PAnTHErS: User Guide

Cyrielle Feron

PAnTHErS: User Guide

Installation & Tutorials

Version 1.0

September 3, 2018

Cyrielle FERON cyrielle.feron@ensta-bretagne.org

About this document

PAnTHErS is a tool written in Python, intented to evaluate Homomorphic Encryption Schemes (HE Schemes). Its goal is to provide a fast insight of execution time and memory consumption for an application using homomorphic encryption.

PAnTHErS can then help users to choose the optimal HE Scheme to use for a given application. It also helps to choose the best parameters to use for a given HE Scheme.

This document is decomposed as follows:

- Chapter 1 explains how to install PAnTHErS and prerequisites.
- Chapter 2 presents how to use PAnTHErS throught its graphical interface.
- Chapter 3 details how a user can add a new HE schemes into PAnTHErS library.
- Chapter 4 details how a user can add a new application into PAnTHErS library.

Contact

Cyrielle FERON - cyrielle.feron@ensta-bretagne.org

Loïc LAGADEC - loic.lagadec@ensta-bretagne.fr

Contents

Chapter 1

PAnTHErS Installation

1.1 Getting Started

Here are the few steps required to get PAnTHErS up and running on your system.

1.1.1 Operating system

The install process described here have been tested on a stock 64 bits Debian Linux 9.5 system. The base system is installed using the **Debian netinst installer**.

1.1.2 Install required software

Sage

Download Sage installer from SageMath website.

At the time we write this Readme, the latest Sage version is 8.2. You may try later versions of Sage, but the final result cannot be guaranteed.

Note that PAnTHErS was originally developped using Sage 7.6.

Here we download Sage 8.2 64 bits for Debian Linux 9 **sage-8.2-Debian*GNU*Linux*9-x86*64.tar.bz2** (1658.98 MB) dated from 2018-05-08 22:01.

MD5: fd83b2b63699b90c41e74e9988c705d2

For this installation process, we followed the official Sage installation guidelines.

Sage Software requirements

See Sage installation guide for more details

```
$> sudo apt update
$> sudo apt install binutils gcc make m4 perl tar git openssl libssl-dev
```

PAnTHErS interface uses Tk, the following package must be installed:

```
$> sudo apt install tk tk-dev
```

From a terminal, extract the Sage archive:

```
$> tar -jxvf sage-8.2-Debian_GNU_Linux_9-x86_64.tar.bz2
```

On the local folder, a SageMath directory is created

```
$> cd SageMath
```

Then start the build. To accelerate the build, you can add extra build jobs by providing a `-j N` option to the make command. Here we start the build with two jobs :

```
$> make -j 2
```

Note: the build process is very time consuming, it can last several hours.

1.2 PAnTHErS installation

1.2.1 Get the code

Download PAnTHErS code on your local file system.

```
        $> git clone https://github.com/cferon/PAnTHErS
OR
        $> wget https://github.com/cferon/PAnTHErS/archive/master.zip
        $> unzip master.zip
        $> mv PAnTHErS-master PAnTHErS
```

Then create a symbolic link named `panthers` on your home directory. This symbolic link is required for PAnTHErS graphical interface.

```
        $> cd ~
        $> ln -s /path/to/PAnTHErS panthers
```

1.2.2 Starting PAnTHErS

Go to your Sage build folder and start Sage with the following command:

```
$> cd /path/to/SageMath
$> ./sage
```

From Sage prompt, go to PAnTHErS interface folder and launch the interface:

```
$sage: cd ~
$sage: cd panthers/Interface
$sage: load("interface.py")
```

PAnTHErS interface should pop-up.

Note: current interface is pretty wide, a screen resolution at least 1600 pixels wide is required to be able to display the graphical interface.

1.3 Troubleshooting

1.3.1 User is not in sudoers file

To be able to run commands as root using the sudo command, a user needs to be registered in the sudoers file.

To do this, you first need to get a root terminal prompt:

```
$> su
```

Type your root password here, then from the root prompt:

```
#> visudo
```

You get to the sudoers file editor.

Below the line:

```
root        ALL=(ALL) ALL
```

Add your user name (here the user is named "user"):

```
root        ALL=(ALL) ALL
user        ALL=(ALL) ALL
```

Save the file and quit the editor.

The user is now able to run commands as root using *sudo*

1.3.2 You want to add Tk support to your already built Sage environment

As explained in the installation guide, you first need to download the Tk package:

```
$> sudo apt install tk tk-dev
```

Then rebuild Sage's Python:

```
$> sage -f python2  # rebuild Python
$> make             # rebuild components of Sage depending on Python
```

1.4 Copyright

Chapter 2

PAnTHErS HowTo

2.1 Prerequisites

First get a PAnTHErS installation up and ready, following instructions from README.md file.

2.2 Notations

HE scheme: Homomorphic Encryption scheme

2.3 Interface Startup

Begin by starting your PAnTHErS interface:

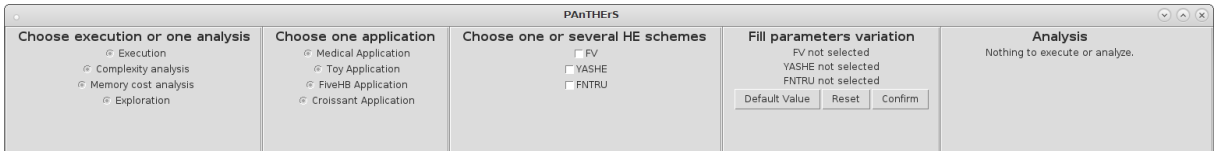
Go to your Sage build folder and start Sage with the following command:

```
$> cd /path/to/SageMath
$> ./sage
```

From Sage prompt, go to PAnTHErS interface folder and launch the interface:

```
$sage: cd ~
$sage: cd panthers/Interface
$sage: load("interface.py")
```

PAnTHErS interface should pop-up.



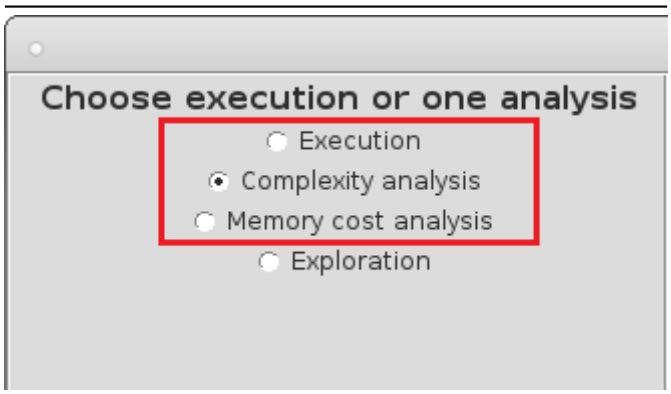
PAnTHErS Interface at startup

2.4 Application Analysis or Execution

This part shows how to use PAnTHErS Interface in order to analyze or execute an application using one or several HE schemes.

