circkit

CryptoExperts

19 Aug 22 at 17:15:48

TUTORIALS

Docs build on 19 Aug 22 at 17:09:17, commit 28f133c0b43d398f7c27c00a5368566aad0b3eba

`circkit` is a small *framework* for defining, constructing and manipulating *computational circuits*. It aims to be very generic, supporing both low-level circuits (e.g. bit/word-based operations or arithmetic circuits over a ring) and high-level circuits (i.e., with gates defining custom non-primitive functions).

CHAPTER ONE

INSTALLATION

```
$ pip3 install circkit
```

TODO: This does not work yet. When it works, we should remove `sys.path.append("../")` in the tutorials

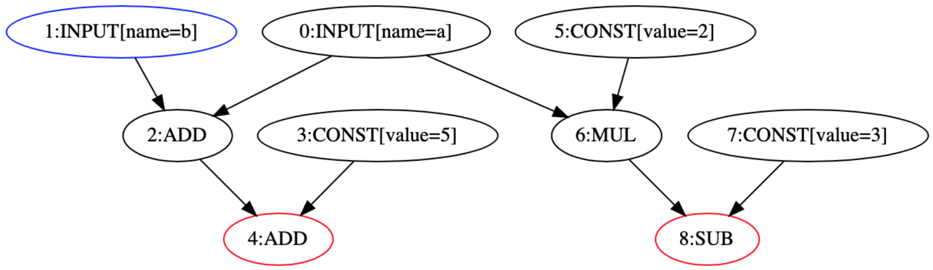
CHAPTER

TWO

QUICK EXAMPLE

```python
from circkit.arithmetic import ArithmeticCircuit

# Step 1: Initialize a new arithmetic circuit
C = ArithmeticCircuit()

# Step 2: Define the input nodes
a = C.add_input("a")
b = C.add_input("b")

# Step 3: Perform the computation
x = a + b + 5
y = 2 * a - 3

# Step 4: Define the output nodes
C.add_output(x)
C.add_output(y)
# To see the graph of the circuit
C.digraph().view()

# Step 5: Evaluate the circuit
# For example, a = 7, b = 9
inp = [7, 9]
out = C.evaluate(inp)
print("Circuit output:")
print(f"x = {out[0]}")
print(f"y = {out[1]}")
```

```
Circuit output:
x = 21
y = 11
```




```
# Verify
a = 7
b = 9
x = a + b + 5
y = 2 * a - 3
print("Verification:")
print(f"x = {x}")
print(f"y = {y}")
```

```
Verification:
x = 21
y = 11
```

2.1 How to build an arithmetic circuit

In this tutorial, we show you how to build an arithmetic circuit based on the `circkit` framework. In particular, we provide the instructions on:

1. *How to build a circuit*
2. *How to visualize a circuit*
3. *How to transform a circuit to a matrix*
4. *How to trace the intermediate values*

First of all, we need to add the path of the `circkit` folder to `sys.path`:

```
import sys
sys.path.append("..")
```

This tutorial sometimes uses `Sagemath`. You should run the code by `sage -python`. To do so, you can install a virual environment of `sage -python`, then run the code in that environment.

```
>>> sage -python -m venv --system-site-packages .venv
>>> source .venv
```

2.1.1 1. How to build a circuit

There are 5 steps to build a circuit.

Step 1: Initialize a new arithmetic circuit

Arithmetic circuit types

There are 2 types of arithmetic circuit that you can use:

- `ArithmeticCircuit`: This supports regular arithmetic operations (e.g., $+$, $-$, $*$, $/$, $\ldots$ See step 3 for more details)
- `OptArithmeticCircuit`: This inherits the `ArithmeticCircuit` type and additionally supports
  - caching operations and nodes
  - precomputing annihilator operations, e.g. a*0 = 0
  - precomputing identity operations, e.g. a*1 = a, a+0 = 0
  - precomputing constant operations

Here we just provide examples on the `ArithmeticCircuit` type. For the `OptArithmeticCircuit`, it is almost similar.

Base ring

In case the computation of the circuit takes place in a field, we can specify the ring when instantiating a new circuit. All constants in the circuit will then be automatically converted to values in the field. Let us take $GF(2^8)$ as an example.

```
from circkit.arithmetic import ArithmeticCircuit
from sage.all import GF
K = GF(2**8)

# Step 1: Initialize a new arithmetic circuit
C = ArithmeticCircuit(base_ring=K, name="AToyCircuit")
print("Circuit information:")
print(C)
```

```
Circuit information:
<ArithmeticCircuit 'AToyCircuit' in:0 out:0 nodes:0>
```

By default, if we do not specify a base ring for the circuit, the operations of the circuit will take place in decimal numbers (see the very first example).

Step 2: Define the input nodes

We can define the input nodes of the circuit by one of the two following methods:

- `add_input`: this creates an input node. We use this method to create the input nodes one by one. Note that the name of a node (e.g., `inp_0`, `inp_1` in the example below) is a mandatory argument.

```
from circkit.arithmetic import ArithmeticCircuit
from sage.all import GF
K = GF(2**8)

# Step 1: Initialize a new arithmetic circuit
C = ArithmeticCircuit(base_ring=K, name="AToyCircuit")

# Step 2: Define the input nodes (one by one)
a = C.add_input("inp_0")
b = C.add_input("inp_1")
print("Circuit input nodes:")
print(C.inputs)
```

```
Circuit input nodes:
[<ArithmeticCircuit:INPUT[name=inp_0]#0 ()>, <ArithmeticCircuit:INPUT[name=inp_1]#1 ()>]
```

- `add_inputs`: this creates a list of input nodes. Note that the names of the nodes are specified by a format, e.g. `inp_%d` where `%d` is automatically replaced by a counter in $[0, n)$. You can see that the following example creates the same input nodes as the previous one.

```
from circkit.arithmetic import ArithmeticCircuit
from sage.all import GF
K = GF(2**8)

# Step 1: Initialize a new arithmetic circuit
C = ArithmeticCircuit(base_ring=K, name="AToyCircuit")

# Step 2: Define the input nodes (by a list)
inp_nodes = C.add_inputs(n=2, format="inp_%d")
print("Circuit input nodes:")
print(C.inputs)
```

```
Circuit input nodes:
[<ArithmeticCircuit:INPUT[name=inp_0]#0 ()>, <ArithmeticCircuit:INPUT[name=inp_1]#1 ()>]
```

Step 3: Perform the computation

Basic operations

Below are the built-in operations in `ArithmeticCircuit` and `OptArithmeticCircuit`. The operators of a operation can be nodes or constants.

Operation	Notation	Note
Addition	$+$	
Subtraction	$-$	
Multiplication	$*$	
Division	$/$	
Exponentiation	$**$	only support constant exponent
Inversion	$\sim$	only for base ring elements
Negation	$-$	unary operation for decimal only

```
from circkit.arithmetic import ArithmeticCircuit
from sage.all import GF
K = GF(2**8)

# Step 1: Initialize a new arithmetic circuit
C = ArithmeticCircuit(base_ring=K, name="AToyCircuit")

# Step 2: Define the input nodes (by a list)
inp_nodes = C.add_inputs(n=2, format="inp_%d")
a, b = inp_nodes

# Step 3: Perform the computations
x0 = a + b
x1 = x0 - 5
x2 = x1 * x0
x3 = x2 / 3
x4 = x3 ** 4
x5 = ~x4
print("Circuit information:")
print(C)
```

```
Circuit information:
<ArithmeticCircuit 'AToyCircuit' in:2 out:0 nodes:10>
```

Other operations

- Random (RND): In a circuit, we can create a random node which contains a random value. See the following example:

```
from circkit.arithmetic import ArithmeticCircuit
from sage.all import GF
K = GF(2**8)

# Step 1: Initialize a new arithmetic circuit
C = ArithmeticCircuit(base_ring=K, name="AToyCircuit")

# Step 2: Define the input nodes (by a list)
a = C.add_input("a")
b = C.add_input("b")

# Step 3: Perform the computations
```

(continues on next page)

(continued from previous page)

```
# x is a node holding a random value
x = C.RND()()
z = a + b + x
```

```
[161, 162]
```

- Lookup table (LUT): Given a node x and a table T of constants, this operation return a new node of value $T[x]$.

```
from circkit.arithmetic import ArithmeticCircuit
from sage.all import GF
K = GF(2**8)

# Step 1: Initialize a new arithmetic circuit
C = ArithmeticCircuit(base_ring=K, name="AToyCircuit")

# Step 2: Define the input nodes (by a list)
a = C.add_input("a")
b = C.add_input("b")

# Step 3: Perform the computations
T = (11, 22, 33, 44, 55)
T = tuple([K.fetch_int(v) for v in T])
x = C.LUT(T)(a)
y = C.LUT(T)(b)
# Or we can write
# x = a.lookup_in(T)
# y = b.lookup_in(T)
```

Step 4: Define the output nodes

`add_output` is the only method used to define output nodes. However, it can be used in two different ways:

- define the output nodes one by one as the following example

```
from circkit.arithmetic import ArithmeticCircuit
from sage.all import GF
K = GF(2**8)

# Step 1: Initialize a new arithmetic circuit
C = ArithmeticCircuit(base_ring=K, name="AToyCircuit")

# Step 2: Define the input nodes (by a list)
inp_nodes = C.add_inputs(n=2, format="inp_%d")
a, b = inp_nodes

# Step 3: Perform the computations
x0 = a + b
x1 = x0 - 5
x2 = x1 * x0
x3 = x2 / 3
x4 = x3 ** 4
```

(continues on next page)

(continued from previous page)

```
x5 = ~x4

# Step 4: Define the output nodes (one by one)
C.add_output(x4)
C.add_output(x5)
print("Circuit output nodes:")
print(C.outputs)
```

```
Circuit output nodes:
[<ArithmeticCircuit:EXP[power=4]#8 (7)>, <ArithmeticCircuit:INV#9 (8)>]
```

- define a list of output nodes as the following example.

```
from circkit.arithmetic import ArithmeticCircuit
from sage.all import GF
K = GF(2**8)

# Step 1: Initialize a new arithmetic circuit
C = ArithmeticCircuit(base_ring=K, name="AToyCircuit")

# Step 2: Define the input nodes (by a list)
inp_nodes = C.add_inputs(n=2, format="inp_%d")
a, b = inp_nodes

# Step 3: Perform the computations
x0 = a + b
x1 = x0 - 5
x2 = x1 * x0
x3 = x2 / 3
x4 = x3 ** 4
x5 = ~x4

# Step 4: Define the output nodes (one by one)
C.add_output([x4, x5])
print("Circuit output:")
print(C.outputs)
```

```
Circuit output:
[<ArithmeticCircuit:EXP[power=4]#8 (7)>, <ArithmeticCircuit:INV#9 (8)>]
```

Step 5: Evaluate the circuit

`evaluate(input: list, convert_input: bool, convert_output: bool)` is the method used to evaluate a circuit. It returns a list of output values corresponding to the output nodes. This method has 3 arguments:

- `input`: this is a list of input whose length equals to the number of input nodes.
- `convert_input`: this indicates that the input elements should be converted from decimal numbers to base ring elements or not
- `convert_output`: this indicates that the output elements should be converted from base ring elements to decimal numbers or not.

By default, `convert_input = True` and `convert_output = True`

```
from circkit.arithmetic import ArithmeticCircuit
from sage.all import GF
K = GF(2**8)

# Step 1: Initialize a new arithmetic circuit
C = ArithmeticCircuit(base_ring=K, name="AToyCircuit")

# Step 2: Define the input nodes (by a list)
inp_nodes = C.add_inputs(n=2, format="inp_%d")
a, b = inp_nodes

# Step 3: Perform the computations
x0 = a + b
x1 = x0 - 5
x2 = x1 * x0
x3 = x2 / 3
x4 = x3 ** 4
x5 = ~x4

# Step 4: Define the output nodes (one by one)
C.add_output([x4, x5])

# Step 5: Evaluate the circuit
inp = [7, 9]
out = C.evaluate(inp, convert_input=True, convert_output=False)
print("Circuit output:")
print(out)
```

```
Circuit output:
[z8^7 + z8^6 + z8^2 + z8 + 1, z8^7 + z8^4 + z8^3 + z8^2 + z8]
```

2.1.2 2. How to visualize a circuit

We use graphviz to visualize a circuit. Once we have a circuit, we can visualize it by calling the method `C.digraph().view()`

```
from circkit.arithmetic import ArithmeticCircuit
from sage.all import GF
K = GF(2**8)

# Step 1: Initialize a new arithmetic circuit
C = ArithmeticCircuit(base_ring=K, name="AToyCircuit")

# Step 2: Define the input nodes (by a list)
inp_nodes = C.add_inputs(n=2, format="inp_%d")
a, b = inp_nodes

# Step 3: Perform the computations
x0 = a + b
x1 = x0 - 5 + a
```

(continues on next page)

(continued from previous page)

```
x2 = x1 * x0
x3 = x2 / 3
x4 = x3 ** 4
x5 = ~x4

# Step 4: Define the output nodes (one by one)
C.add_output([x4, x5])

# Visualize the circuit (Run the code to see)
C.digraph().view()
```

```
'Digraph.gv.pdf'
```

2.1.3 3. How to transform a circuit to a matrix

An arithmetic circuit can be transformed to an affine $y = Ax + b$, where x is the input and y is the output of the circuit. It is a linear mapping when $b = 0$.

