + vs_3_1_0_9 + vs_3_1_1_9 - vs_3_2_0_9 + ADD_3_1_0_p_9 >= 0
vs_3_2_0_10 - vs_3_1_0_9 - vs_3_1_1_9 - vs_3_2_0_9 + ADD_3_1_0_p_9 >= -2
vs_3_1_0_9 - vs_3_1_1_10 - vs_3_1_1_9 - vs_3_2_0_9 + ADD_3_1_0_p_9 >= -2
vs_3_1_1_9 - vs_3_1_1_10 - vs_3_1_0_9 - vs_3_2_0_9 + ADD_3_1_0_p_9 >= -2
vs_3_2_0_9 - vs_3_1_1_10 - vs_3_1_0_9 - vs_3_1_1_9 + ADD_3_1_0_p_9 >= -2
vs_3_1_1_11 - vs_3_2_0_11 + ADD_3_1_0_p_10 >= 0
vs_3_1_0_11 - vs_3_1_1_11 + ADD_3_1_0_p_10 >= 0
vs_3_2_0_11 - vs_3_1_0_11 + ADD_3_1_0_p_10 >= 0
vs_3_1_0_11 + vs_3_1_1_11 + vs_3_2_0_11 + ADD_3_1_0_p_10 <= 3
vs_3_1_0_11 + vs_3_1_1_11 + vs_3_2_0_11 - ADD_3_1_0_p_10 >= 0
vs_3_1_0_10 + vs_3_1_1_10 + vs_3_2_0_10 + ADD_3_1_0_p_10 - vs_3_1_1_11 >= 0
vs_3_1_1_11 + vs_3_1_0_10 - vs_3_1_1_10 + vs_3_2_0_10 + ADD_3_1_0_p_10 >= 0
vs_3_1_1_11 - vs_3_1_0_10 + vs_3_1_1_10 + vs_3_2_0_10 + ADD_3_1_0_p_10 >= 0
vs_3_1_0_11 + vs_3_1_0_10 + vs_3_1_1_10 - vs_3_2_0_10 + ADD_3_1_0_p_10 >= 0
vs_3_2_0_11 - vs_3_1_0_10 - vs_3_1_1_10 - vs_3_2_0_10 + ADD_3_1_0_p_10 >= -2
vs_3_1_0_10 - vs_3_1_1_11 - vs_3_1_1_10 - vs_3_2_0_10 + ADD_3_1_0_p_10 >= -2
vs_3_1_1_10 - vs_3_1_1_11 - vs_3_1_0_10 - vs_3_2_0_10 + ADD_3_1_0_p_10 >= -2
vs_3_2_0_10 - vs_3_1_1_11 - vs_3_1_0_10 - vs_3_1_1_10 + ADD_3_1_0_p_10 >= -2
vs_3_1_1_12 - vs_3_2_0_12 + ADD_3_1_0_p_11 >= 0
vs_3_1_0_12 - vs_3_1_1_12 + ADD_3_1_0_p_11 >= 0
vs_3_2_0_12 - vs_3_1_0_12 + ADD_3_1_0_p_11 >= 0
vs_3_1_0_12 + vs_3_1_1_12 + vs_3_2_0_12 + ADD_3_1_0_p_11 <= 3
vs_3_1_0_12 + vs_3_1_1_12 + vs_3_2_0_12 - ADD_3_1_0_p_11 >= 0
vs_3_1_0_11 + vs_3_1_1_11 + vs_3_2_0_11 + ADD_3_1_0_p_11 - vs_3_1_1_12 >= 0
vs_3_1_1_12 + vs_3_1_0_11 - vs_3_1_1_11 + vs_3_2_0_11 + ADD_3_1_0_p_11 >= 0
vs_3_1_1_12 - vs_3_1_0_11 + vs_3_1_1_11 + vs_3_2_0_11 + ADD_3_1_0_p_11 >= 0
vs_3_1_0_12 + vs_3_1_0_11 + vs_3_1_1_11 - vs_3_2_0_11 + ADD_3_1_0_p_11 >= 0
vs_3_2_0_12 - vs_3_1_0_11 - vs_3_1_1_11 - vs_3_2_0_11 + ADD_3_1_0_p_11 >= -2
vs_3_1_0_11 - vs_3_1_1_12 - vs_3_1_1_11 - vs_3_2_0_11 + ADD_3_1_0_p_11 >= -2
vs_3_1_1_11 - vs_3_1_1_12 - vs_3_1_0_11 - vs_3_2_0_11 + ADD_3_1_0_p_11 >= -2
vs_3_2_0_11 - vs_3_1_1_12 - vs_3_1_0_11 - vs_3_1_1_11 + ADD_3_1_0_p_11 >= -2
vs_3_1_1_13 - vs_3_2_0_13 + ADD_3_1_0_p_12 >= 0
```

MILP

One operator – several modelisations

Current Operators: Equal, NOT, AND, OR, XOR, NXOR, Sbox, Modular Addition, Rotation/Shift, Matrix multiplication, Constant Addition

Soon to be added: AES round, Modular multiplication

One operator can have several types of modelisation !