2.4.1 Analysis selection

Select the kind of analysis you want to perform from the first interface pane:



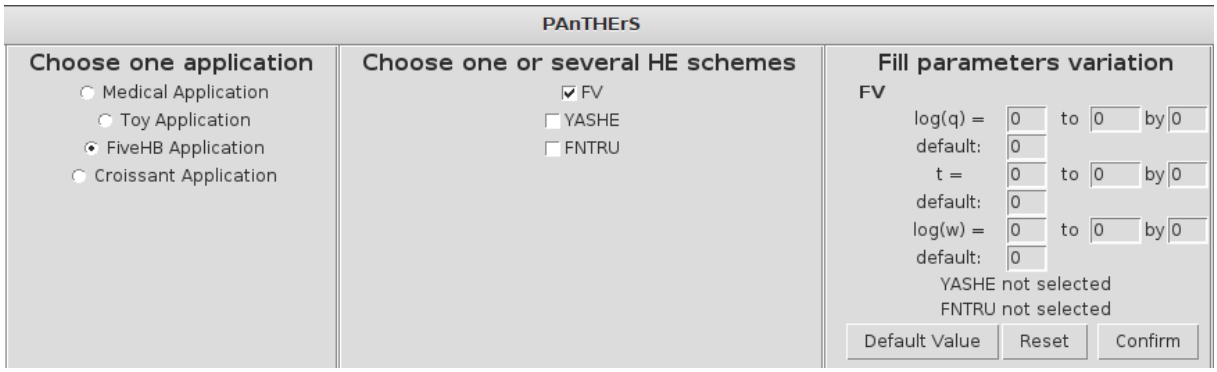
Analysis selection from first pane

You can either choose:

- **Execution**: Execute all parameters combinations for a given application and HE schemes selection.
- **Complexity analysis**: Get an estimation of execution time in seconds required for all parameters combinations for a given application and HE schemes selection.
- **Memory cost analysis**: Get an estimation of memory cost in mebibytes (MiB) required for all parameters combinations for a given application and HE schemes selection.

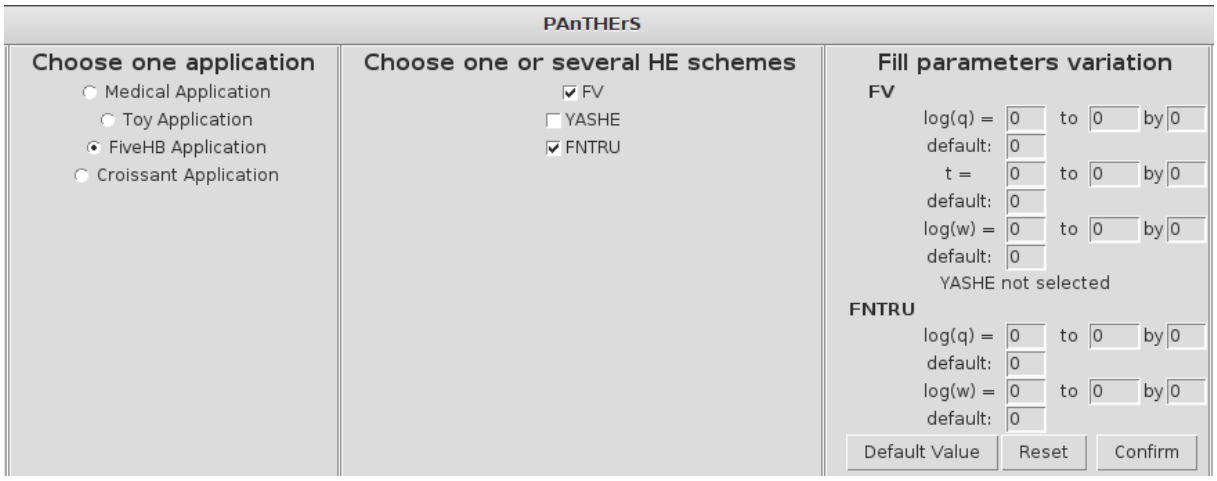
2.4.2 Application and HE scheme(s) selection

Select the application and HE schemes to analyze or execute from the second and third interface pane:



Application and HE schemes selection: one scheme selected

You can either choose one or more HE schemes. For each scheme selected, its parameters show up in the 4th interface pane.



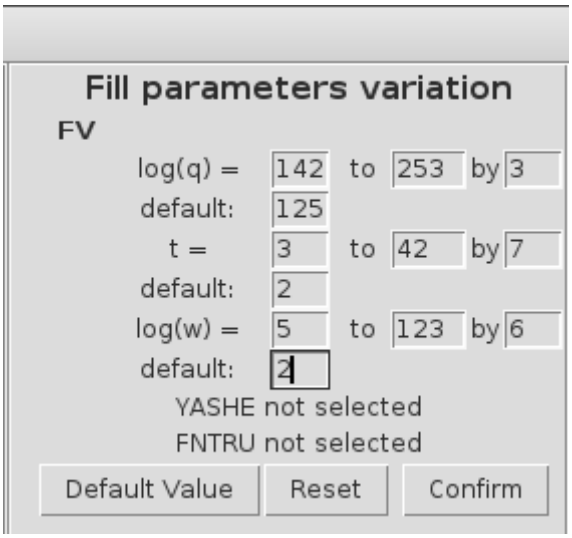
Application and HE schemes selection: several schemes selected

2.4.3 Fill HE schemes parameters variation

Fill HE schemes parameters with default values and ranges of values.

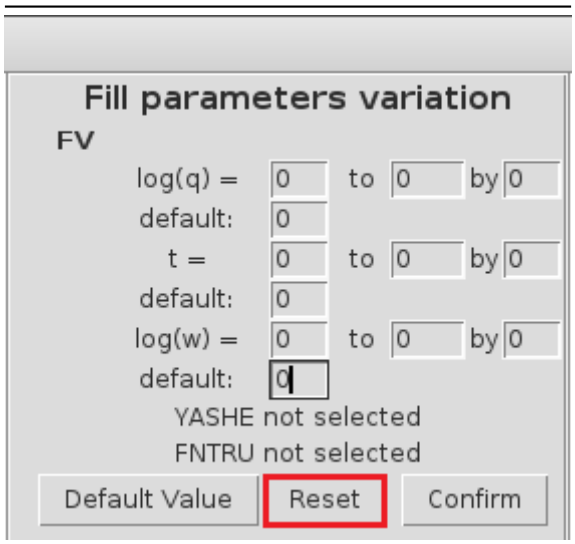
> ⚠ **Important:** You need at least one parameter with a range of values per HE scheme.

> ⚠ **Important:** For each parameter value you have to provide a default value. This value is used when the HE scheme is executed or analyzed using varying values for another parameter.



Selection of custom parameter values or range of values for HE schemes

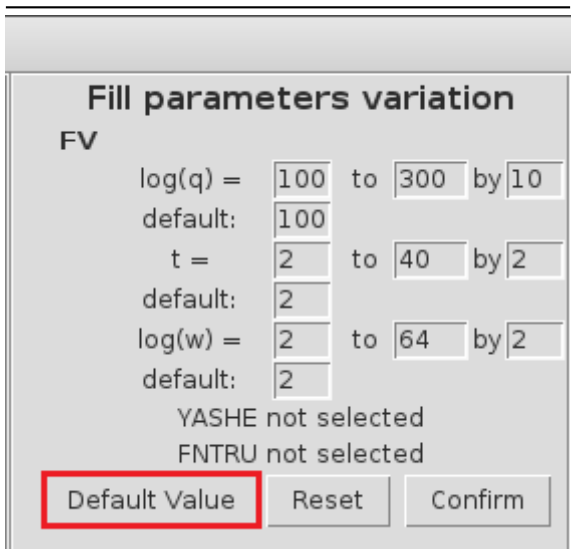
Parameters values can be globally reset to 0 using the *Reset* button:



Reset HE schemes parameter values

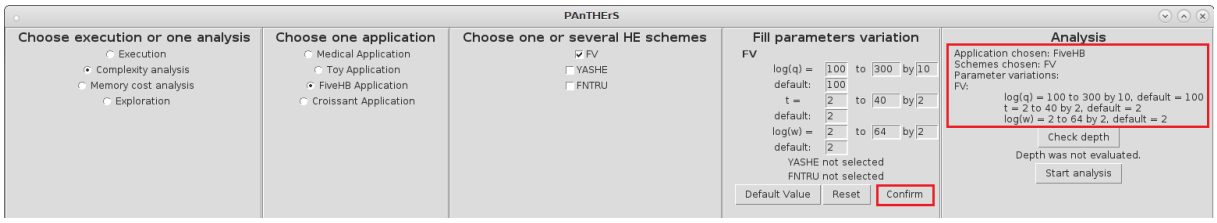
Parameters values can be set to a default value using the *Default Value* button. These default values can be used to quickly test the interface.