To be able to transform to an affine mapping, the operations of the circuit must be in the following set:

Operation	Notation	2 nodes	a node and a constant
Addition	$+$	Yes	Yes
Subtraction	$-$	Yes	Yes
Multiplication	$*$	No	Yes

```
from circkit.arithmetic import ArithmeticCircuit
from sage.all import GF, matrix, vector
K = GF(2**8)

# Step 1: Initialize a new arithmetic circuit
C = ArithmeticCircuit(base_ring=K, name="AToyCircuit")

# Step 2: Define the input nodes (by a list)
inp_nodes = C.add_inputs(n=2, format="inp_%d")
a, b = inp_nodes

# Step 3: Perform the computation
x0 = a + b
x1 = x0 * 19
x2 = x1 + x0
x3 = x2 * 3
x4 = x3 - x2
x5 = x1 + 2

# Step 4: Define the output nodes (one by one)
C.add_output([x4, x5])

# Transform to a matrix
A, b = C.to_matrix()
```

(continues on next page)

(continued from previous page)

```
print(f"A = {A}")
print(f"b = {b}")
```

```
A = [[z8^5 + z8^2, z8^5 + z8^2], [z8^4 + z8 + 1, z8^4 + z8 + 1]]
b = [0, z8]
```

Let us verify that the result of the computation $y = Ax + b$ is the same as the output of the circuit's evaluation.

```
# Verify
A = matrix(A)
b = vector(b)

inp = [15, 20]
out = C.evaluate(inp, convert_input=True, convert_output=False)

x = vector([K.fetch_int(v) for v in inp])
y = A*x + b

print("Circuit output:")
print(out)
print("Verifycation")
print(y)
print(f"circuit output = verification? {list(y) == out}")
```

```
Circuit output:
[z8^5 + z8^3 + z8 + 1, z8^7 + z8]
Verifycation
(z8^5 + z8^3 + z8 + 1, z8^7 + z8)
circuit output = verification? True
```

2.1.4 4. How to trace the intermediate values

Given an input, we can trace the values of each node in a circuit when evaluating the circuit. To do so, we use the function `trace(input: list, convert_input: bool, convert_values: bool, as_list: bool)`

- `input`: list of values fedding the input nodes
- `convert_input` (`True` by default): convert the input values from decimal to values on base ring
- `convert_values` (`True` by default): convert the intermediate values from base ring to decimal
- `as_list` (`False` by default): it returns a list of values when `as_list=True`. Otherwise, it displays the details of nodes and their corresponding values

```
from circkit.arithmetic import ArithmeticCircuit
from sage.all import GF, matrix, vector
K = GF(2**8)

# Step 1: Initialize a new arithmetic circuit
C = ArithmeticCircuit(base_ring=K, name="AToyCircuit")

# Step 2: Define the input nodes (by a list)
```

(continues on next page)

(continued from previous page)

```
inp_nodes = C.add_inputs(n=2, format="inp_%d")
a, b = inp_nodes

# Step 3: Perform the computation
x0 = a + b
x1 = x0 * 19
x2 = x1 + x0
x3 = x2 * 3
x4 = x3 - x2
x5 = x1 + 2

# Step 4: Define the output nodes (one by one)
C.add_output([x4, x5])

# Trace the intermediate values
inp = [15, 20]
T = C.trace(inp, convert_input=True, convert_values=True, as_list=False)
print("Trace information:")
print(T)

# Display the graph (Run the code to see)
C.digraph().view()
```

```
Trace information:
{<ArithmeticCircuit:INPUT[name=inp_0]#0 ()>: 15, <ArithmeticCircuit:INPUT[name=inp_1]#1␣
↪()>: 20, <ArithmeticCircuit:ADD#2 (0,1)>: 27, <ArithmeticCircuit:CONST[value=z8^4 + z8␣
↪+ 1]#3 ()>: 19, <ArithmeticCircuit:MUL#4 (2,3)>: 128, <ArithmeticCircuit:ADD#5 (4,2)>:␣
↪155, <ArithmeticCircuit:CONST[value=z8 + 1]#6 ()>: 3, <ArithmeticCircuit:MUL#7 (5,6)>:␣
↪176, <ArithmeticCircuit:SUB#8 (7,5)>: 43, <ArithmeticCircuit:CONST[value=z8]#9 ()>: 2,
↪<ArithmeticCircuit:ADD#10 (4,9)>: 130}
```

```
'Digraph.gv.pdf'
```

As we can see in the output, every node in the circuit is shown along with its value. We can see in the graph for a better illustration. Now, let's set `as_list=True` to see the output:

```
T = C.trace(inp, convert_input=True, convert_values=True, as_list=True)
print("Trace values:")
print(T)
```

```
Trace values:
[15, 20, 27, 19, 128, 155, 3, 176, 43, 2, 130]
```

2.2 How to define a new circuit type

In this tutorial, we show you how to define a new circuit type, i.e. define operations of your preference in a circuit, based on the `circkit` framework. In particular, we provide the guidance on:

1. *Defining a new circuit type*
2. *Syntactic sugar*
3. *Some examples*

Note: Note that this tutorial shows you how to define a new circuit type. In practice it is better to use the built-in *ArithmeticCircuit* or *OptArithmeticCircuit* as in the tutorial of building an arithmetic circuit. If necessary, you could inherit those circuits and then define your own operations.

First of all, we need to add the path of the `circkit` folder to `sys.path`:

```
import sys
sys.path.append("../")
```

2.2.1 1. Defining a new circuit type

Defining syntax

To define a new circuit type, we follow the steps:

1. Inherit the `Circuit` class
2. Inherit the `Circuit.Operations` class inside the new circuit type
3. Define the operations of the new circuit type as classes nested inside the `Operations` class. Depending on the numbers of input nodes and output nodes, each operation should inherit one of the following types of `Operation`:

Class	Usage
`Operation.Unary`	Operation with 1 input node
`Operation.Binary`	Operation with 2 input nodes
`Operation.Ternary`	Operation with 3 input nodes
`Operation.Variadic`	Operation with variable number of input nodes
`Operation.MultiNullary`	Operation with no inputs and variable number of output nodes
`Operation.MultiUnary`	Operation with 1 input and variable number of output nodes
`Operation.MultiBinary`	Operation with 2 inputs and variable number of output nodes
`Operation.MultiTernary`	Operation with 3 inputs and variable number of output nodes
`Operation.MultiVariadic`	Operation with variable number of input nodes and variable number of output nodes

Let's define addition operation as an example:

```
from circkit import Circuit, Operation

class NewCircuitType(Circuit):
    class Operations(Circuit.Operations):
        class ADD(Operation.Binary):
            pass
```

Now, we can construct a circuit with the addition defined above. Recall that we still can use the following useful functions (as shown in the tutorial of building an arithmetic circuit) to build a circuit:

- `add_input(name)`: add an input node
- `add_inputs(n, format)`: add `n` input nodes
- `add_output(node)`: mark `node` as an output node
- `add_output(nodelist)`: mark `nodelist` as a list of output nodes
- `digraph().view()`: draw and view the graph of the circuit

```
circuit = NewCircuitType(name="A test circuit")

x = circuit.add_input("x")
y = circuit.add_input("y")
z = circuit.ADD()(x, y)
circuit.add_output(z)

# circuit.digraph().view()
print("Circuit's input nodes:")
print(circuit.inputs)
print("Circuit's output nodes:")
print(circuit.outputs)
```

```
Circuit's input nodes:
[<NewCircuitType:INPUT[name=x]#0 ()>, <NewCircuitType:INPUT[name=y]#1 ()>]
Circuit's output nodes:
[<NewCircuitType:ADD#2 (0,1)>]
```

2.2.2 Defining evaluation

So far, the circuit is just about the syntax since we define the computational graph but no computational rules. This is fine, since typical applications are not all about computing the circuit. However, evaluating the circuit can be useful for testing purposes. To achieve this, we simply need to define the evaluation function for our operation.

```
from circkit import Circuit, Operation

class NewCircuitType(Circuit):
    class Operations(Circuit.Operations):
        class ADD(Operation.Binary):
            def eval(self, a, b):
                return a + b
```

Now we can evaluate the circuit.

```
circuit = NewCircuitType(name="A test circuit")

x = circuit.add_input("x")
y = circuit.add_input("y")
z = circuit.ADD()(x, y)
circuit.add_output(z)

inp = [10, 20]
```

(continues on next page)

(continued from previous page)

```
out = circuit.evaluate(inp)
print("Circuit's output:")
print(out)
```

```
Circuit's output:
[30]
```

Defining operations with parameters

Defining an operation parameter is done through annotations, with possible assignment to mark a default value. It is then stored as an attribute of the operation instance, accessible e.g. for evaluation. Let's define `EXP` operation with the `power` parameter as an example.

```
from circkit import Circuit, Operation, Param

class NewCircuitType(Circuit):
    class Operations(Circuit.Operations):
        class ADD(Operation.Binary):
            def eval(self, a, b):
                return a + b

        class EXP(Operation.Unary):
            power : Param.Int(min_value=0) = 2
            def eval(self, a):
                return a**self.power
```

In the example above, `power` takes 2 as the default value. Let's build a circuit to test it:

```
circuit = NewCircuitType(name="test circuit")

x = circuit.add_input("x")
xsquare = circuit.EXP()(x)
xcube = circuit.EXP(3)(x)
circuit.add_output([xsquare, xcube])

inp = [5]
out = circuit.evaluate([5])
print("Circuit's output:")
print(out)
```

```
Circuit's output:
[25, 125]
```

Here, we used Param.Int to constraint the parameter type and value. The following table contains the parameter constraints supported by `circkit`:

Class	Usage
Param.Const	constants
Param.Int	integers
Param.Bool	booleans
Param.Str	strings
Param.Tuple	tuples
Param.InputName	name of an input, can be string or integer

If we provide an incompatible value, it will cause an error. For example:

```
xquartic = circuit.EXP("four")(x)
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

/Users/nvietsang/Work/wbc/circkit/docs/tuto-p2_new-circuit-type.ipynb Cell 22 in <cell line: 1>()
----> <a href='vscode-notebook-cell:/Users/nvietsang/Work/wbc/circkit/docs/tuto-p2_new-circuit-type.ipynb#ch0000022?line=0'>1</a> xquartic = circuit.EXP("four")(x)


File ~/Work/wbc/circkit/docs/../circkit/operation.py:160, in OperationMeta.__call__(cls, *values, **kvalues)
    152     raise TypeError(
    153         "Creating an operation that is not linked to a circuit."
    154         " Calling an operation from a circuit class?"
    155         " Or using a removed operation from a superclass?"
    156     )
    158 # create the Operation instance anyway
    159 # (not avoiding it to unify parsing of parameters)
--> 160 op_new = super().__call__(*values, **kvalues)
    161 if cls._circuit is not None and cls._circuit.CACHE_OPERATIONS:
    162     # if a similar operation is in cache,
    163     # return it instead (the new one will be deleted)
    164     cache = op_new._circuit._operations_cache


File ~/Work/wbc/circkit/docs/../circkit/operation.py:278, in Operation.__init__(self, *values, **kvalues)
    272 # 2: process through validators/converters
    273 for name, param in self._param_descriptions.items():
    274     # param can use previously set value, or not?
    275     # possible:
    276     # - param checks only single value, is independent
    277     # - to check groups, add methods to the op class
--> 278     setattr(self, name, param.create(self, value=kvalues[name]))


File ~/Work/wbc/circkit/docs/../circkit/param.py:60, in IntParam.create(self, operation, value)
     59 def create(self, operation, value: int):
```

(continues on next page)

(continued from previous page)

```
---> 60     value = int(value)
     61     if self.min_value is not None and self.min_value > value:
     62         raise Param.InvalidValue(
     63             f"Integer value should not be smaller than {self.min_value}")


ValueError: invalid literal for int() with base 10: 'four'
```

2.2.3 2. Syntactic sugar

It is a bit clumsy to write ADD, EXP when building circuits, when these are basic arithmetic operations. We can define syntax sugar naturally by subclassing the Node class.

```
class NewCircuitType(Circuit):
    class Operations(Circuit.Operations):
        class ADD(Operation.Binary):
            def eval(self, a, b):
                return a + b

        class EXP(Operation.Unary):
            power : Param.Int(min_value=0) = 2
            def eval(self, a):
                return a**self.power

    class Node(Circuit.Node):
        def __add__(self, other):
            return self.circuit.ADD()(self, other)

        def __pow__(self, power):
            return self.circuit.EXP(power)(self)
```

Life gets much easier now:

```
circuit = NewCircuitType()
x = circuit.add_input("x")
y = circuit.add_input("y")
z = (x + y)**2 + x**5
circuit.add_output(z)

inp = [10, 1]
out = circuit.evaluate(inp)
print("Circuit's output:")
print(out)
```

```
Circuit's output:
[100121]
```

2.2.4 3. Some examples

In this section, we demonstrate examples of defining circuit types with some interesting operations (rather than basic addition, substraction, multiplication, ...). This aims to show that we can define a new circuit type with *our own operations*.