Ex1: with an 8-bit variable, we can track exact or truncated differences

Ex2: Modular Addition can have different strategies of modeling

In OCP, **you can switch any operator from one type of modelisation to another** very easily (simply change a flag).

=> You can have entire modeling strategies where some operators behave differently than others (Ex: SHA attacks with linear/non linear)

Ciphers already implemented

- AES
- Ascon
- GIFT
- ROCCA
- Simon
- SipHash
- Skinny
- Speck
- ChaCha (probably this week ☺)
- more to come soon …

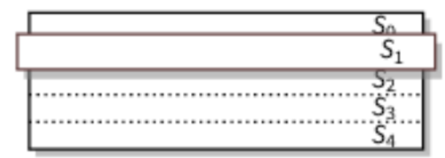
One primitive – several modelisations

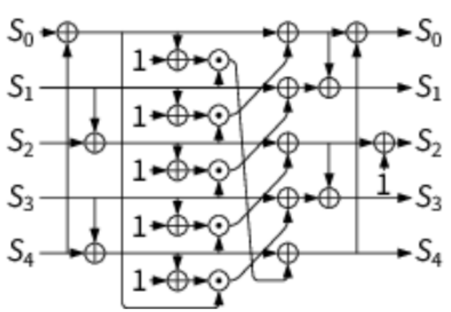
One cipher can have several modelisations !

<u>Ex1:</u> ASCON state with 5 64-bit variables or 320 1-bit variables

<u>Ex2:</u> SuperSbox representation for a few rounds

In OCP, **you can choose one type of modelisation** very easily (simply set a flag).





$$S_0 := S_0 \oplus (S_0 \ggg 19) \oplus (S_0 \ggg 28)$$
$$S_1 := S_1 \oplus (S_1 \ggg 61) \oplus (S_1 \ggg 39)$$
$$S_2 := S_2 \oplus (S_2 \ggg 1) \oplus (S_2 \ggg 6)$$
$$S_3 := S_3 \oplus (S_3 \ggg 10) \oplus (S_3 \ggg 17)$$
$$S_4 := S_4 \oplus (S_4 \ggg 7) \oplus (S_4 \ggg 41)$$

ASCON round

OCP Current State

Current capabilities of OCP:

- Differential cryptanalysis (single or related key)
- Truncated differential cryptanalysis (single or related key)
- Implementations automatically output test vectors, makes sure your entire model is correct

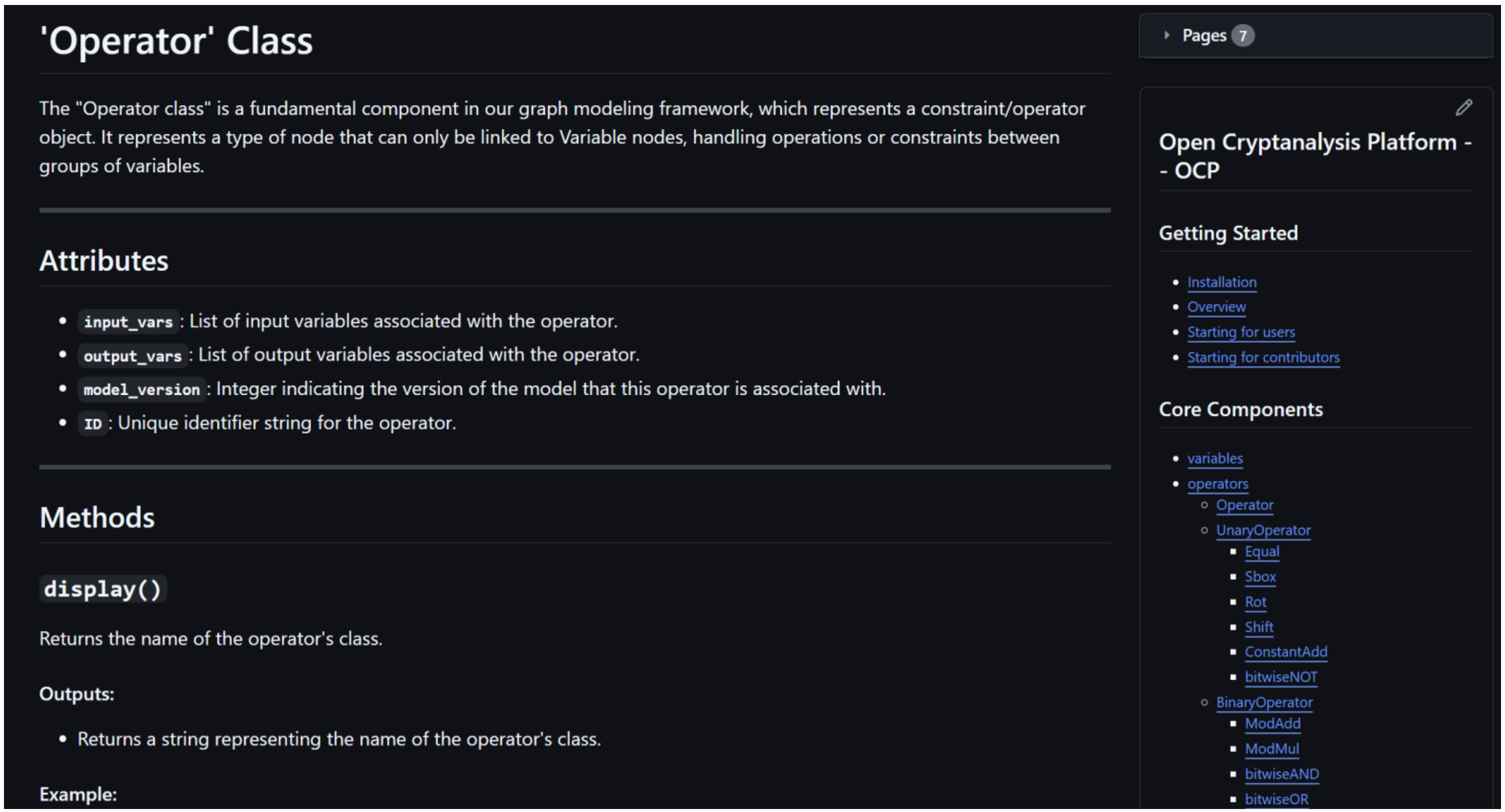
Coming soon: linear cryptanalysis, differential-linear cryptanalysis, additional modeling improvements (Matsui branch and bound option), additional modeling of the operators (window heuristic for mod addition, etc.)

Help us include more attacks / models !

Issue: inverse of primitives ?

Documentation

https://github.com/Open-CP/OCP/wiki



'Operator' Class