> ⚠ Default values are currently only valid for the *FiveHB* Application.



Set default values for HE schemes parameters

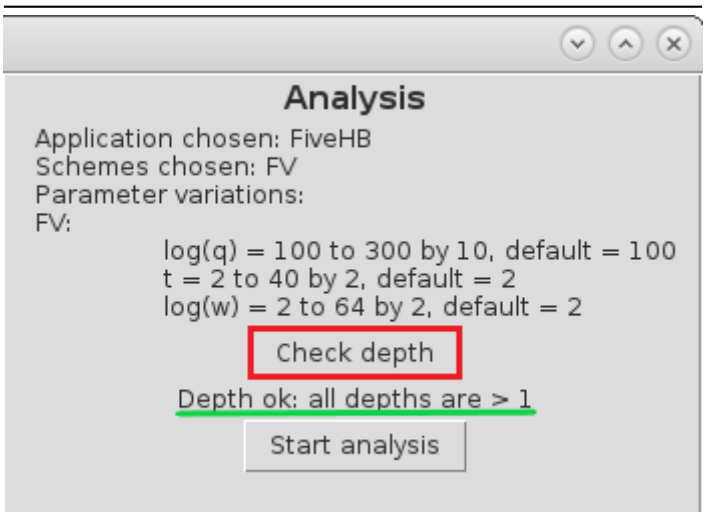
Validate the HE schemes parameters values using the Confirm button. In the last pane of the interface, a brief summary of all analysis configuration is shown:



Validation of HE schemes parameters values

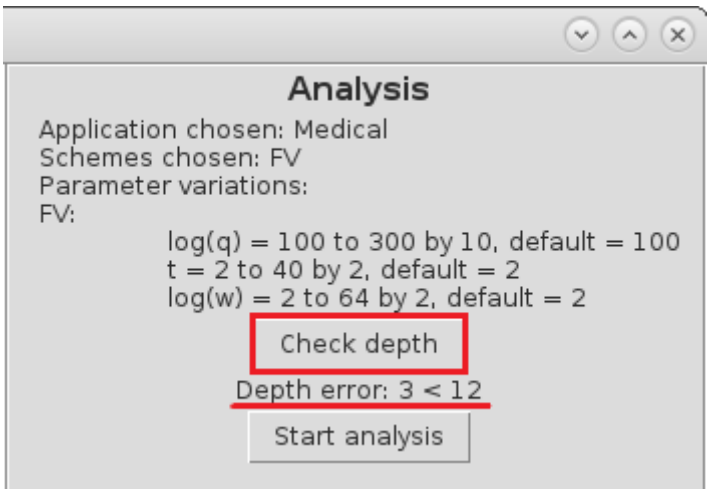
2.4.4 Optional: Check multiplicative depth

Before starting the execution or analysis, you can check that HE schemes parameters values provide sufficient multiplicative depth for the selected application. To check the multiplicative depth click on the *Check depth* button.



Checking HE schemes parameters multiplicative depth

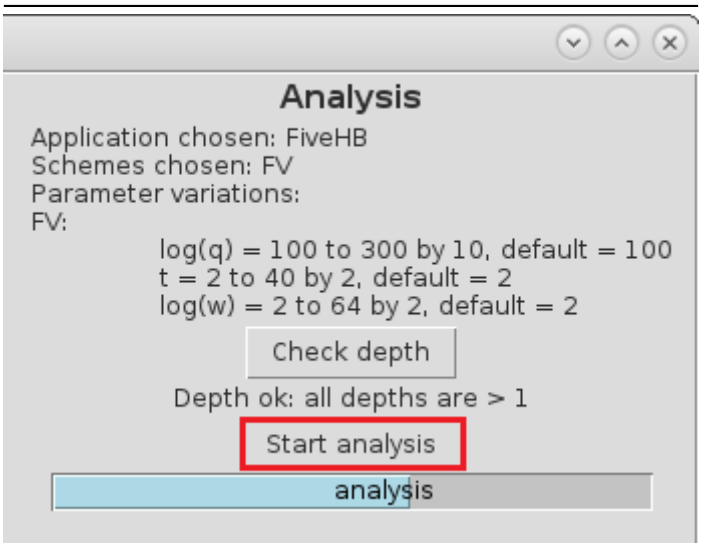
If the multiplicative depth test fails, it means that theoretically at least one set of selected values will not provide enough multiplicative depth to perform homomorphic encryption. In this case you can go back to the value selection phase to select different values.



HE schemes parameters do not provide enough multiplicative depth

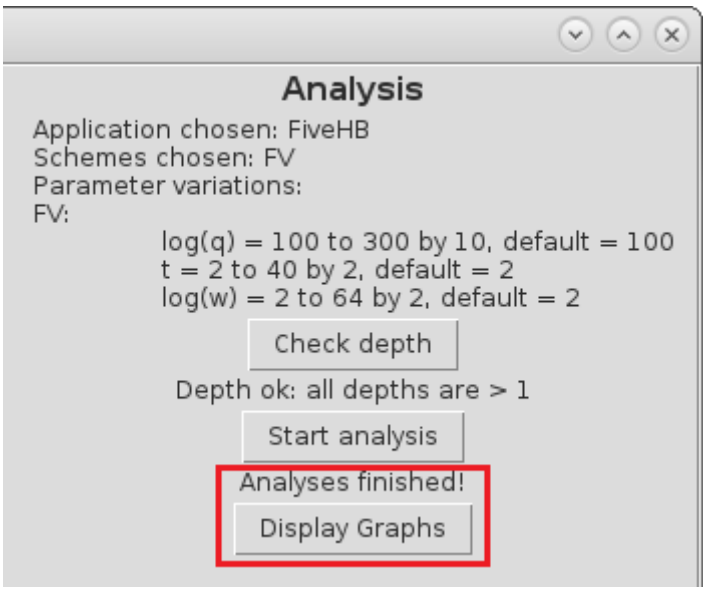
2.4.5 Start execution or analysis

Start the execution or analysis by clicking on the *Start analysis* button. A progress bar shows informations about execution/analysis evolution.