Example 1

We define a new circuit type with 2 operations:

- MADD: given a list $(x_1, x_2, \ldots, x_n)$, it returns the sum $x_1 + x_2 + \ldots + x_n$
- MMUL: given a list $(x_1, x_2, \ldots, x_n)$, it returns the product $x_1 \times x_2 \times \ldots \times x_n$

```
# Define a new circuit type
class NewCircuitType(Circuit):
    class Operations(Circuit.Operations):
        class MADD(Operation.Variadic):
            def eval(self, *operands):
                return sum(operands)

        class MMUL(Operation.Variadic):
            def eval(self, *operands):
                r = 1
                for x in operands:
                    r *= x
                return r

# Create a new circuit instance
circuit = NewCircuitType(name="test circuit")
x = circuit.add_inputs(5, "x%d")
y = circuit.MADD()(*x)
z = circuit.MMUL()(*x)
circuit.add_output(y)
circuit.add_output(z)

# Evaluate the circuit
inp = [x+1 for x in range(5)]
out = circuit.evaluate(inp)
print("Circuit's output:")
print(out)
```

```
Circuit's output:
[15, 120]
```

Example 2

We define a new circuit type with 2 operations:

- MADDC: given a constant c and a list $(x_1, x_2, \ldots, x_n)$, it returns a list $(c + x_1, c + x_2, \ldots, c + x_n)$
- MMULC: given a constant c and a list $(x_1, x_2, \ldots, x_n)$, it returns a list $(cx_1, cx_2, \ldots, cx_n)$

```
# Define a new circuit type
class NewCircuitType(Circuit):
    class Operations(Circuit.Operations):
        class MADDC(Operation.Variadic):
            def eval(self, c, *operands):
                return [c + x for x in operands]

        class MMULC(Operation.Variadic):
            def eval(self, c, *operands):
                return [c * x for x in operands]

# Create a new circuit instance
circuit = NewCircuitType(name="test circuit")
c = 10
x = circuit.add_inputs(5, "x%d")
y = circuit.MADDC()(c, *x)
z = circuit.MMULC()(c, *x)
circuit.add_output(y)
circuit.add_output(z)

# Evaluate the circuit
inp = [x+1 for x in range(5)]
out = circuit.evaluate(inp)
print("Circuit's output:")
print(out)
```

```
Circuit's output:
[[11, 12, 13, 14, 15], [10, 20, 30, 40, 50]]
```

2.3 How to define a transformer

By defining a transformer, we can transform a circuit into another circuit (possibly of a new circuit type). In this tutorial, we show you:

- ISW transformer: given a boolean circuit, we transform it into a new circuit working on shares (ISW circuit, see ISW03). This is the built-in transformer which you can import and use directly from the :mod:`circkit` framework.
- How to define your own transformer: we show you the steps of defining the ISW transformer. You will see how to define a new transformer from those steps.

2.3.1 ISW Transformer

A boolean circuit can be created by an arithmetic circuit working on GF(2) with addition and multiplication operations (corresponding to XOR and AND in boolean).

```
import sys
sys.path.append("../")
```

```
from circkit.transformers.isw import IswOnArithmetic
from circkit.arithmetic import ArithmeticCircuit
from sage.all import GF
K = GF(2)

C = ArithmeticCircuit(base_ring=K)
x = C.add_input("x")
y = C.add_input("y")

z = x * y + 1
t = z + x + 1
C.add_output(t)

# ISW transformer
transformer = IswOnArithmetic(order=2)
iswC = transformer.transform(C)

# see the graph and verify the ISW circuit
iswC.digraph().view()

# Evaluate on original circuit
inp = [1, 0]
out = C.evaluate(inp)
print(f"Original circuit's output: {out}")

# Evaluate on ISW circuit
# 1 = 1 + 0 + 0 and 0 = 1 + 1 + 0
inp_shares = [1, 0, 0, 1, 1, 0]
n_tests = 5
for i in range(n_tests):
    out_shares = iswC.evaluate(inp_shares)
    ret = 0
    for s in out_shares:
        ret ^= s
    print(f"Output shares: {out_shares} --> {ret}")
```

```
Original circuit's output: [1]
Output shares: [1, 1, 1] --> 1
Output shares: [0, 0, 1] --> 1
Output shares: [0, 0, 1] --> 1
Output shares: [0, 1, 0] --> 1
Output shares: [1, 1, 1] --> 1
```

2.3.2 How to define your transformer

In this section, we show how to define the ISW transformer from which we can see the steps of defining a new transformer.

Given a *source circuit*, our goal is to transform it into a *target circuit*. The high-level idea is to visit all nodes in the source circuit and process each node in the way we want to define the transformer. The :mod:`circkit` framework already provides the skeleton of the transformation in the `CircuitTransformation` class. We just need to inherit this class and then define the `visit_<OP>` functions where <OP> are the operations (or node types) defined in the circuit type.

In a boolean circuit, there are 4 node types. Therefore, we define 4 functions:

- `visit_INPUT`: for each input node in the source circuit, we create its nodes of shares in the target circuit.
- `visit_ADD` (XOR): a XOR node in the source circuit represents by some XOR nodes on the shares of the operands in the target circuit.
- `visit_MUL` (AND): to transform an AND node in the source circuit, we have to generate some randomnesses and create some XOR and AND nodes on those randomnesses and the shares.
- `visit_CONST`: a constant is represented by some shares in the target circuit.

The following code is the implementation of the ISW transformer:

```
from circkit.transformers.core import CircuitTransformer
from circkit.arithmetic import ArithmeticCircuit
from circkit.array import Array

class IswOnArithmetic(CircuitTransformer):
    # circuit type of the target circuit
    TARGET_CIRCUIT = ArithmeticCircuit

    def __init__(self, order: int):
        """
        Arguments
        ---------
        :order: ISW masking order
        """
        super().__init__()
        self.order = order
        self.n_shares = order + 1

    def visit_INPUT(self, node):
        shares = []
        for i in range(self.n_shares):
            new_name = f"{node.operation.name}_share{i}"
            x = self.target_circuit.add_input(new_name)
            shares.append(x)
        shares = Array(shares)

        return shares

    def visit_ADD(self, node, x, y):
        return x + y

    def visit_MUL(self, node, x, y):
```

(continues on next page)

(continued from previous page)

```
        r = [[0] * self.n_shares for _ in range(self.n_shares)]
        for i in range(self.n_shares):
            for j in range(i+1, self.n_shares):
                r[i][j] = self.target_circuit.RND()()
                r[j][i] = r[i][j] + x[i]*y[j] + x[j]*y[i]

        z = x * y
        for i in range(self.n_shares):
            for j in range(self.n_shares):
                if i != j:
                    z[i] = z[i] + r[i][j]
        return z

    def visit_CONST(self, node):
        shares = Array(self.target_circuit.RND()() for i in range(self.order))
        c = self.target_circuit.add_const(node.operation.value)

        for i in range(self.order):
            c = c + shares[i]
        shares.append(c)

        return shares
```

2.4 Bit-slicing AES

In this example, we build a circuit of bit-slicing AES. This implementation is based on the white-box AES of BU18.

2.4.1 Operations on vectors

```
import operator

class Vector(list):
    ZERO = 0
    WIDTH = None

    @classmethod
    def make(cls, lst):
        lst = list(lst)
        if cls.WIDTH is not None:
            assert len(lst) == cls.WIDTH
        return cls(lst)

    def split(self, n=2):
        assert len(self) % n == 0
        w = len(self) // n
        return Vector(self.make(self[i:i+w]) for i in range(0, len(self), w))

    def rol(self, n=1):
        n %= len(self)
```

(continues on next page)

(continued from previous page)

```
        return self.make(self[n:] + self[:n])

    def ror(self, n=1):
        return self.rol(-n)

    def __repr__(self):
        return "<Vector len=%d list=%r>" % (len(self), list(self))

    def flatten(self):
        if isinstance(self[0], Vector):
            return self[0].concat(*self[1:])
        return reduce(operator.add, list(self))

    def map(self, f, with_coord=False):
        if with_coord:
            return self.make(f(i, v) for i, v in enumerate(self))
        else:
            return self.make(f(v) for v in self)

    def __xor__(self, other):
        assert isinstance(other, Vector)
        assert len(self) == len(other)
        return self.make(a ^ b for a, b in zip(self, other))

    def __or__(self, other):
        assert isinstance(other, Vector)
        assert len(self) == len(other)
        return self.make(a | b for a, b in zip(self, other))

    def __and__(self, other):
        assert isinstance(other, Vector)
        assert len(self) == len(other)
        return self.make(a & b for a, b in zip(self, other))

    def set(self, x, val):
        return self.make(v if i != x else val for i, v in enumerate(self))
```

2.4.2 Operations on matrices (Rect)

```
class Rect(object):
    def __init__(self, vec, h=None, w=None):
        assert h or w
        if h:
            w = len(vec) // h
        elif w:
            h = len(vec) // w
        assert w * h == len(vec)
        self.w, self.h = w, h

        self.lst = []
```

(continues on next page)

(continued from previous page)

```
        for i in range(0, len(vec), w):
            self.lst.append(list(vec[i:i+w]))

    @classmethod
    def from_rect(cls, rect):
        self = object.__new__(cls)
        self.lst = rect
        self.h = len(rect)
        self.w = len(rect[0])
        return self

    def __getitem__(self, pos):
        y, x = pos
        return self.lst[y][x]

    def __setitem__(self, pos, val):
        y, x = pos
        self.lst[y][x] = val

    def row(self, i):
        return Vector(self.lst[i])

    def col(self, i):
        return Vector(self.lst[y][i] for y in range(self.h))

    def set_row(self, y, vec):
        for x in range(self.w):
            self.lst[y][x] = vec[x]
        return self

    def set_col(self, x, vec):
        for y in range(self.h):
            self.lst[y][x] = vec[y]
        return self

    def apply(self, f, with_coord=False):
        for y in range(self.h):
            if with_coord:
                self.lst[y] = [f(y, x, v) for x, v in enumerate(self.lst[y])]
            else:
                self.lst[y] = list(map(f, self.lst[y]))
        return self

    def apply_row(self, x, func):
        return self.set_row(x, func(self.row(x)))

    def apply_col(self, x, func):
        return self.set_col(x, func(self.col(x)))

    def flatten(self):
        lst = []
        for v in self.lst:
```

(continues on next page)

(continued from previous page)

```
            lst += v
        return Vector(lst)

    def zipwith(self, f, other):
        assert isinstance(other, Rect)
        assert self.h == other.h
        assert self.w == other.w
        return Rect(
            [f(a, b) for a, b in zip(self.flatten(), other.flatten())],
            h=self.h, w=self.w
        )

    def transpose(self):
        rect = [[self.lst[y][x] for y in range(self.h)] for x in range(self.w)]
        return Rect.from_rect(rect=rect)

    def __repr__(self):
        return "<Rect %dx%d>" % (self.h, self.w)
```

2.4.3 Bit-slicing implementation of Sbox

```
def Not(x):
    return 1^x

def GF_SQ_2(A): return A[1], A[0]
def GF_SCLW_2(A): return A[1], A[1] ^ A[0]
def GF_SCLW2_2(A): return A[1] ^ A[0], A[0]

def GF_MULS_2(A, ab, B, cd):
    abcd = (ab & cd)
    p = ((A[1] & B[1])) ^ abcd
    q = ((A[0] & B[0])) ^ abcd
    return q, p

def GF_MULS_SCL_2(A, ab, B, cd):
    t = (A[0] & B[0])
    p = ((ab & cd)) ^ t
    q = ((A[1] & B[1])) ^ t
    return q, p

def XOR_LIST(a, b):
    return [a ^ b for a, b in zip(a, b)]

def NotOr(a, b):
    # return Not(a | b)
    return Not(a) & Not(b)

def GF_INV_4(A):
    a = A[2:4]
    b = A[0:2]
```

(continues on next page)