The "Operator class" is a fundamental component in our graph modeling framework, which represents a constraint/operator object. It represents a type of node that can only be linked to Variable nodes, handling operations or constraints between groups of variables.

Attributes

- `input_vars` : List of input variables associated with the operator.
- `output_vars` : List of output variables associated with the operator.
- `model_version` : Integer indicating the version of the model that this operator is associated with.
- `ID` : Unique identifier string for the operator.

Methods

`display()`

Returns the name of the operator's class.

Outputs:

- Returns a string representing the name of the operator's class.

Example:

Pages 7

Open Cryptanalysis Platform -- OCP

Getting Started

Core Components

Documentation (Quick Start for Users)

https://github.com/Open-CP/OCP/wiki

Quick Start for Users

This guide provides a detailed, step-by-step tutorial on using OCP to implement various functionalities, including Python and C code implementations, as well as MILP and SAT models. All these features are integrated within `OCP.py`.

Implementation of Primitives

Take Speck block cipher as an example, we demonstrate the implementation of cryptographic primitives within OCP. This includes variable initialization, cipher instantiation, Python and C code generation, and a visual figure of the primitive.

1. Import the necessary modules and instantiate the speck block cipher.

```
import os
import primitives.speck as speck
import variables.variables as var
import implementations.implementations as imp
import visualisations.visualisations as vis

r = None
p_bitsize, k_bitsize, word_size, m = 32, 64, 16, 4
my_plaintext, my_key, my_ciphertext = [var.Variable(word_size,ID="in"+str(i)) for i in range(2)], [var.Variable(word_s
cipher = speck.Speck_block_cipher(f"SPECK{p_bitsize}_{k_bitsize}", (p_bitsize, k_bitsize), my_plaintext, my_key, my_ci
```