Start analysis or execution

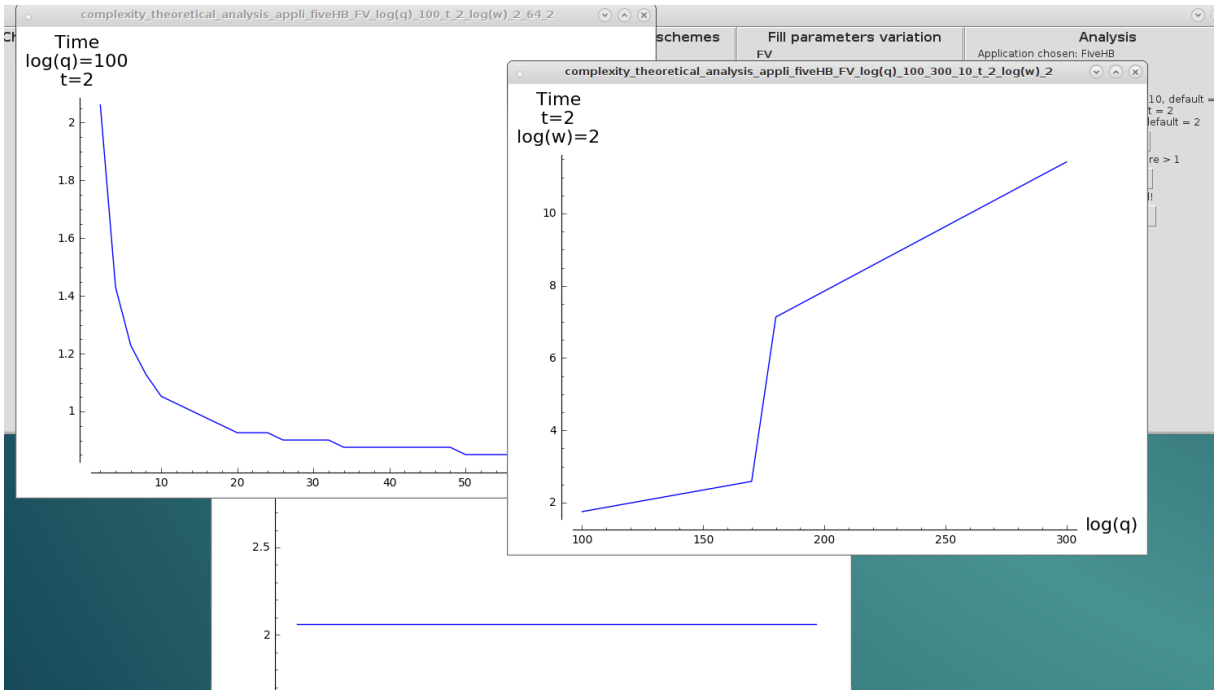
Once execution/analysis is finished, a success message is prompted in the last interface pane, and a *Display Graphs* button appears.



Success message and Display Graphs button, shown at the end of analysis/execution process

2.4.6 Execution or analysis results

Results can be viewed as graphs by clicking on the *Display Graphs* button.



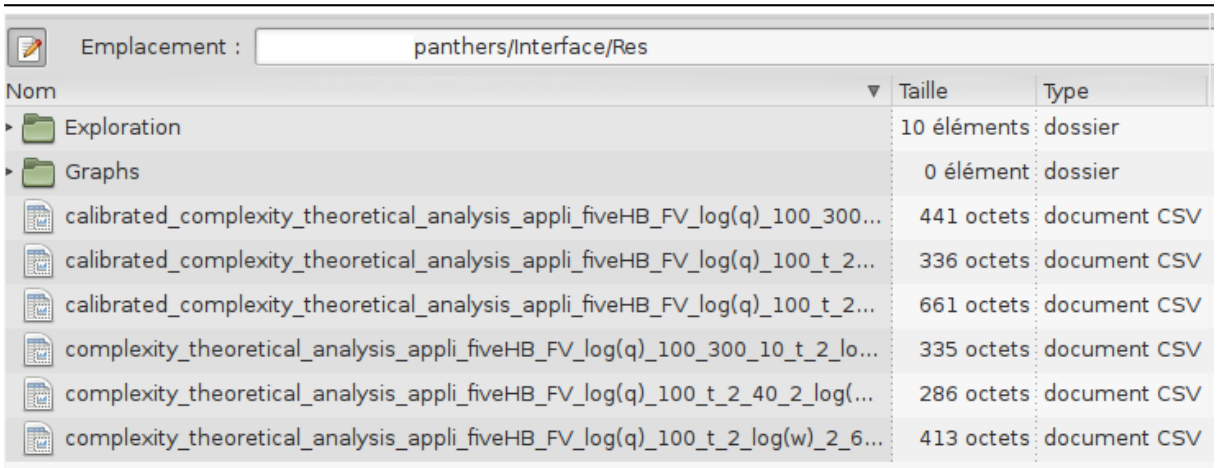
Results shown after clicking on Display Graphs button

All result values are stored in CSV files in following directory:

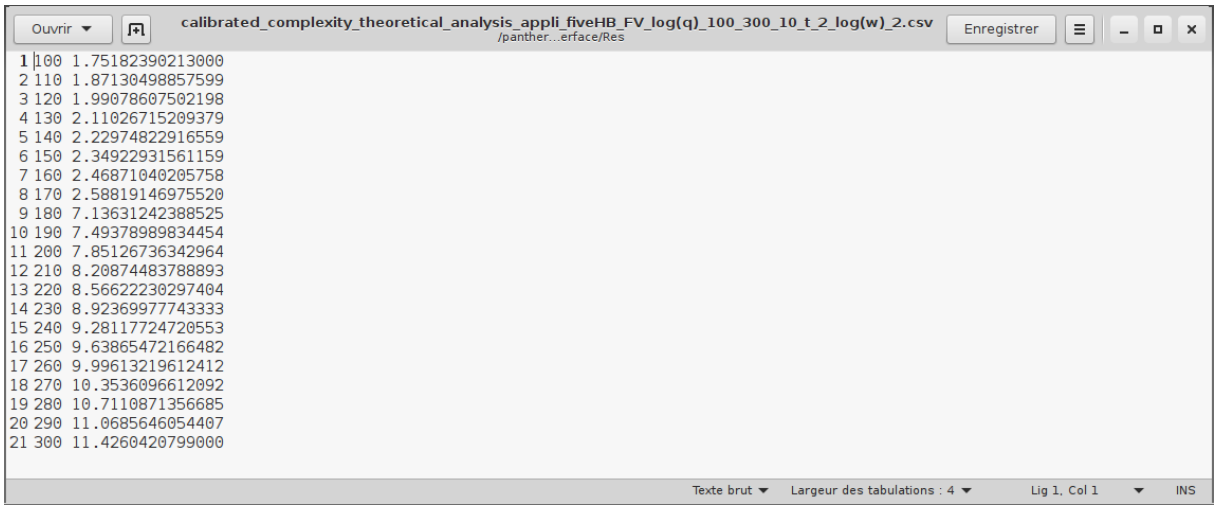
```
/path/to/panthers/Interface/Res
```

Graphs obtained from these results can be found in following directory:

```
/path/to/panthers/Interface/Res/Graphs
```



CSV files containing execution/analysis results




```
100 1.75182390213000
110 1.87130498857599
120 1.99078607502198
130 2.11026715209379
140 2.22974822916559
150 2.34922931561159
160 2.46871040205758
170 2.58819146975520
180 7.13631242388525
190 7.49378989834454
200 7.85126736342964
210 8.20874483788893
220 8.56622230297404
230 8.92369977743333
240 9.28117724720553
250 9.63865472166482
260 9.99613219612412
270 10.3536096612092
280 10.7110871356685
290 11.0685646054407
300 11.4260420799000
```

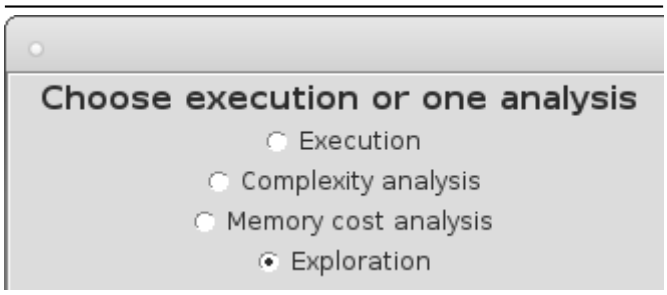
Results contained in a CSV file

2.5 Application Exploration

This part shows how to use PAnTHErS Interface in order to explore all parameters of one or several HE schemes to find optimal parameter sets for a given application.