(continued from previous page)

```
    sa = a[1] ^ a[0]
    sb = b[1] ^ b[0]

    ab = GF_MULS_2(a, sa, b, sb)
    ab2 = GF_SQ_2(XOR_LIST(a, b))
    ab2N = GF_SCLW2_2(ab2)
    d = GF_SQ_2(XOR_LIST(ab, ab2N))

    c = [
        NotOr(sa, sb) ^ (Not(a[0] & b[0])),
        NotOr(a[1], b[1]) ^ (Not(sa & sb)),
    ]

    sd = d[1] ^ d[0]
    p = GF_MULS_2(d, sd, b, sb)
    q = GF_MULS_2(d, sd, a, sa)
    return q + p

def GF_SQ_SCL_4(A):
    a = A[2:4]
    b = A[0:2]
    ab2 = GF_SQ_2(a ^ b)
    b2 = GF_SQ_2(b)
    b2N2 = GF_SCLW_2(b2)
    return b2N2 + ab2

def GF_MULS_4(A, a, Al, Ah, aa, B, b, Bl, Bh, bb):
    ph = GF_MULS_2(A[2:4], Ah, B[2:4], Bh)
    pl = GF_MULS_2(A[0:2], Al, B[0:2], Bl)
    p = GF_MULS_SCL_2(a, aa, b, bb)
    return XOR_LIST(pl, p) + XOR_LIST(ph, p) #(pl ^ p), (ph ^ p)

def GF_INV_8(A):
    a = A[4:8]
    b = A[0:4]
    sa = XOR_LIST(a[2:4], a[0:2])
    sb = XOR_LIST(b[2:4], b[0:2])
    al = a[1] ^ a[0]
    ah = a[3] ^ a[2]
    aa = sa[1] ^ sa[0]
    bl = b[1] ^ b[0]
    bh = b[3] ^ b[2]
    bb = sb[1] ^ sb[0]

    c1 = (ah & bh)
    c2 = (sa[0] & sb[0])
    c3 = (aa & bb)

    c = [
        (NotOr(a[0] , b[0] ) ^ ((al & bl))) ^ ((sa[1] & sb[1])) ^ Not(c2), #0
        (NotOr(al    , bl    ) ^ (Not(a[1] & b[1]))) ^ c2 ^ c3 , #1
        (NotOr(sa[1], sb[1]) ^ (Not(a[2] & b[2]))) ^ c1 ^ c2 , #2
```

(continues on next page)

(continued from previous page)

```
        (NotOr(sa[0], sb[0]) ^ (Not(a[3] & b[3]))) ^ c1 ^ c3 , #3
    ]
    d = GF_INV_4(c)

    sd = XOR_LIST(d[2:4], d[0:2])
    dl = d[1] ^ d[0]
    dh = d[3] ^ d[2]
    dd = sd[1] ^ sd[0]
    p = GF_MULS_4(d, sd, dl, dh, dd, b, sb, bl, bh, bb)
    q = GF_MULS_4(d, sd, dl, dh, dd, a, sa, al, ah, aa)
    return q + p


def MUX21I(A, B, s): #return ((~A & s) ^ (~B & ~s)
    return Not(A if s else B)

def SELECT_NOT_8( A, B, s):
    Q = [None] * 8
    for i in range(8):
        Q[i] = MUX21I(A[i], B[i], s)
    return Q

def Sbox(A, encrypt):
    R1 = A[7] ^ A[5]
    R2 = A[7] ^ Not(A[4])
    R3 = A[6] ^ A[0]
    R4 = A[5] ^ Not(R3)
    R5 = A[4] ^ R4
    R6 = A[3] ^ A[0]
    R7 = A[2] ^ R1
    R8 = A[1] ^ R3
    R9 = A[3] ^ R8

    B = [None] * 8
    B[7] = R7 ^ Not(R8)
    B[6] = R5
    B[5] = A[1] ^ R4
    B[4] = R1 ^ Not(R3)
    B[3] = A[1]^ R2 ^ R6
    B[2] = Not( A[0])
    B[1] = R4
    B[0] = A[2] ^ Not(R9)

    Y = [None] * 8
    Y[7] = R2
    Y[6] = A[4] ^ R8
    Y[5] = A[6] ^ A[4]
    Y[4] = R9
    Y[3] = A[6] ^ Not(R2)
    Y[2] = R7
    Y[1] = A[4] ^ R6
    Y[0] = A[1] ^ R5
```

(continues on next page)

(continued from previous page)

```
    Z = SELECT_NOT_8(B, Y, encrypt)
    C = GF_INV_8(Z)

    T1 = C[7] ^ C[3]
    T2 = C[6] ^ C[4]
    T3 = C[6] ^ C[0]
    T4 = C[5] ^ Not(C[3])
    T5 = C[5] ^ Not(T1)
    T6 = C[5] ^ Not(C[1])
    T7 = C[4] ^ Not(T6)
    T8 = C[2] ^ T4
    T9 = C[1] ^ T2
    T10 = T3 ^ T5

    D = [None] * 8
    D[7] = T4
    D[6] = T1
    D[5] = T3
    D[4] = T5
    D[3] = T2 ^ T5
    D[2] = T3 ^ T8
    D[1] = T7
    D[0] = T9

    X = [None] * 8
    X[7] = C[4] ^ Not(C[1])
    X[6] = C[1] ^ T10
    X[5] = C[2] ^ T10
    X[4] = C[6] ^ Not(C[1])
    X[3] = T8 ^ T9
    X[2] = C[7] ^ Not(T7)
    X[1] = T6
    X[0] = Not(C[2])
    return SELECT_NOT_8(D, X, encrypt)


def bitSbox(A, inverse=False):
    res = Sbox(A[::-1], encrypt=1-inverse)[::-1]
    return res
```

2.4.4 Bit-slicing implementations of ShiftRow and MixColumn

```
def ShiftRow(row, nr, inverse=False):
    if inverse:
        nr = -nr
    off = nr % 4
    return row[off:] + row[:off]

def MixColumn(col, inverse=False):
```

(continues on next page)

(continued from previous page)

```
    res = [[0] * 8 for _ in range(4)]
    table = MCi_TABLE if inverse else MC_TABLE
    for yi in range(4):
        for yj in range(8):
            y = yi * 8 + yj
            for x in table[y]:
                xi, xj = divmod(x, 8)
                res[yi][yj] ^= col[xi][xj]
    return res

# y -> set of x indices to xor
MC_TABLE = [{1, 8, 9, 16, 24}, {2, 9, 10, 17, 25}, {3, 10, 11, 18, 26}, {0, 4, 8, 11, 12,
↪ 19, 27}, {0, 5, 8, 12, 13, 20, 28}, {6, 13, 14, 21, 29}, {0, 7, 8, 14, 15, 22, 30},
↪{0, 8, 15, 23, 31}, {0, 9, 16, 17, 24}, {1, 10, 17, 18, 25}, {2, 11, 18, 19, 26}, {3,␣
↪8, 12, 16, 19, 20, 27}, {4, 8, 13, 16, 20, 21, 28}, {5, 14, 21, 22, 29}, {6, 8, 15, 16,
↪ 22, 23, 30}, {7, 8, 16, 23, 31}, {0, 8, 17, 24, 25}, {1, 9, 18, 25, 26}, {2, 10, 19,␣
↪26, 27}, {3, 11, 16, 20, 24, 27, 28}, {4, 12, 16, 21, 24, 28, 29}, {5, 13, 22, 29, 30},
↪ {6, 14, 16, 23, 24, 30, 31}, {7, 15, 16, 24, 31}, {0, 1, 8, 16, 25}, {1, 2, 9, 17, 26}
↪, {2, 3, 10, 18, 27}, {0, 3, 4, 11, 19, 24, 28}, {0, 4, 5, 12, 20, 24, 29}, {5, 6, 13,␣
↪21, 30}, {0, 6, 7, 14, 22, 24, 31}, {0, 7, 15, 23, 24}]
MCi_TABLE = [{1, 2, 3, 8, 9, 11, 16, 18, 19, 24, 27}, {0, 2, 3, 4, 8, 9, 10, 12, 16, 17,␣
↪19, 20, 24, 25, 28}, {1, 3, 4, 5, 8, 9, 10, 11, 13, 17, 18, 20, 21, 24, 25, 26, 29},
↪{2, 4, 5, 6, 8, 9, 10, 11, 12, 14, 16, 18, 19, 21, 22, 25, 26, 27, 30}, {1, 2, 5, 6, 7,
↪ 10, 12, 13, 15, 16, 17, 18, 20, 22, 23, 24, 26, 28, 31}, {1, 6, 7, 8, 9, 13, 14, 17,␣
↪21, 23, 24, 25, 29}, {2, 7, 8, 9, 10, 14, 15, 16, 18, 22, 25, 26, 30}, {0, 1, 2, 8, 10,
↪ 15, 17, 18, 23, 26, 31}, {0, 3, 9, 10, 11, 16, 17, 19, 24, 26, 27}, {0, 1, 4, 8, 10,␣
↪11, 12, 16, 17, 18, 20, 24, 25, 27, 28}, {0, 1, 2, 5, 9, 11, 12, 13, 16, 17, 18, 19,␣
↪21, 25, 26, 28, 29}, {1, 2, 3, 6, 10, 12, 13, 14, 16, 17, 18, 19, 20, 22, 24, 26, 27,␣
↪29, 30}, {0, 2, 4, 7, 9, 10, 13, 14, 15, 18, 20, 21, 23, 24, 25, 26, 28, 30, 31}, {0,␣
↪1, 5, 9, 14, 15, 16, 17, 21, 22, 25, 29, 31}, {1, 2, 6, 10, 15, 16, 17, 18, 22, 23, 24,
↪ 26, 30}, {2, 7, 8, 9, 10, 16, 18, 23, 25, 26, 31}, {0, 2, 3, 8, 11, 17, 18, 19, 24,␣
↪25, 27}, {0, 1, 3, 4, 8, 9, 12, 16, 18, 19, 20, 24, 25, 26, 28}, {1, 2, 4, 5, 8, 9, 10,
↪ 13, 17, 19, 20, 21, 24, 25, 26, 27, 29}, {0, 2, 3, 5, 6, 9, 10, 11, 14, 18, 20, 21,␣
↪22, 24, 25, 26, 27, 28, 30}, {0, 1, 2, 4, 6, 7, 8, 10, 12, 15, 17, 18, 21, 22, 23, 26,␣
↪28, 29, 31}, {1, 5, 7, 8, 9, 13, 17, 22, 23, 24, 25, 29, 30}, {0, 2, 6, 9, 10, 14, 18,␣
↪23, 24, 25, 26, 30, 31}, {1, 2, 7, 10, 15, 16, 17, 18, 24, 26, 31}, {0, 1, 3, 8, 10,␣
↪11, 16, 19, 25, 26, 27}, {0, 1, 2, 4, 8, 9, 11, 12, 16, 17, 20, 24, 26, 27, 28}, {0, 1,
↪ 2, 3, 5, 9, 10, 12, 13, 16, 17, 18, 21, 25, 27, 28, 29}, {0, 1, 2, 3, 4, 6, 8, 10, 11,
↪ 13, 14, 17, 18, 19, 22, 26, 28, 29, 30}, {2, 4, 5, 7, 8, 9, 10, 12, 14, 15, 16, 18,␣
↪20, 23, 25, 26, 29, 30, 31}, {0, 1, 5, 6, 9, 13, 15, 16, 17, 21, 25, 30, 31}, {0, 1, 2,
↪ 6, 7, 8, 10, 14, 17, 18, 22, 26, 31}, {0, 2, 7, 9, 10, 15, 18, 23, 24, 25, 26}]
```

2.4.5 Bit-slicing implementation of Key Schedule

```
Rcon = [0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36,
        0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97,
        0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72,
        0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66,
        0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
        0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d,
        0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
        0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61,
        0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,
        0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40,
        0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc,
        0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5,
        0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a,
        0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d,
        0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c,
        0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,
        0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4,
        0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,
        0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08,
        0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,
        0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
        0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2,
        0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74,
        0xe8, 0xcb ]

def tobin(x, n):
    return tuple(map(int, bin(x).lstrip("0b").rjust(n, "0")))

def c8(c):
    return Vector(tobin(c, 8))

BitRcon = list(map(c8, Rcon))

def ks_rotate(word):
    return word[1:] + word[:1]

def ks_core(word, iteration):
    word = word.rol(1)
    word = word.map(lambda b: Vector(bitSbox(b)))
    word = word.set(0, word[0] ^ BitRcon[iteration])
    return word

def KS_round(kstate, rno):
    t = ks_core(kstate.col(3), rno+1)
    kstate.apply_col(0, lambda c: c ^ t)
    t = kstate.col(0)
    kstate.apply_col(1, lambda c: c ^ t)
    t = kstate.col(1)
    kstate.apply_col(2, lambda c: c ^ t)
    t = kstate.col(2)
    kstate.apply_col(3, lambda c: c ^ t)
```

(continues on next page)

(continued from previous page)

```
    return kstate
```

2.4.6 Bit-slicing AES

```
def BitAES(plaintext, key, rounds=10):
    bx = Vector(plaintext).split(16)
    bk = Vector(key).split(16)

    state = Rect(bx, w=4, h=4).transpose()
    kstate = Rect(bk, w=4, h=4).transpose()

    for rno in range(rounds):
        state = AK(state, kstate)
        state = SB(state)
        state = SR(state)
        if rno < rounds-1:
            state = MC(state)
        kstate = KS(kstate, rno)
    state = AK(state, kstate)

    state = state.transpose()
    kstate = kstate.transpose()
    bits = sum( map(list, state.flatten()), [])
    kbits = sum( map(list, kstate.flatten()), [])
    return bits, kbits

def AK(state, kstate):
    return state.zipwith(lambda a, b: a ^ b, kstate)

def SB(state, inverse=False):
    return state.apply(lambda v: Vector(bitSbox(v, inverse=inverse)))

def SR(state, inverse=False):
    for y in range(4):
        state.apply_row(y, lambda row: ShiftRow(row, y, inverse=inverse))
    return state

def MC(state, inverse=False):
    for x in range(4):
        state.apply_col(x, lambda v: list(map(Vector, MixColumn(v))))
    return state

def KS(kstate, rno):
    return KS_round(kstate, rno)
```