2. Generate and save Python and C code to implement the primitive encryption in both standard and unrolled versions.

```
# Ensure the directory for storing files exists
os.makedirs("files", exist_ok=True)

# Generate Python and C code files
imp.generate_implementation(cipher,"files/" + cipher.name + ".py", "python")
imp.generate_implementation(cipher,"files/" + cipher.name + "_unrolled.py", "python", True)
imp.generate_implementation(cipher,"files/" + cipher.name + ".c", "c")
imp.generate_implementation(cipher,"files/" + cipher.name + "_unrolled.c", "c", True)
```

Pages 8

Open Cryptanalysis Platform - - OCP

Getting Started

Core Components

Primitives

Documentation (Quick Start for Contributors)

https://github.com/Open-CP/OCP/wiki

Quick Start for Contributors

This guide provides instructions for contributing to OCP.

Contribution to Operators

1. **Adding New Operators**

- Define a new operator class in `operators/operators.py`.
- Implement the operator's functionality and modeling methods in alignment with existing operators.

2. **Extending Existing Operators**

- Add new Python and C implementations to optimize performance in `generate_implementation()`.
- Add new modeling techniques (e.g., "MILP", "SAT") by specifying `model_type` in `generate_model()`
- Add new modeling versions (e.g., "diff", "truncated_diff", "linear") for various cryptanalysis techniques by specifying `model_version` in `generate_model()`.

Contribution to Primitives

1. **Adding New Primitives**

- Create a `new_primitive.py` file in `primitives/`.

Pages 9

Open Cryptanalysis Platform - - OCP

Getting Started

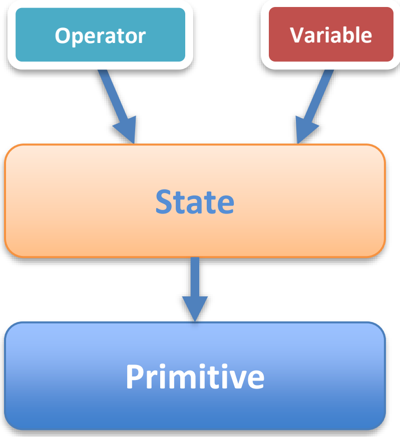
- Installation
- Overview
- Starting for users
- Starting for contributors

Core Components

- variables
- operators
  - Operator
  - UnaryOperator
    - Equal
    - Sbox
    - Rot
    - Shift
    - ConstantAdd
    - bitwiseNOT
  - BinaryOperator
    - ModAdd
    - ModMul

Main TODOs: Core and Attack Module

- **Core OCP:**
  - Growing the pre-existing **library of ciphers**
  - Growing the pre-existing **library of operators**
  - Growing the number of **different models/representations** for each operators/ciphers
  - Conversions between variable types ?
  - Operating modes ?



- **Attacks module:**
  - **More attack types** ! (linear / differential-linear / boomerang / impossible diff / division property / etc.)
  - **Key recovery** phase (started incorporating Autoguess)
  - Pre-existing **library of attacks**
  - **Standardized benchmarks** for comparing attacks
  - Allow **modular combination of attacks/models**
  - **not** limited to MILP and SAT ! Ad-hoc algorithms also welcome

Main TODOs: Other Modules

- **Implementations module:**
  - **VHDL**, Rust, other use cases?
  - Automatic verification of **test vectors** for new representations
  - Faster C implementations
  - Optimized Sbox / Diffusion matrix **implementations database**
  - Long-term future: side-channels resistant implementations

Implementations module

- **Solving module:**
  - CP ? ML ? Others ?
  - More MILP and SAT solvers
  - Parallelization

Solving module

- **Visualisation module:**
  - Automated visualisations of the attacks
  - Graphical interface for user interaction (cipher design / attack config.)
  - **Automated generation of LaTeX/TikZ figures !!!**

GUI / Visualisation module

OCP - Organisation

We need to establish (simple) **governance** to have proper:

- development processes into place,
- responsibilities,
- communication, regular meetings, valuable feedback.

One person responsible for each (sub)module ?

Philosophy:

- Remain Free and Open Source (careful if you copy code!)
- Don’t rely on external tools unless really necessary (keep install simple)
- Keep code clean and compact
- Keep code generic !

We want YOU !

If interested to contribute / getting updates:

- contact thomas.peyrin@ntu.edu.sg
- or join the googlegroup

automated-cryptanalysis@googlegroups.com

- or click on this link:

https://groups.google.com/g/automated-cryptanalysis

- GitHub:

https://github.com/Open-CP/OCP

Thanks to all the cryptanalysts that already joined the list and gave feedback !

Thank You !

Grazie !