2.5.1 Exploration selection

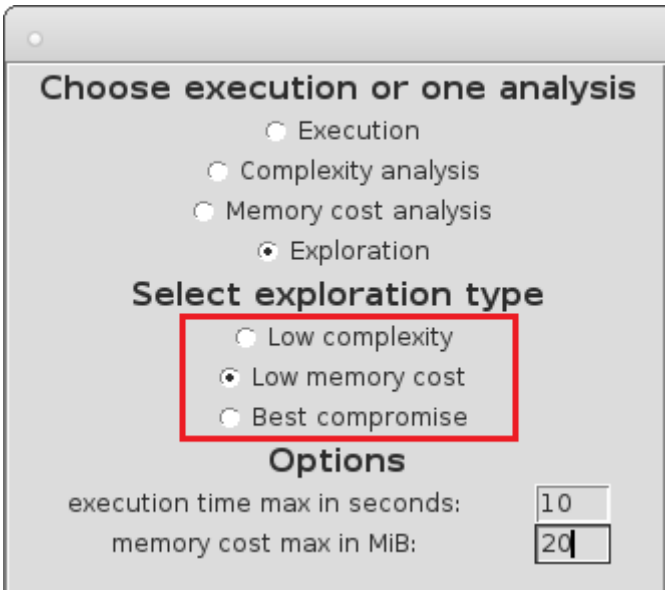
Select the exploration mode from the first pane of the interface.



Selection of the exploration mode

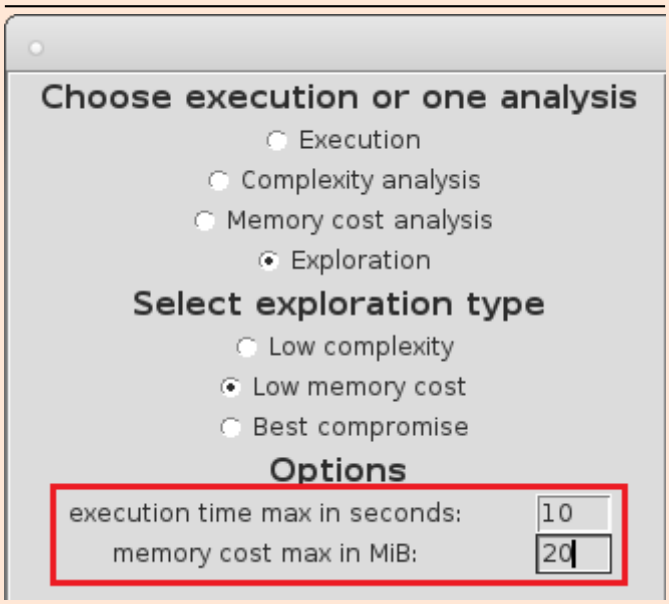
Select the kind of exploration you want to perform. You can choose from:

- **Low complexity**: Exploration will select results inducing lowest execution time.
- **Low memory cost**: Exploration will select results inducing lowest memory consumption.
- **Best compromise**: Exploration will select results based on a tradeoff of the two criteria above.



Selection of the exploration type

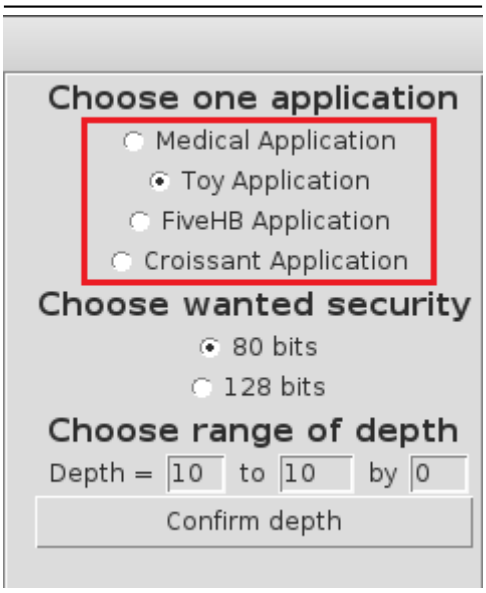
Optional: You can set constraints on the results selected by the exploration by providing maximum values allowed for execution duration and/or memory cost.



Selection of optional constraints

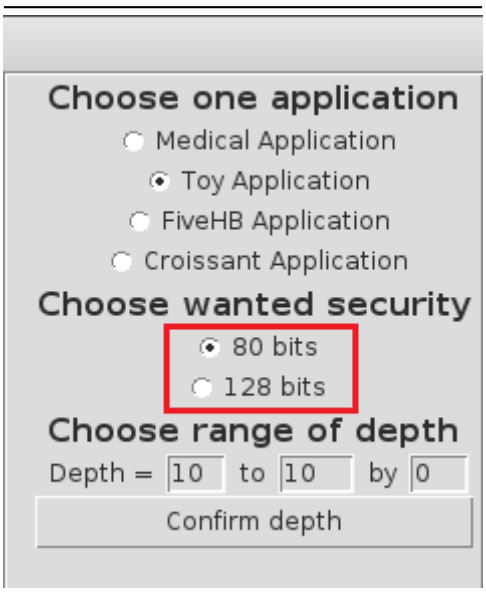
2.5.2 Application selection

Select the application to explore from the second interface pane:



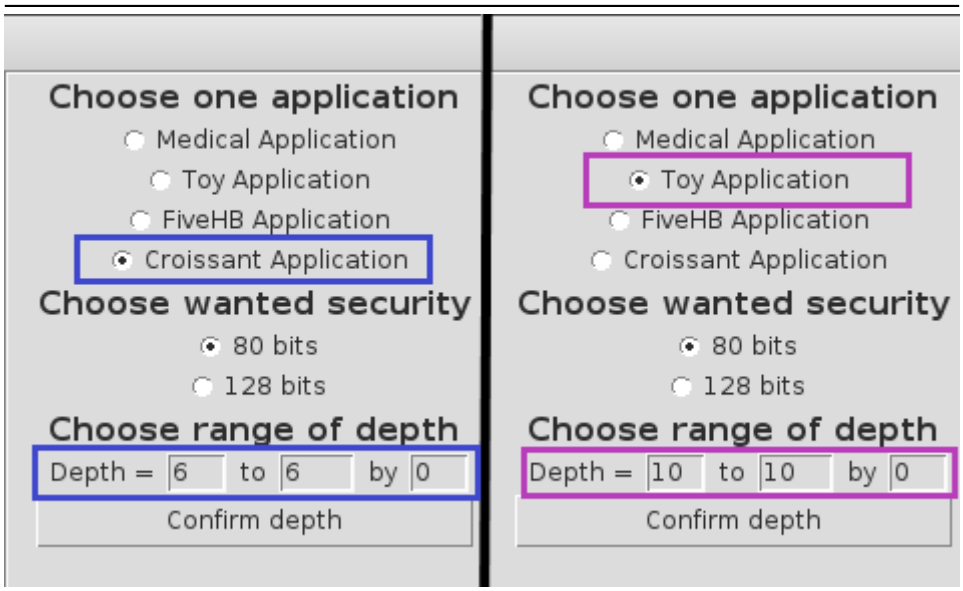
Application selection

After application selection, you have to select the level of security for the HE schemes (80 or 128 bits).