2.4.7 Define a new circuit type

This new circuit type composes of binary operations (AND, OR, XOR) which are necessary for the bit-slicing implementation.

```
import sys
sys.path.append("..")
from circkit import Operation, Circuit

class BitCircuit(Circuit):
    class Operations(Circuit.Operations):
        class AND(Operation.Binary):
            SYMMETRIC = True
            def eval(self, a, b):
                return a & b

        class OR(Operation.Binary):
            SYMMETRIC = True
            def eval(self, a, b):
                return a | b

        class XOR(Operation.Binary):
            SYMMETRIC = True
            def eval(self, a, b):
                return a ^ b

    class Node(Circuit.Node):
        __slots__ = ()

        def __xor__(self, b):
            return self.circuit.XOR()(self, b)

        def __rxor__(self, b):
            return self.circuit.XOR()(b, self)

        def __or__(self, b):
            return self.circuit.OR()(self, b)

        def __and__(self, b):
            return self.circuit.AND()(self, b)
```

2.4.8 Build a circuit for bit-slicing AES

```
C = BitCircuit()
pt = C.add_inputs(n=128, format="m%d")
k = C.add_inputs(n=128, format="k%d")
ct, k10 = BitAES(pt, k, rounds=10)
C.add_output(ct)
```

Then we can evaluate the circuit to test its correctness.

```
def str2bin(s):
    return list(map(int, "".join(bin(ord(c))[2:].zfill(8) for c in s)))

def hex2bin(s):
    return list(map(int, "".join(bin(c)[2:].zfill(8) for c in bytes.fromhex(s))))

def bin2hex(s):
    assert len(s) % 8 == 0
    v = int("".join(map(str, s)), 2)
    v = ("%x" %v).zfill(len(s) // 4)
    return v

# Expected ciphertext: 69c4e0d86a7b0430d8cdb78070b4c55a
# See the AES documentation of NIST: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.
↪197.pdf
MSG = "00112233445566778899aabbccddeeff"
KEY = "000102030405060708090a0b0c0d0e0f"

inp = hex2bin(MSG) + hex2bin(KEY)
out = C.evaluate(inp)
out = bin2hex(out)
print(f"Ciphertext: {out}")
```

```
Ciphertext: 69c4e0d86a7b0430d8cdb78070b4c55a
```

For a white-box implementation, we can treat the key as 128 constants of bits instead of 128 input nodes.

```
C = BitCircuit()
pt = C.add_inputs(n=128, format="m%d")
k = [C.add_const(b) for b in hex2bin(KEY)]
ct, k10 = BitAES(pt, k, rounds=10)
C.add_output(ct)

inp = hex2bin(MSG)
out = C.evaluate(inp)
out = bin2hex(out)
print(f"Ciphertext: {out}")
```

```
Ciphertext: 69c4e0d86a7b0430d8cdb78070b4c55a
```

2.5 Simon Cipher

2.5.1 Implementation of Simon cipher

```
# Reference: https://github.com/inmcm/Simon_Speck_Ciphers/blob/master/Python/
↪simonspeckciphers/simon/simon.py
# For the sake of simplicity, we modified some functions of the original implementation
from collections import deque
```

(continues on next page)

(continued from previous page)

```
class SimonCipher(object):
    """Simon Block Cipher Object"""

    # Z Arrays (stored bit reversed for easier usage)
    z0 = 0b01100111000011010100100010111110110011100001101010010001011111
    z1 = 0b01011010000110010011111011100010101101000011001001111101110001
    z2 = 0b11001101101001111110001000010100011001001011000000111011110101
    z3 = 0b11110000101100111001010001001000000111101001100011010111011011
    z4 = 0b11110111001001010011000011101000000100011011010110011110001011

    # valid cipher configurations stored:
    # block_size:{key_size:(number_rounds,z sequence)}
    __valid_setups = {32: {64: (32, z0)},
                      48: {72: (36, z0), 96: (36, z1)},
                      64: {96: (42, z2), 128: (44, z3)},
                      96: {96: (52, z2), 144: (54, z3)},
                      128: {128: (68, z2), 192: (69, z3), 256: (72, z4)}}

    def __init__(self, key, key_size=128, block_size=128):
        """
        Initialize an instance of the Simon block cipher.
        :param key: Int representation of the encryption key
        :param key_size: Int representing the encryption key in bits
        :param block_size: Int representing the block size in bits
        :return: None
        """

        # Setup block/word size
        self.possible_setups = self.__valid_setups[block_size]
        self.block_size = block_size
        self.word_size = self.block_size >> 1

        self.rounds, self.zseq = self.possible_setups[key_size]
        self.key_size = key_size

        # Create Properly Sized bit mask for truncating addition and left shift outputs
        self.mod_mask = (2 ** self.word_size) - 1

        # Check key length
        assert len(key) == (self.key_size // self.word_size)
        self.key = key

        # Pre-compute the key schedule
        self.key_schedule()


    def key_schedule(self):
        m = self.key_size // self.word_size
        m1 = m - 1
        self.round_keys = []
        # Create list of subwords from encryption key
        k_init = self.key
```

(continues on next page)

(continued from previous page)

```
        k_reg = deque(k_init)  # Use queue to manage key subwords

        round_constant = self.mod_mask ^ 3  # Round Constant is 0xFFFF..FC

        # Generate all round keys
        for x in range(self.rounds):
            s3 = self.word_size - 3
            rs_3 = ((k_reg[0] << s3) + (k_reg[0] >> 3)) & self.mod_mask

            if m == 4:
                rs_3 = rs_3 ^ k_reg[2]

            s1 = self.word_size - 1
            rs_1 = ((rs_3 << s1) + (rs_3 >> 1)) & self.mod_mask

            c_z = ((self.zseq >> (x % 62)) & 1) ^ round_constant

            new_k = c_z ^ rs_1 ^ rs_3 ^ k_reg[m1]

            self.round_keys.append(k_reg.pop())
            k_reg.appendleft(new_k)

        return self.round_keys

    def encrypt(self, l, r):
        x = l
        y = r

        # Run Encryption Steps For Appropriate Number of Rounds
        for k in self.round_keys:
             # Generate all circular shifts
            s1 = self.word_size - 1
            s8 = self.word_size - 8
            s2 = self.word_size - 2
            ls_1_x = ((x >> s1) + (x << 1)) & self.mod_mask
            ls_8_x = ((x >> s8) + (x << 8)) & self.mod_mask
            ls_2_x = ((x >> s2) + (x << 2)) & self.mod_mask

            # XOR Chain
            xor_1 = (ls_1_x & ls_8_x) ^ y
            xor_2 = xor_1 ^ ls_2_x
            y = x
            x = k ^ xor_2

        return x, y

    def decrypt(self, l, r):
        x = l
        y = r

        # Run Encryption Steps For Appropriate Number of Rounds
```

(continues on next page)

(continued from previous page)

```
        for k in reversed(self.round_keys):
             # Generate all circular shifts
            ls_1_x = ((x >> (self.word_size - 1)) + (x << 1)) & self.mod_mask
            ls_8_x = ((x >> (self.word_size - 8)) + (x << 8)) & self.mod_mask
            ls_2_x = ((x >> (self.word_size - 2)) + (x << 2)) & self.mod_mask

            # XOR Chain
            xor_1 = (ls_1_x & ls_8_x) ^ y
            xor_2 = xor_1 ^ ls_2_x
            y = x
            x = k ^ xor_2

        return x, y
```

Then, we can test the above implementation with some test vectors given by the authors of Simon

```
k3264 = [0x1918, 0x1110, 0x0908, 0x0100]
l3264, r3264 = 0x6565, 0x6877
c3264 = SimonCipher(k3264, key_size=64, block_size=32)
t3264 = c3264.encrypt(l3264, r3264)
assert t3264 == (0xc69b, 0xe9bb)

k128256 = [0x1f1e1d1c1b1a1918, 0x1716151413121110, 0x0f0e0d0c0b0a0908,␣
↪0x0706050403020100]
l128256, r128256 = 0x74206e69206d6f6f, 0x6d69732061207369
c128256 = SimonCipher(k128256, key_size=256, block_size=128)
t128256 = c128256.encrypt(l128256, r128256)
assert t128256 == (0x8d2b5579afc8a3a0, 0x3bf72a87efe7b868)
```

2.5.2 Define a new circuit type for Simon cipher

By observing the above implementation of Simon cipher, we can see that the new circuit type has to support the following operations on integers:

- ADD
- SUB
- XOR
- AND
- MOD
- SHL
- SHR

```
import sys
sys.path.append("..")
```

```
from circkit import Operation, Param, Circuit

class SimonCircuit(Circuit):
```

(continues on next page)

(continued from previous page)

```
    class Operations(Circuit.Operations):
        class ADD(Operation.Binary):
            SYMMETRIC = True
            def eval(self, a, b):
                return a + b

        class SUB(Operation.Binary):
            SYMMETRIC = True
            def eval(self, a, b):
                return a - b

        class AND(Operation.Binary):
            SYMMETRIC = True
            def eval(self, a, b):
                return a & b

        class XOR(Operation.Binary):
            SYMMETRIC = True
            def eval(self, a, b):
                return a ^ b

        class MOD(Operation.Binary):
            SYMMETRIC = True
            def eval(self, a, b):
                return a % b

        class SHL(Operation.Unary):
            """Shift left"""
            shift: Param.Int()
            def eval(self, a):
                return a << self.shift

        class SHR(Operation.Unary):
            """Shift right"""
            shift: Param.Int()
            def eval(self, a):
                return a >> self.shift

    class Node(Circuit.Node):
        __slots__ = ()

        def __add__(self, b):
            return self.circuit.ADD()(self, b)

        def __radd__(self, b):
            return self.circuit.ADD()(b, self)

        def __sub__(self, b):
            return self.circuit.SUB()(self, b)

        def __rsub__(self, b):
            return self.circuit.SUB()(b, self)
```

(continues on next page)

(continued from previous page)

```
    def __xor__(self, b):
        return self.circuit.XOR()(self, b)

    def __rxor__(self, b):
        return self.circuit.XOR()(b, self)

    def __and__(self, b):
        return self.circuit.AND()(self, b)

    def __rand__(self, b):
        return self.circuit.AND()(b, self)

    def __mod__(self, b):
        return self.circuit.MOD()(self, b)

    def __rmod__(self, b):
        return self.circuit.MOD()(b, self)

    def __lshift__(self, v):
        return self.circuit.SHL(v)(self)

    def __rshift__(self, v):
        return self.circuit.SHR(v)(self)
```

2.5.3 Build a circuit for Simon cipher

```
C = SimonCircuit()
key = C.add_inputs(n=4, format="k%d")
msg = C.add_inputs(n=2, format="m%d")
simon = SimonCipher(key, key_size=64, block_size=32)
cpt = simon.encrypt(msg[0], msg[1])
C.add_output(cpt)
```

Then, we can evaluate the circuit to test its correctness.

```
k3264 = [0x1918, 0x1110, 0x0908, 0x0100]
m3264 = [0x6565, 0x6877]
inp = k3264 + m3264
out = C.evaluate(inp)
print(out)
```

```
[50843, 59835]
```

2.6 Speck Cipher

2.6.1 Implementation of Speck cipher

```
# Reference: https://github.com/inmcm/Simon_Speck_Ciphers/blob/master/Python/
↪simonspeckciphers/speck/speck.py
# For the sake of simplicity, we modified some functions of the original implementation

class SpeckCipher(object):
    """Speck Block Cipher Object"""
    # valid cipher configurations stored:
    # block_size:{key_size:number_rounds}
    __valid_setups = {32: {64: 22},
                      48: {72: 22, 96: 23},
                      64: {96: 26, 128: 27},
                      96: {96: 28, 144: 29},
                      128: {128: 32, 192: 33, 256: 34}}

    def __init__(self, key, key_size=128, block_size=128):

        # Setup block/word size
        self.possible_setups = self.__valid_setups[block_size]
        self.block_size = block_size
        self.word_size = self.block_size >> 1

        # Setup Number of Rounds and Key Size
        self.rounds = self.possible_setups[key_size]
        self.key_size = key_size

        # Create Properly Sized bit mask for truncating addition and left shift outputs
        self.mod_mask = (2 ** self.word_size) - 1

        # Mod mask for modular subtraction
        self.mod_mask_sub = (2 ** self.word_size)

        # Setup Circular Shift Parameters
        if self.block_size == 32:
            self.beta_shift = 2
            self.alpha_shift = 7
        else:
            self.beta_shift = 3
            self.alpha_shift = 8

        assert len(key) == (self.key_size // self.word_size)
        self.key = key

        # Pre-compile key schedule
        self.key_schedule()

    def key_schedule(self):
        self.round_keys = [self.key[-1]]
        l_schedule = [self.key[i] for i in reversed(range(self.key_size // self.word_
```

(continues on next page)

(continued from previous page)

```
↪size - 1))]

        for x in range(self.rounds - 1):
            new_l_k = self.encrypt_round(l_schedule[x], self.round_keys[x], x)
            l_schedule.append(new_l_k[0])
            self.round_keys.append(new_l_k[1])

        return self.round_keys

    def encrypt_round(self, x, y, k):
        """Complete One Round of Feistel Operation"""
        rs_x = ((x << (self.word_size - self.alpha_shift)) + (x >> self.alpha_shift)) &␣
↪self.mod_mask

        add_sxy = (rs_x + y) & self.mod_mask

        new_x = k ^ add_sxy

        ls_y = ((y >> (self.word_size - self.beta_shift)) + (y << self.beta_shift)) &␣
↪self.mod_mask

        new_y = new_x ^ ls_y

        return new_x, new_y

    def decrypt_round(self, x, y, k):
        """Complete One Round of Inverse Feistel Operation"""

        xor_xy = x ^ y

        new_y = ((xor_xy << (self.word_size - self.beta_shift)) + (xor_xy >> self.beta_
↪shift)) & self.mod_mask

        xor_xk = x ^ k

        msub = ((xor_xk - new_y) + self.mod_mask_sub) % self.mod_mask_sub

        new_x = ((msub >> (self.word_size - self.alpha_shift)) + (msub << self.alpha_
↪shift)) & self.mod_mask

        return new_x, new_y

    def encrypt(self, x, y):
        # Run Encryption Steps For Appropriate Number of Rounds
        for k in self.round_keys:
            x, y = self.encrypt_round(x, y, k)

        return x, y

    def decrypt(self, x, y):
        # Run Encryption Steps For Appropriate Number of Rounds
        for k in reversed(self.round_keys):
```

(continues on next page)

(continued from previous page)

```
            x, y = self.decrypt_round(x, y, k)

        return x, y
```

Then, we can test the above implementation with some test vectors given by the authors of Speck

```
k3264 = [0x1918, 0x1110, 0x0908, 0x0100]
m3264 = [0x6574, 0x694c]
c3264 = SpeckCipher(k3264, 64, 32)
t3264 = c3264.encrypt(m3264[0], m3264[1])
assert t3264 == (0xa868, 0x42f2)

k128256 = [0x1f1e1d1c1b1a1918, 0x1716151413121110, 0x0f0e0d0c0b0a0908,␣
↪0x0706050403020100]
m128256 = [0x65736f6874206e49, 0x202e72656e6f6f70]
c128256 = SpeckCipher(k128256, 256, 128)
t128256 = c128256.encrypt(m128256[0], m128256[1])
assert t128256 == (0x4109010405c0f53e, 0x4eeeb48d9c188f43)
```

2.6.2 Define a new circuit type for Simon cipher

By observing the above implementation of Simon cipher, we can see that the new circuit type has to support the following operations on integers:

- ADD
- SUB
- XOR
- AND
- MOD
- SHL
- SHR

```
import sys
sys.path.append("..")
```

```
from circkit.arithmetic import ArithmeticCircuit
from circkit import Operation, Param, Circuit

class SpeckCircuit(Circuit):
    class Operations(Circuit.Operations):
        class ADD(Operation.Binary):
            SYMMETRIC = True
            def eval(self, a, b):
                return a + b

        class SUB(Operation.Binary):
            SYMMETRIC = True
            def eval(self, a, b):
```

(continues on next page)

(continued from previous page)

```
                return a - b

        class AND(Operation.Binary):
            SYMMETRIC = True
            def eval(self, a, b):
                return a & b

        class XOR(Operation.Binary):
            SYMMETRIC = True
            def eval(self, a, b):
                return a ^ b

        class MOD(Operation.Binary):
            SYMMETRIC = True
            def eval(self, a, b):
                return a % b

        class SHL(Operation.Unary):
            """Shift left"""
            shift: Param.Int()
            def eval(self, a):
                return a << self.shift

        class SHR(Operation.Unary):
            """Shift right"""
            shift: Param.Int()
            def eval(self, a):
                return a >> self.shift

    class Node(Circuit.Node):
        __slots__ = ()

        def __add__(self, b):
            return self.circuit.ADD()(self, b)

        def __radd__(self, b):
            return self.circuit.ADD()(b, self)

        def __sub__(self, b):
            return self.circuit.SUB()(self, b)

        def __rsub__(self, b):
            return self.circuit.SUB()(b, self)

        def __xor__(self, b):
            return self.circuit.XOR()(self, b)

        def __rxor__(self, b):
            return self.circuit.XOR()(b, self)

        def __and__(self, b):
            return self.circuit.AND()(self, b)
```

(continues on next page)

(continued from previous page)

```
    def __rand__(self, b):
        return self.circuit.AND()(b, self)

    def __mod__(self, b):
        return self.circuit.MOD()(self, b)

    def __rmod__(self, b):
        return self.circuit.MOD()(b, self)

    def __lshift__(self, v):
        return self.circuit.SHL(v)(self)

    def __rshift__(self, v):
        return self.circuit.SHR(v)(self)
```

2.6.3 Build a circuit for Simon cipher

```
C = SpeckCircuit()
key = C.add_inputs(n=4, format="k%d")
msg = C.add_inputs(n=2, format="m%d")
simon = SpeckCipher(key, key_size=64, block_size=32)
cpt = simon.encrypt(msg[0], msg[1])
C.add_output(cpt)
```

Then, we can evaluate the circuit to test its correctness.

```
k3264 = [0x1918, 0x1110, 0x0908, 0x0100]
m3264 = [0x6574, 0x694c]
inp = k3264 + m3264
out = C.evaluate(inp)
print(out)
```

```
[43112, 17138]
```

2.7 Minimalist quadratic masking transformer

This is an example for the transformer of minimalist quadratic masking scheme which was proposed in BU18

2.7.1 Define the transformer

```
import sys
sys.path.append("../")
```

```
from circkit.transformers.core import CircuitTransformer
from circkit.arithmetic import ArithmeticCircuit
from circkit.array import Array

class BUQuadraticMasking(CircuitTransformer):
    # circuit type of the target circuit
    TARGET_CIRCUIT = ArithmeticCircuit

    def __init__(self):
        """
        Arguments
        ---------
        :order: ISW masking order
        """
        super().__init__()
        # fixed number of shares
        self.n_shares = 3

    def refresh(self, shares, randshares):
        a, b, c = shares
        ra, rb, rc = randshares

        ma = ra * (b + rc)
        mb = rb * (a + rc)
        rc = ma + mb + (ra + rc)*(rb + rc) + rc

        a1 = a + ra
        b1 = b + rb
        c1 = c + rc
        new_shares = Array([a1, b1, c1])
        return new_shares

    def visit_INPUT(self, node):
        shares = []
        for i in range(self.n_shares):
            new_name = f"{node.operation.name}_share{i}"
            x = self.target_circuit.add_input(new_name)
            shares.append(x)
        shares = Array(shares)

        return shares
```

(continues on next page)

(continued from previous page)

```
    def visit_ADD(self, node, shares_1, shares_2):
        ra = self.target_circuit.RND()()
        rb = self.target_circuit.RND()()
        rc = self.target_circuit.RND()()
        randshares_1 = Array([ra, rb, rc])

        rd = self.target_circuit.RND()()
        re = self.target_circuit.RND()()
        rf = self.target_circuit.RND()()
        randshares_2 = Array([rd, re, rf])

        a, b, c = self.refresh(shares_1, randshares_1)
        d, e, f = self.refresh(shares_2, randshares_2)

        x = a + d
        y = b + e
        z = c + f + a*e + b*d

        return Array([x, y, z])

    def visit_MUL(self, node, shares_1, shares_2):
        ra = self.target_circuit.RND()()
        rb = self.target_circuit.RND()()
        rc = self.target_circuit.RND()()
        randshares_1 = Array([ra, rb, rc])

        rd = self.target_circuit.RND()()
        re = self.target_circuit.RND()()
        rf = self.target_circuit.RND()()
        randshares_2 = Array([rd, re, rf])

        a, b, c = self.refresh(shares_1, randshares_1)
        d, e, f = self.refresh(shares_2, randshares_2)

        ma = b*f + rc * e
        md = c*e + rf * b

        x = a*e + rf
        y = b*d + rc
        z = a*ma + d*md + rc*rf + c*f

        return Array([x, y, z])

    def visit_CONST(self, node):
        ra = self.target_circuit.RND()()
        rb = self.target_circuit.RND()()

        x = self.target_circuit.add_const(node.operation.value)
        rx = ra*rb + x

        shares = Array([ra, rb, rx])
        return shares
```

2.7.2 Test on an arithmetic circuit

```
from circkit.arithmetic import ArithmeticCircuit
from sage.all import GF
K = GF(2)

C = ArithmeticCircuit(base_ring=K)
x = C.add_input("x")
y = C.add_input("y")

z = x * y + 1
t = z + x + 1
C.add_output(t)

# ISW transformer
transformer = BUQuadraticMasking()
newC = transformer.transform(C)

# see the graph and verify the ISW circuit
# iswC.digraph().view()

# # Evaluate on original circuit
inp = [1, 0]
out = C.evaluate(inp)
print(f"Original circuit's output: {out}")

# Evaluate on BU quadratic masking circuit
# 1 = 1 * 0 + 1 and 0 = 1 * 1 + 1
inp_shares = [1, 0, 1, 1, 1, 1]
n_tests = 10
for i in range(n_tests):
    out_shares = newC.evaluate(inp_shares)
    a, b, c = out_shares
    ret = a*b + c
    print(f"Output shares: {out_shares} --> {ret}")
```

```
Original circuit's output: [1]
Output shares: [0, 0, 1] --> 1
Output shares: [1, 1, 0] --> 1
Output shares: [1, 1, 0] --> 1
Output shares: [0, 1, 1] --> 1
Output shares: [1, 1, 0] --> 1
Output shares: [0, 1, 1] --> 1
Output shares: [1, 1, 0] --> 1
Output shares: [0, 1, 1] --> 1
Output shares: [1, 0, 1] --> 1
Output shares: [0, 1, 1] --> 1
```

2.8 circkit.operation

Operations may include arbitrary parameters. It is important to distinguish *parameters* (arbitrary objects) from node's *inputs* (other nodes in the same circuit).

An `Operation` *(sub)class* describes an operation and all its parameters, including validation of parameters, number of node's inputs and outputs.

An `Operation` *instance* describes a concrete operation with all parameters set. It does not however define a node in a circuit or node's inputs/outputs. One Operation instance can be used by multiple nodes.

The node/operation creation API is a double-step call (similar to e.g. Keras Layers API):

```
x = circuit.INPUT("x")()
xcube = circuit.EXP(3)(x)

assert issubclass(circuit.EXP, Operation)
assert isinstance(circuit.EXP(3), Operation)
assert isinstance(circuit.EXP(3)(x), circuit.Node)
```

Here, both lines follow this API, in which the first call creates an `Operation` instance and the second call creates a `circkit.node.Node` instance.

In the first line, we first create the `Circuit.INPUT` *operation*, which has the input name "x" as the only parameter: `circuit.INPUT("x")`. Then, we call this operation without any incoming nodes to create a new node in the circuit, corresponding to the input "x".

In the second line, we first create the *EXP* operation with the parameter *power = 3*. We then call this operation with a node as an argument which defines the incoming node for exponentitation.

The reason for the double-call is to separate clearly the two different notions - parameters and inputs, and to allow using the full python's call syntax. For example, we could use keyword args such as `Circuit.EXP(power=3)`. This is particularly useful for complex operations to reduce the number of errors e.g. in parameter/input order.

Defining operation example:

```
class NewCircuitType(Circuit):
    class Operations(Circuit.Operations):  # subclass to include
                                           # standard INPUT, GET, CONST
        class ADD(Operation.Binary):
            def eval(self, a, b):
                return a + b

        class SUB(Operation.Binary):
            SYMMETRIC = False
            def eval(self, a, b):
                return a - b

        class MIX(Operation.MultiVariadic):
            alpha : Param.Int(min_value=1) = 2

            def determine_n_outputs(self, node):
                return len(node.incoming)

            def eval(self, *args):
                t = self.alpha * sum(args)
                return [a - t for a in args]
```

2.8.1 Module contents

`circkit.operation.VARIABLE = <VARIABLE>`

Special *Operation.n_inputs* / *Operation.n_outputs* value - unspecified number of inputs/outputs (or determined dynamically).