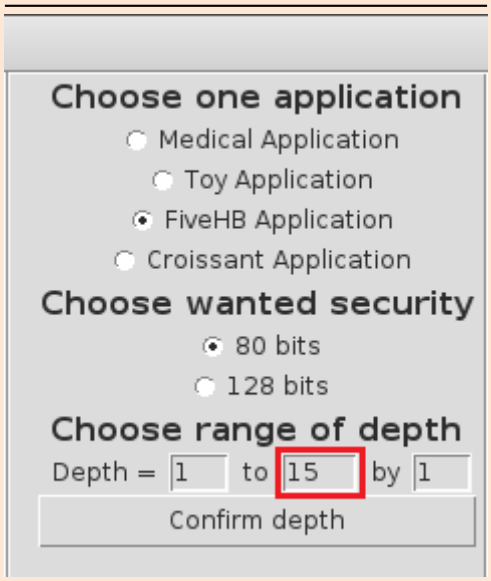
Level of security selection

The multiplicative depth is automatically updated to fit the application selected.



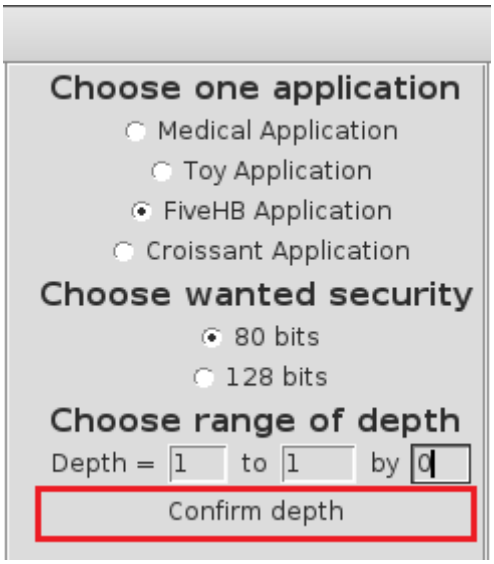
Multiplicative depth is automatically updated when an application is selected

Optional: You can choose a larger range for the multiplicative depth, but the lower depth bound must not be chosen under automatically selected value.



Choosing custom multiplicative depth

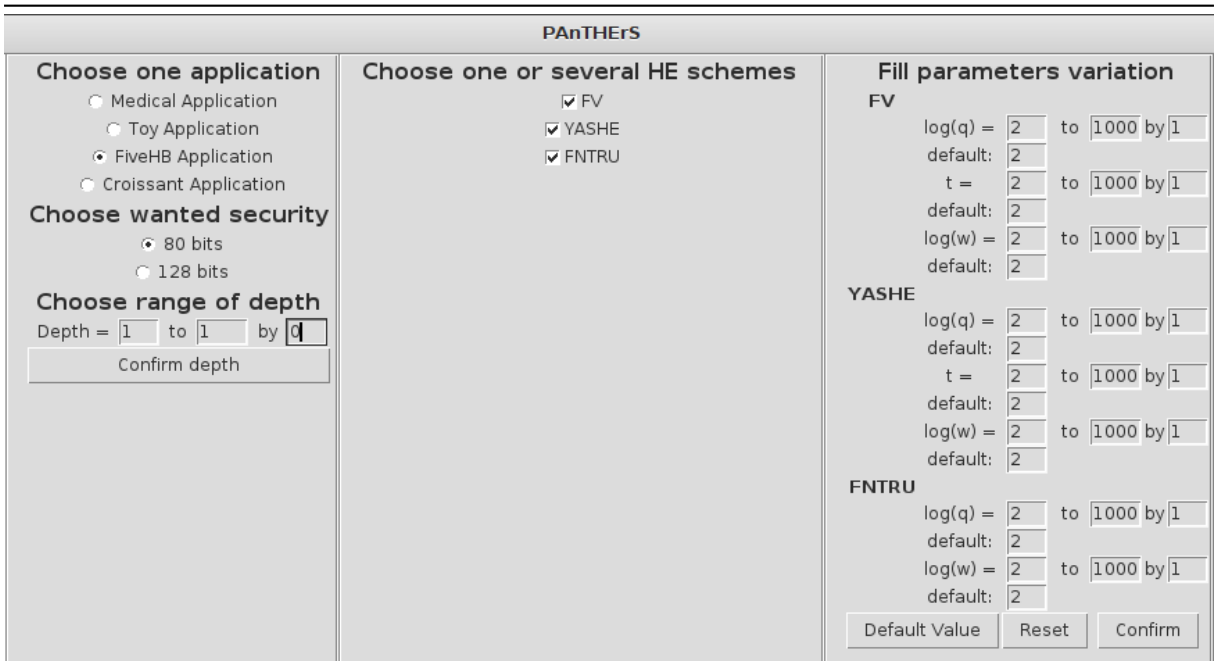
Finalize application selection by clicking on the *Confirm depth* button.



Validation of multiplicative depth by clicking on Confirm depth button

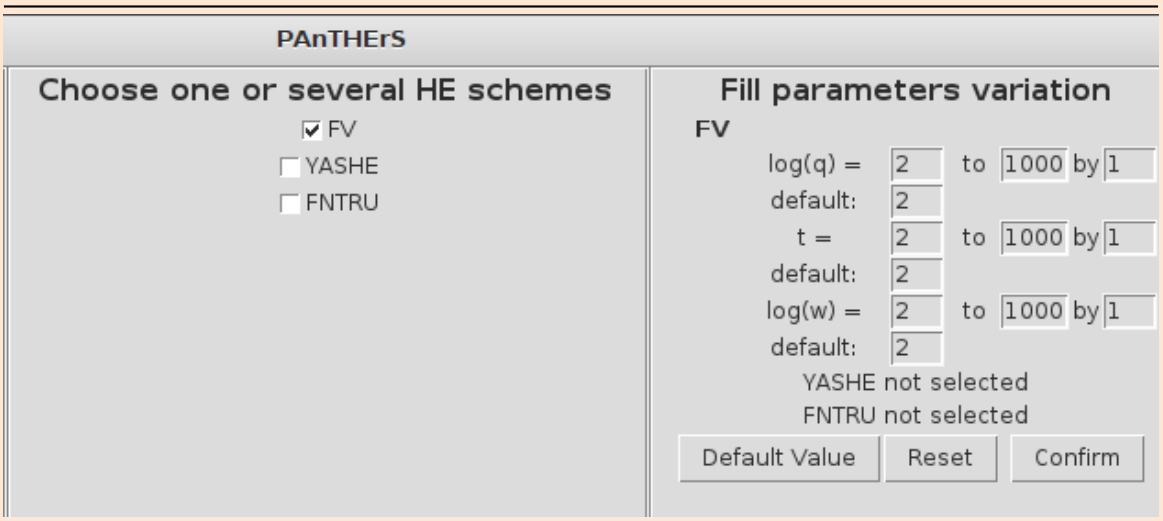
2.5.3 HE schemes and parameters variation selection

After confirmation of mutiplicative depth, all available HE schemes are selected for current exploration.