`circkit.operation.UNIT = <UNIT>`

Special *Operation.n_outputs* value - non-iterable output.

class `circkit.operation.OperationMeta`(*clsname*, *bases*, *attrs*)

Bases: `type`

Metaclass for the *Operation* class.

Manages definition of operations (classes) and creation of operation instances (e.g. preparing parameters, caching operatios).

static `__new__`(*meta*, *clsname*, *bases*, *attrs*)

Define a new Operation class (in a circuit class).

1. Parse definitions of parameters from the annotations and assignments (defaults).
2. Update `__slots__` to include new parameters.
3. Collects parameters from superclasses. If such a parameter attribute is overriden by a non-parameter attribute, the parameter is excluded. This allows e.g. to remove superclass' parameter by setting attribute `param = None`.
4. Create dynamically new *Operation* class with given slots and parameters descriptions.

`__call__`(**values*, ***kvalues*)

Create an instance of *Operation*.

Currently, only implements caching of `Operation`s (if enabled), and passes all the parameters to the constructor of the :class:`Operation`.

class `circkit.operation.Operation`(**values*, ***kvalues*)

Bases: `object`

Describes a (parametrized) operation that can be used in a circuit.

Parameters of an *Operation* instance are accessible directly as attributes.

All supporting attributes are either prefixed with "_" or are UPPERCASE to avoid collisions with possible parameter names. Exceptions: *n_inputs* and *n_outputs*.

Defining attributes (may/should be overwritten in subclasses)

`n_inputs`

Number of node inputs the operation requires.

Type
int

`n_outputs`

Number of node outputs the operation has (to be retrieved with the *Circuit.Operations.GET* operation). By default, is set to the special object *UNIT*, which means that the output is non iterable. Note that this is different from `n_outputs = 1`, where the actual output has to be retrieved e.g. as `out = node[0]` or *out, = node*.

Type
int = UNIT

`SYMMETRIC`

Whether the order of the input nodes does not matter. Used e.g. for caching nodes (`b + a` may return the cached node `a + b`).

Type
bool

`PRECOMPUTABLE`

Whether the operation is precomputable (given the inputs and the parameters).

Type
bool

`STR_LIMIT`

Maximum length of the string describing parameters to keep (for `__str__()`).

Type
int = 30

`Functional attributes (should not be overriden manually)`

`_circuit`

Circuit instance in which this `Operation` class/instance is defined/subclassed.

Type
`Circuit`

`_circuit_class`

Circuit class in which this `Operation` class/instance is defined/subclassed.

Type
`Circuit`

`_param_descriptions`

Mapping from parameter names to `Param` objects describing the parameters.

Type
*dict`[str, :class:*.Param`]

`__init__`(**values*, ***kvalues*)

Create an `Operation` *instance* by specifying parameters (if any).

Note that it is not linked to any `Node` yet.

`__call__`(**incoming*, ***kwargs*)

Create a new node using this operation.

`reapply`(*circuit*, *name=None*)

Create new operation instance with same parameters (by name), but in the other circuit. @name define name of the operation, by default it is the name of this operation.

`__eq__`(*other*)

Test equality of two operations.

Two operations are equal if their names are equal and all parameters are equal.

To check the class of the operation (e.g. INPUT, ADD, CONST, etc.) use:

- `isinstance(op, CircuitClass.ADD)` : checks circuit class;
- `isinstance(op, CircuitInstance.ADD)` : checks circuit instance too;

- `op.is_ADD()` : ADD must be present in the circuit's class;
- `op._name == "ADD"` : by name, does not check circuit class/instance.

  Parameters
  other (`Operation`) –

`__hash__()`
Return hash(self).

`__str__()`
Return str(self).

`__repr__()`
Return repr(self).

`eval_with_node(node, *args)`
This method should be overriden if the evaluation requires some information from the node. By default, it ignores the node and calls `eval()`.

`eval(*args)`
Evaluate the operation on given inputs (typically values, not nodes).

`before_create_node(*args)`
Validate node inputs before creating a new node with this operation.

`after_create_node(node)`
Check new node after creation (node uses this operation).

`classmethod on_new_circuit(circuit)`
Callback on linking the operation class to a new circuit instance.

E.g. can store common circuit-level data in the circuit instance.

`determine_n_outputs(node)`
Determine number of outputs of a new node using this operation.

By default, returns operation's `n_outputs`. However, this method must be overriden for multi-operations. For example, the number of outputs may be set equal to the number of inputs.

Return type
int

`__reduce__()`
Helper for pickle.

`__setstate__(src)`
To update class.__dict__ with dict/map

`class Binary(*values, **kvalues)`
Bases: `Operation`

Operation with 2 inputs.

`class MultiBinary(*values, **kvalues)`
Bases: `Operation`

Operation with 2 inputs and (possibly) multiple number of outputs.

class **MultiNullary**(**values*, ***kvalues*)

Bases: *Operation*

Operation with no inputs and (possibly) multiple number of outputs.

class **MultiTernary**(**values*, ***kvalues*)

Bases: *Operation*

Operation with 3 inputs and (possibly) multiple number of outputs.

class **MultiUnary**(**values*, ***kvalues*)

Bases: *Operation*

Operation with 1 input and (possibly) multiple number of outputs.

class **MultiVariadic**(**values*, ***kvalues*)

Bases: *Operation*

Operation with variable number of inputs and (possibly) multiple number of outputs.

class **Nullary**(**values*, ***kvalues*)

Bases: *Operation*

Operation with no inputs.

class **Ternary**(**values*, ***kvalues*)

Bases: *Operation*

Operation with 3 inputs.

class **Unary**(**values*, ***kvalues*)

Bases: *Operation*

Operation with 1 inputs.

class **Variadic**(**values*, ***kvalues*)

Bases: *Operation*

Operation with variable number of inputs.

class circkit.operation.**Nullary**(**values*, ***kvalues*)

Bases: *Operation*

Operation with no inputs.

class circkit.operation.**Unary**(**values*, ***kvalues*)

Bases: *Operation*

Operation with 1 inputs.

class circkit.operation.**Binary**(**values*, ***kvalues*)

Bases: *Operation*

Operation with 2 inputs.

class circkit.operation.**Ternary**(**values*, ***kvalues*)

Bases: *Operation*

Operation with 3 inputs.

class `circkit.operation.`**`Variadic`**`(`**values*, ***kvalues*`)`

Bases: *`Operation`*

Operation with variable number of inputs.

class `circkit.operation.`**`MultiNullary`**`(`**values*, ***kvalues*`)`

Bases: *`Operation`*

Operation with no inputs and (possibly) multiple number of outputs.

class `circkit.operation.`**`MultiUnary`**`(`**values*, ***kvalues*`)`

Bases: *`Operation`*

Operation with 1 input and (possibly) multiple number of outputs.

class `circkit.operation.`**`MultiBinary`**`(`**values*, ***kvalues*`)`

Bases: *`Operation`*

Operation with 2 inputs and (possibly) multiple number of outputs.

class `circkit.operation.`**`MultiTernary`**`(`**values*, ***kvalues*`)`

Bases: *`Operation`*

Operation with 3 inputs and (possibly) multiple number of outputs.

class `circkit.operation.`**`MultiVariadic`**`(`**values*, ***kvalues*`)`

Bases: *`Operation`*

Operation with variable number of inputs and (possibly) multiple number of outputs.

exception `circkit.operation.`**`UnhashableOperationError`**

Bases: `Exception`

Raised when an *`Operation`* is being hashed but hashing is not defined (e.g. some parameters are unhashable objects).

`__weakref__`

list of weak references to the object (if defined)

2.9 circkit.param

class `circkit.param.`**`Param`**

Bases: `object`

Describes a type of a parameter of an operation,
mainly its validation/conversion.

JW: how about rename this as *ParameterConstraint*

Default (this type): no validation/conversion

exception **`InvalidConstraint`**

Bases: `Exception`

exception **`InvalidValue`**

Bases: `Exception`

`Bool`

alias of *`BoolParam`*

Const
alias of *ConstParam*

InputName
alias of *InputNameParam*

Int
alias of *IntParam*

Str
alias of *StrParam*

Tuple
alias of *TupleParam*

class circkit.param.**ConstParam**
Bases: *Param*

class circkit.param.**IntParam**(*, *min_value=None*, *max_value=None*)
Bases: *Param*

Parameters

- **min_value** (*int*) –
- **max_value** (*int*) –

__init__(*, *min_value=None*, *max_value=None*)

Parameters

- **min_value** (*Optional[int]*) –
- **max_value** (*Optional[int]*) –

class circkit.param.**BoolParam**
Bases: *Param*

class circkit.param.**StrParam**
Bases: *Param*

class circkit.param.**TupleParam**
Bases: *Param*

class circkit.param.**InputNameParam**
Bases: *Param*

Accepts any subclass of str,int or arbitrarily nested tuple of those

2.10 circkit.circuit

class circkit.circuit.**BaseCircuit**(**args*, ***kwargs*)
Bases: `object`

Base circuit class, defining all the "magic" of `Operation`s and :class:`.Node`s.

Responsible for tracking *Operation* classes, dynamically subclassing *Operation* and *Node* classes, and copying/pickling.

class Operations

Bases: `object`

Container of Operation classes. Apriori is not linked to any Circuit class, can be reused/inherited from.

class Node(*operation*, *incoming*)

Bases: `object`

Node in a circuit.

__init__(*operation*, *incoming*)

is_OUTPUT()

Returns whether this node is an output node in its circuit.

reapply(**incoming*, *circuit=None*, *inherit_info=True*, *auto_output=False*)

Apply the same operation to new *incoming* nodes/values in th. # AU: auto_output option? (outputs in new circuit if is output here)

siblings_by_outgoing(*node=None*)

Find the siblings of current by outgoing nodes (or by the single @node node)

class circkit.circuit.**Circuit**(**args*, ***kwargs*)

Bases: `BaseCircuit`

Main Circuit class, includes skeleton structure and common methods, and the basic INPUT, CONST, GET nodes.

class Operations

Bases: `Operations`

class INPUT(**values*, ***kvalues*)

Bases: `Operation`

Input node. Parameters:

- name (str): name of the input

__init__(*name*, **args*, ***kwargs*)

Create an `Operation` *instance* by specifying parameters (if any).

Note that it is not linked to any `Node` yet.

classmethod on_new_circuit(*circuit*)

Callback on linking the operation class to a new circuit instance.

E.g. can store common circuit-level data in the circuit instance.

before_create_node()

Validate node inputs before creating a new node with this operation.

eval(*value*)

Evaluate the operation on given inputs (typically values, not nodes).

class CONST(**values*, ***kvalues*)

Bases: `Operation`

Constant node. Parameters:

- value (circuit constant type): constant value to use

eval()

Evaluate the operation on given inputs (typically values, not nodes).

class GET(**values*, ***kvalues*)

Bases: `Operation`

Getter node for multi-output nodes. Parameters:
- index (int): index of the element to get

before_create_node(*node*)

Validate node inputs before creating a new node with this operation.

eval(*x*)

Evaluate the operation on given inputs (typically values, not nodes).

__init__(*base_ring=None*, *name=None*, ***kwargs*)

in_place_remove_unused_nodes()

WARNING: modifies the circuit (and nodes) in place

trace(*input*, *convert_input=True*, *convert_values=True*, *as_list=False*)

Trace the circuit execution on a given input.

Parameters

- **input** (*list[values]*) – List of input values, one per input node.
- **convert_input** (*bool = True*) – Convert input values using circuits `ConstManager`?
- **convert_values** (*bool = True*) – Convert traced values using circuits `ConstManager`?
- **as_list** (*bool = False*) – Output as list instead (with order defined by the circuit's order)?

Returns

trace

Return type

dict[`Node`, value]

evaluate(*input*, *convert_input=True*, *convert_output=True*, *with_mem=False*)

convert_input validates and converts input to the base_ring
element should be disabled e.g. for evaluating on Arrays (bit-sliced fashion), or for evaluating on symbolic objects (e.g. another circuit nodes).

convert_output converts the output to a simple value, defined by
ConstManager maybe later we could specify different formats for conversion.

concat_on_same_inputs(**circuits*, *name=None*)

Concatenate two or more circuits by reusing the inputs

concat_parallel(**circuits*, *name=None*)

Concatenate two or more circuits fully in parallel

print_stats(*function=<built-in function print>*, *tab=' | '*, *exclude=[]*, *by_address=False*)

Shows basic information about the circuit.

Example

```
circuit.print_stats()
```

output:

```
circuit_name(ArithmeticCircuit):
32 inputs,   129 outputs,    8487 nodes
```

class CONST(**values*, ***kvalues*)

Bases: *CONST*

class GET(**values*, ***kvalues*)

Bases: *GET*

class INPUT(**values*, ***kvalues*)

Bases: *INPUT*

Node

alias of *Node*

Node_unlinked

alias of *Node*

2.11 circkit.node

class circkit.node.**Node**(*operation*, *incoming*)

Bases: object

Node in a circuit.

__init__(*operation*, *incoming*)

is_OUTPUT()

Returns whether this node is an output node in its circuit.

siblings_by_outgoing(*node=None*)

Find the siblings of current by outgoing nodes (or by the single @node node)

reapply(**incoming*, *circuit=None*, *inherit_info=True*, *auto_output=False*)

Apply the same operation to new *incoming* nodes/values in th. # AU: auto_output option? (outputs in new circuit if is output here)

2.12 circkit.const_manager

class circkit.const_manager.**ArithmeticConstManager**(*circuit*)

Bases: ConstManager

Generic manager for arithmetic constants. Should cover most cases, including SageMath's Zmod, GF(p), GF(q)

For validation:

the input value should be of type int or .parent() should return the base_ring

For conversion:
int -> const tries base_ring.fetch_int(int), otherwise base_ring(int)

const -> int tries base_ring.integer_representation(int), otherwise int(const)

`__init__`(*circuit*)

`create`(*value*)

Create a constant of unified type from value of possibly various types

Default implementation: no checks, no conversion

`output`(*value*)

Convert a constant of unified type to simple representation (e.g. int or string?) AU: maybe have different output formats?

Default implementation: no conversion

2.13 circkit.arithmetic

class `circkit.arithmetic.`**`Table`**(*iterable=()*, */*)

Bases: `tuple`

class `circkit.arithmetic.`**`ArithmeticCircuit`**(**args*, ***kwargs*)

Bases: *`Circuit`*

Generic class for arithmetic circuits.

class **`Operations`**

Bases: *`Operations`*

Class gathering `Operation`s of :class:`.ArithmeticCircuit`.

class **`ADD`**(**values*, ***kvalues*)

Bases: *`Binary`*

`eval`(*a*, *b*)

Evaluate the operation on given inputs (typically values, not nodes).

class **`SUB`**(**values*, ***kvalues*)

Bases: *`Binary`*

`eval`(*a*, *b*)

Evaluate the operation on given inputs (typically values, not nodes).

class **`MUL`**(**values*, ***kvalues*)

Bases: *`Binary`*

`eval`(*a*, *b*)

Evaluate the operation on given inputs (typically values, not nodes).

class **`DIV`**(**values*, ***kvalues*)

Bases: *`Binary`*

`eval`(*a*, *b*)

Evaluate the operation on given inputs (typically values, not nodes).

`class EXP(*values, **kvalues)`

Bases: `Unary`

`eval(a)`

Evaluate the operation on given inputs (typically values, not nodes).

`class INV(*values, **kvalues)`

Bases: `Unary`

`eval(a)`

Evaluate the operation on given inputs (typically values, not nodes).

`class NEG(*values, **kvalues)`

Bases: `Unary`

`eval(a)`

Evaluate the operation on given inputs (typically values, not nodes).

`class LUT(*values, **kvalues)`

Bases: `Unary`

`eval(idx)`

Evaluate the operation on given inputs (typically values, not nodes).

`class RND(*values, **kvalues)`

Bases: `Nullary`

`eval()`

Evaluate the operation on given inputs (typically values, not nodes).

`Node`

alias of `Node`

`to_matrix(zero=0, one=1, n_tests=0)`

Attempts to express the circuit as an affine map C(x) = A*x + b, where A is a matrix b is a vector, b = 0 when the map is linear

Done by evaluating the circuit at unit-vector inputs and at zero vector. Does not verify that the circuit is actually linear.

Note: current implementation does batch execution using Array inputs
may be doing single calls is more reliable for some circuit variants.

Todo: add option for non-batch execution

Parameters

- **zero** (*const*) – constant representing zero
- **one** (*const*) – constant representing one

Returns
matrix A Array: vector b

Return type
Matrix

class ADD(*values, **kvalues)

Bases: *ADD*

class CONST(*values, **kvalues)

Bases: *CONST*

class DIV(*values, **kvalues)

Bases: *DIV*

class EXP(*values, **kvalues)

Bases: *EXP*

class GET(*values, **kvalues)

Bases: *GET*

class INPUT(*values, **kvalues)

Bases: *INPUT*

class INV(*values, **kvalues)

Bases: *INV*

class LUT(*values, **kvalues)

Bases: *LUT*

class MUL(*values, **kvalues)

Bases: *MUL*

class NEG(*values, **kvalues)

Bases: *NEG*

Node_unlinked

alias of *Node*

class RND(*values, **kvalues)

Bases: *RND*

class SUB(*values, **kvalues)

Bases: *SUB*

class circkit.arithmetic.**OptArithmeticCircuit**(*args, **kwargs)

Bases: *ArithmeticCircuit*

Optimized arithmetic circuits.

Optimizations include:

- caching operations and nodes;
- precomputing annihilator operations, e.g. a*0 = 0;
- precomputing identity operations, e.g. a*1 = a, a+0 = 0;
- precomputing constant operations;

Node

alias of *Node*

class ADD(*values, **kvalues)

Bases: *ADD*

class CONST(**values*, ***kvalues*)

Bases: *CONST*

class DIV(**values*, ***kvalues*)

Bases: *DIV*

class EXP(**values*, ***kvalues*)

Bases: *EXP*

class GET(**values*, ***kvalues*)

Bases: *GET*

class INPUT(**values*, ***kvalues*)

Bases: *INPUT*

class INV(**values*, ***kvalues*)

Bases: *INV*

class LUT(**values*, ***kvalues*)

Bases: *LUT*

class MUL(**values*, ***kvalues*)

Bases: *MUL*

class NEG(**values*, ***kvalues*)

Bases: *NEG*

Node_unlinked

alias of *Node*

class RND(**values*, ***kvalues*)

Bases: *RND*

class SUB(**values*, ***kvalues*)

Bases: *SUB*

2.14 circkit.bitwise

class circkit.bitwise.circuit.**BitwiseCircuit**(**args*, ***kwargs*)

Bases: *Circuit*

Generic class for bitwise circuits (unsigned words).

DEFAULT_BASE_RING

alias of *BitwiseRing*

class Operations

Bases: *Operations*

Class gathering `Operation`s of :class:`.ArithmeticCircuit`.

class AND(**values*, ***kvalues*)

Bases: *Binary*

eval(*a*, *b*)

Evaluate the operation on given inputs (typically values, not nodes).

class OR(**values*, ***kvalues*)

Bases: *Binary*

eval(*a*, *b*)

Evaluate the operation on given inputs (typically values, not nodes).

class XOR(**values*, ***kvalues*)

Bases: *Binary*

eval(*a*, *b*)

Evaluate the operation on given inputs (typically values, not nodes).

class NOT(**values*, ***kvalues*)

Bases: *Unary*

eval(*a*)

Evaluate the operation on given inputs (typically values, not nodes).

class SHL(**values*, ***kvalues*)

Bases: *Unary*

Shift left

eval(*a*)

Evaluate the operation on given inputs (typically values, not nodes).

class SHR(**values*, ***kvalues*)

Bases: *Unary*

Shift right

eval(*a*)

Evaluate the operation on given inputs (typically values, not nodes).

class ROL(**values*, ***kvalues*)

Bases: *Unary*

Rotate left

eval(*a*)

Evaluate the operation on given inputs (typically values, not nodes).

class ROR(**values*, ***kvalues*)

Bases: *Unary*

Rotate left

eval(*a*)

Evaluate the operation on given inputs (typically values, not nodes).

class ADD(**values*, ***kvalues*)

Bases: *Binary*

eval(*a*, *b*)

Evaluate the operation on given inputs (typically values, not nodes).

class SUB(**values*, ***kvalues*)

Bases: *Binary*

eval(*a*, *b*)

Evaluate the operation on given inputs (typically values, not nodes).

class MUL(**values*, ***kvalues*)

Bases: `Binary`

eval(*a*, *b*)

Evaluate the operation on given inputs (typically values, not nodes).

class DIV(**values*, ***kvalues*)

Bases: `Binary`

eval(*a*, *b*)

Evaluate the operation on given inputs (typically values, not nodes).

class NEG(**values*, ***kvalues*)

Bases: `Unary`

eval(*a*)

Evaluate the operation on given inputs (typically values, not nodes).

class LUT(**values*, ***kvalues*)

Bases: `Variadic`

eval(**args*)

Evaluate the operation on given inputs (typically values, not nodes).

class RND(**values*, ***kvalues*)

Bases: `Nullary`

eval()

Evaluate the operation on given inputs (typically values, not nodes).

Node

alias of `Node`

__init__(**args*, *word_size=None*, ***kwargs*)

class ADD(**values*, ***kvalues*)

Bases: `ADD`

class AND(**values*, ***kvalues*)

Bases: `AND`

class CONST(**values*, ***kvalues*)

Bases: `CONST`

class DIV(**values*, ***kvalues*)

Bases: `DIV`

class GET(**values*, ***kvalues*)

Bases: `GET`

class INPUT(**values*, ***kvalues*)

Bases: `INPUT`

class LUT(**values*, ***kvalues*)

Bases: `LUT`

class MUL(**values*, ***kvalues*)

Bases: *MUL*

class NEG(**values*, ***kvalues*)

Bases: *NEG*

class NOT(**values*, ***kvalues*)

Bases: *NOT*

Node_unlinked

alias of *Node*

class OR(**values*, ***kvalues*)

Bases: *OR*

class RND(**values*, ***kvalues*)

Bases: *RND*

class ROL(**values*, ***kvalues*)

Bases: *ROL*

class ROR(**values*, ***kvalues*)

Bases: *ROR*

class SHL(**values*, ***kvalues*)

Bases: *SHL*

class SHR(**values*, ***kvalues*)

Bases: *SHR*

class SUB(**values*, ***kvalues*)

Bases: *SUB*

class XOR(**values*, ***kvalues*)

Bases: *XOR*

class circkit.bitwise.circuit.**BooleanCircuit**(**args*, ***kwargs*)

Bases: *BitwiseCircuit*

class Operations

Bases: *Operations*

SHL = None

SHR = None

ROL = None

ROR = None

class ADD(**values*, ***kvalues*)

Bases: *ADD*

class AND(**values*, ***kvalues*)

Bases: *AND*

class CONST(**values*, ***kvalues*)

Bases: *CONST*

class `DIV`(**values*, ***kvalues*)

Bases: *DIV*

class `GET`(**values*, ***kvalues*)

Bases: *GET*

class `INPUT`(**values*, ***kvalues*)

Bases: *INPUT*

class `LUT`(**values*, ***kvalues*)

Bases: *LUT*

class `MUL`(**values*, ***kvalues*)

Bases: *MUL*

class `NEG`(**values*, ***kvalues*)

Bases: *NEG*

class `NOT`(**values*, ***kvalues*)

Bases: *NOT*

Node

alias of *Node*

Node_unlinked

alias of *Node*

class `OR`(**values*, ***kvalues*)

Bases: *OR*

class `RND`(**values*, ***kvalues*)

Bases: *RND*

class `SUB`(**values*, ***kvalues*)

Bases: *SUB*

class `XOR`(**values*, ***kvalues*)

Bases: *XOR*

class `circkit.bitwise.ring.BitwiseRing`(*word_size*)

Bases: `object`

Ring of fixed-width bit-words.

`__init__`(*word_size*)

`fetch_int`(*value*)

Call self as a function.

class `Word`(*value*, *ring*)

Bases: `object`

Wrapper for a bit-word.

Parameters

- **value** (*int*) –
- **ring** (`BitwiseRing`) –

`__init__`(*value*, *ring*)

Parameters

- **`value`** (*`int`*) –
- **`ring`** (`BitwiseRing`) –

`class circkit.bitwise.ring.`**`Word`**(*value*, *ring*)

Bases: `object`

Wrapper for a bit-word.

Parameters

- **`value`** (*`int`*) –
- **`ring`** (`BitwiseRing`) –

`__init__`(*value*, *ring*)

Parameters

- **`value`** (*`int`*) –
- **`ring`** (`BitwiseRing`) –

2.15 circkit.transformers

`class circkit.transformers.core.`**`Transformer`**

Bases: `object`

Base transformer class.

`before_visit`(*node*)

Event handler before visiting node

`after_visit`(*node*)

Event handler after visiting node

`class circkit.transformers.core.`**`CircuitTransformer`**

Bases: *`Transformer`*

Base class for circuit->circuit transformers.

`make_output`(*node*, *result*)

Default implementation: mark images of output notes as outputs in new circuit.

2.16 circkit.location

`class circkit.location.`**`Location`**(*iterable=()*, */*)

Bases: `tuple`

CHAPTER

THREE

FRAMEWORK VS DSL

There exist already several *domain specific languages* (DSL) for cryptographic primitives. A nice overview is given by Darius Mercadier: DSLs for Cryptography.

We decided to develop a python framework instead of a DSL to exploit the full power of a mature programming language to create abstractions for designing and working with circuits. Furthermore, the ability to overload arithmetic operations makes the framework as expressive as a DSL could be.

CHAPTER

FOUR

AUTHORS

Matthieu Rivain, Aleksei Udovenko and Junwei Wang.

CHAPTER

FIVE

LICENSE

`circkit` is available under the MIT license.

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

P

R

S

T

U

V

W