After multiplicative depth confirmation, all HE schemes are selected

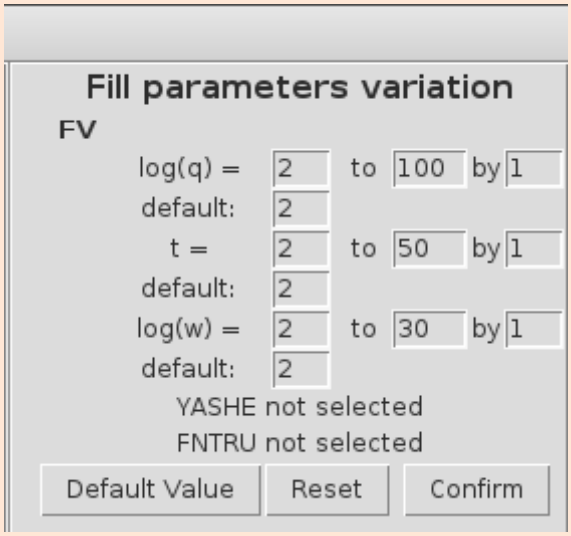
Optional: You can deselect unwanted schemes by unticking them from the HE schemes pane



Deselection of unwanted HE schemes

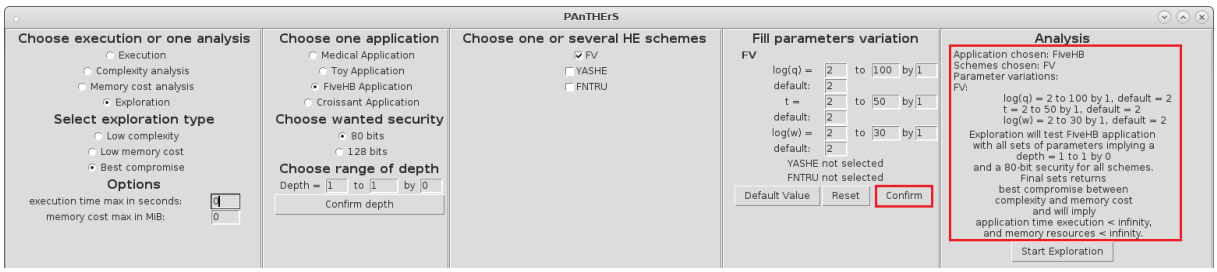
The parameter values for selected HE schemes are automatically filled with chosen values.

Optional: You can change the size of the exploration space by selecting parameters variation as detailed for Analysis/Execution mode.



Selection of custom parameter values or range of values for HE schemes

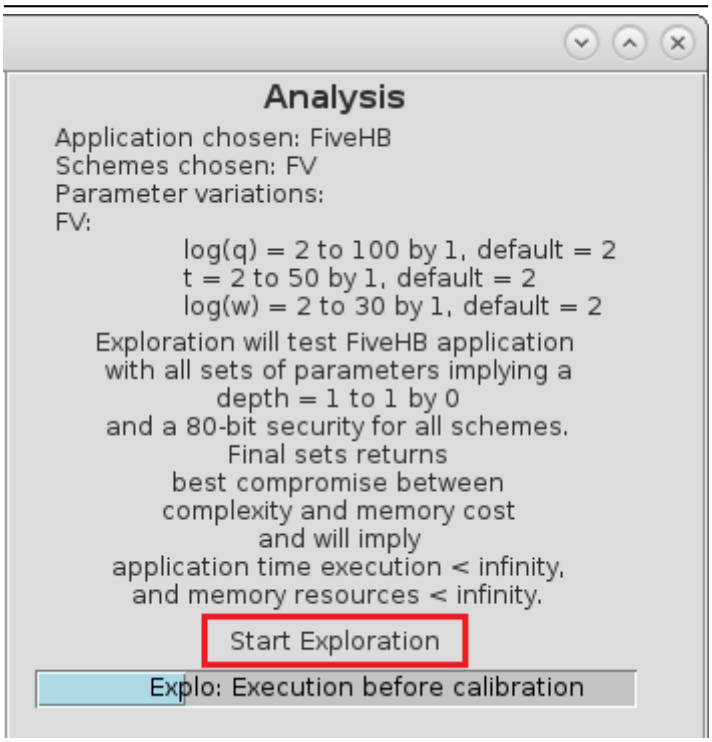
Validate the HE schemes parameters values using the Confirm button. In the last pane of the interface, a brief summary of exploration configuration is shown:



Validation of HE schemes parameters values

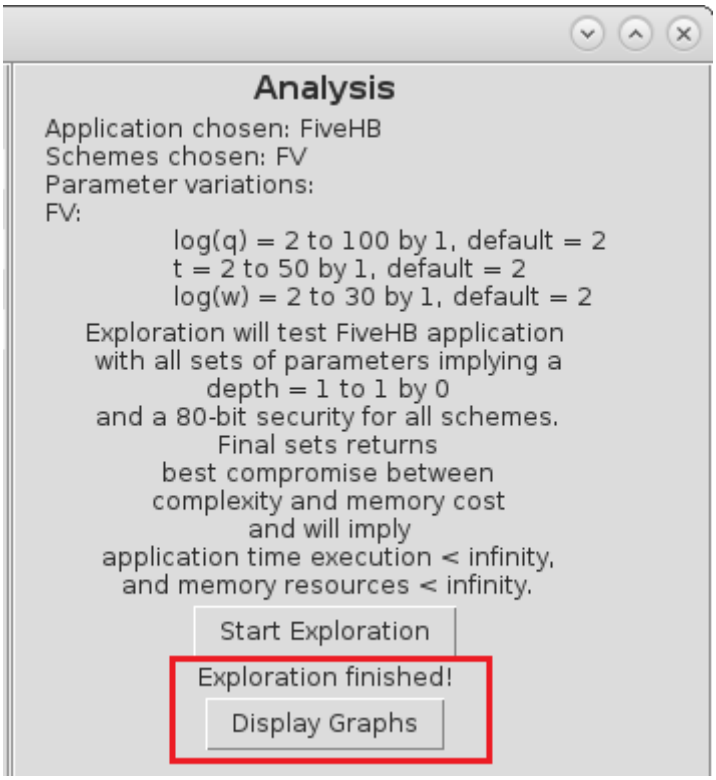
2.5.4 Start exploration

Start the exploration by clicking on the *Start Exploration* button. A progress bar shows informations about exploration evolution.



Start exploration

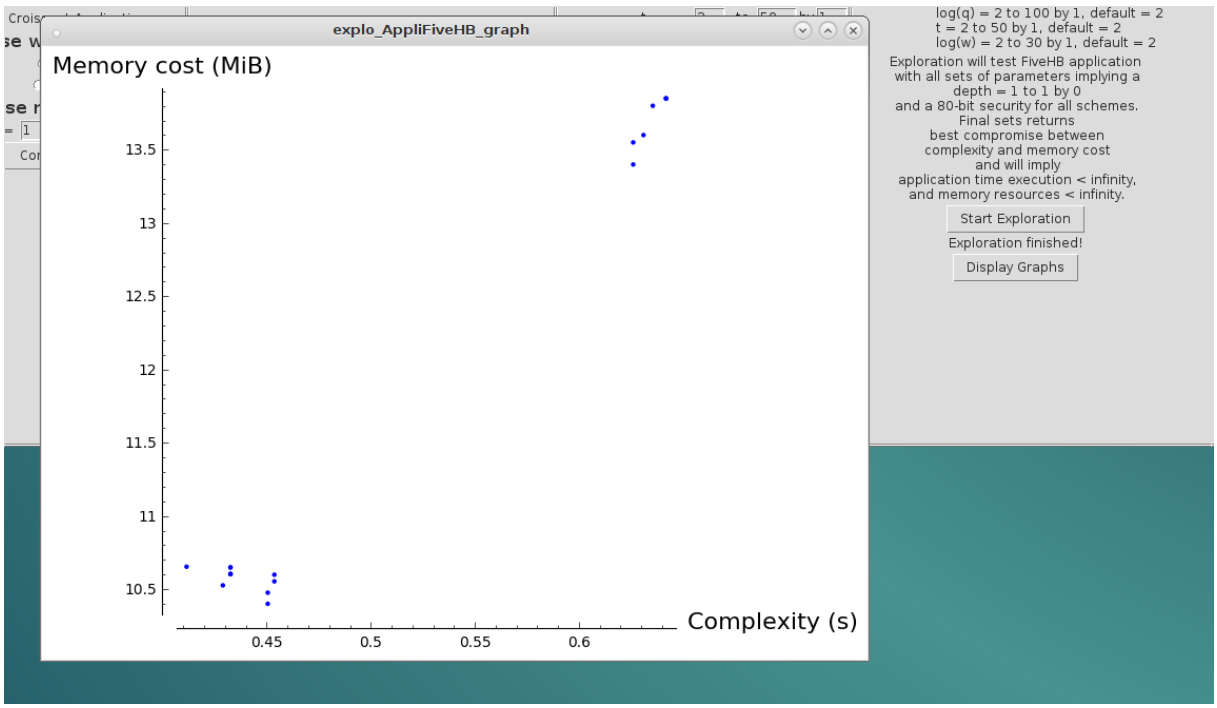
Once exploration is finished, a success message is prompted in the last interface pane, and a *Display Graphs* button appears.



Success message and Display Graphs button, shown at the end of the exploration process

2.5.5 Exploration results

Results can be viewed as graphs by clicking on the *Display Graphs* button.



Results of exploration shown after clicking on Display Graphs button

All explorations result values are stored in CSV files in following directory:

```
/path/to/panthers/Interface/Res/Exploration
```

Graphs obtained from these results can be found in following directory:

```
/path/to/panthers/Interface/Res/Graphs
```

Emplacement : /panthers/Interface/Res/Exploration

Nom	Taille	Type
explo_all_parameters_calibrated_AppliFiveHB_Scheme_FV.csv	1,1 ko	document CSV
explo_All_parameters_theoretical_complexity_AppliFiveHB_Scheme_0.csv	401 octets	document CSV
explo_All_parameters_theoretical_memory_AppliFiveHB_Scheme_0.csv	343 octets	document CSV
explo_optimized_points_AppliFiveHB.csv	313 octets	document CSV
explo_parameters_completed_Pareto_sorted_AppliFiveHB.csv	1,1 ko	document CSV
explo_parameters_executed_complexity_AppliFiveHB_Scheme_0.csv	194 octets	document CSV
explo_parameters_executed_memory_AppliFiveHB_Scheme_0.csv	135 octets	document CSV
explo_parameters_theoretical_complexity_AppliFiveHB_Scheme_0.csv	321 octets	document CSV
explo_parameters_theoretical_memory_AppliFiveHB_Scheme_0.csv	275 octets	document CSV
explo_pre_calibration_AppliFiveHB_Scheme_FV.csv	1,1 ko	document CSV

CSV files containing exploration results

Ouvrir — explo_parameters_completed_Pareto_sorted_AppliFiveHB.csv — /panthers/Interface/Res/Exploration — Enregistrer

```
If scheme == 0 or 1, Parameters = q, t, w
If scheme == 2, Parameters = q, w
If scheme == 3, Parameters = q
Scheme  Depth   Parameters  Complexity  Memory  Time    MiB
0   1   43  2   2   14495952    20071   0.450498819351  10.4
0   1   44  3   2   14495952    20693   0.428920984268  10.5258944913
0   1   46  6   2   15071718    21327   0.411475920188  10.653253185
0   1   44  2   2   14495952    20693   0.450498819351  10.4758944913
0   1   45  3   2   15071718    21699   0.432574407322  10.6486434595
0   1   46  3   2   15071718    21327   0.432574407322  10.603253185
0   1   45  2   2   15071718    21699   0.453672894457  10.5986434595
0   1   45  4   2   15071718    21699   0.432574407322  10.6486434595
0   1   46  2   2   15071718    21327   0.453672894457  10.553253185
0   1   46  4   2   15071718    21327   0.432574407322  10.603253185
0   1   46  5   2   15071718    21327   0.432574407322  10.603253185
0   1   47  2   2   46355505    44652   0.626133737132  13.3992966086
0   1   48  2   2   46355505    45888   0.626133737132  13.5501094563
0   1   48  3   2   46355505    45888   0.631080238954  13.6001094563
0   1   49  2   2   48064840    47936   0.635556936264  13.8
0   1   49  3   2   48064840    47936   0.641926527023  13.85
0   1   49  4   2   48064840    47936   0.641926527023  13.85
0   1   49  5   2   48064840    47936   0.641926527023  13.85
```

Texte brut — Largeur des tabulations : 4 — Lig 1, Col 1 — INS

Results contained in a CSV file

Chapter 3

Adding a new Homomorphic Encryption scheme to PAnTHErS

This chapter explains how to integrate a new HE scheme into PAnTHErS.

3.1 Prerequisites

Before the integration, the new HE scheme algorithm must be decomposed into HE Basic, Specific and Atomic functions.

3.2 Adding a new Specific to PAnTHErS

3.2.1 Get the Specific algorithm template

A template to build a new Specific is available in PAnTHErS source files.

Copy the template file and name it upon your own Specific:

```
$> cd /path/to/panthers/
$> cp Templates/template_Specific.sage mySpecific.sage
```

Now edit the file `mySpecific.sage` and adapt it to your own Specific algorithm.

3.2.2 `make_SpecificName` function

```
    #Rename the function to match your Specific algorithm name
    #Here the new specific created is named 'MySpecific'
    def make_MySpecific(self) :
        """ Build a SpecificFunction object having :
            - a list of input Parameters
            - a function containing its operations (ope function),
            - a list of output Parameters
            """

        #Build a SpecificFunction object named "MySpecific"
        spec = self.builder.specificFunction("MySpecific")
```

```
        #Create initial inputs Parameters
        #Here the input parameters are named q1 and q2
        #If an input Parameter can have different type, put "NoType"

        #Build q1 Parameter,
        #its name is "q1"
        #its type is "int"
        #its initial value is 0
        #its dimension is 1,1 (row count, column count)
        #its degree is 0
        q1 = self.builder.parameter("q1","int",0,1,1,0)

        #Build q2 Parameter,
        #its name is "q2"
        #its type is "NoType" (the type can vary depending on the inputs)
        #its initial value is 0
        #its dimension is 1,1 (row count, column count)
        #its degree is 0
        q2 = self.builder.parameter("q2","NoType",0,1,1,0)

        #List here the newly created parameters q1 and q2
        inputs = [q1, q2]
        spec.inputs = inputs

        #Retrieve Atomics that you will use in your Specific
        prodScal = self.finder.atomic(self.allAtomics, "prodScal")
```

3.2.3 check_outputs function

The check_outputs function checks whether the provided outputs are simple strings or actual parameters.

If a string is provided, it is replaced in the `outputs` list by new Parameter object named upon that string value.

```
    def check_outputs(outputs, count) :
        #If a temporary variable is required, you can use the following name generator
        #to avoid name collisions:
        #"OutSpecName_spec_" + spec.count.str()

        if len(outputs) == 0 :

            #No output name or Parameter provided
            #Create outputs Parameters from scratch
            #Here the output parameters are o1 and o2

            #Their names are generated using predefined prefixs concatenated with a
            #counter incremented each time this Specific function is called

            #Build o1 Parameter,
            #its name is "OutSpecMySpecific_spec_O1_<Specific_call_counter>"
            #its type is "int"
            #its initial value is 0
            #its dimension is 1,1 (row count, column count)
```

```
            #its degree is 0
            o1 = self.builder.parameter("OutSpecMySpecific_spec_O1_" + \
                 spec.count.str(),"int",0,1,1,0)

            #Build o2 Parameter,
            #its name is "OutSpecMySpecific_spec_O2_<Specific_call_counter>"
            #its type is "int"
            #its initial value is 0
            #its dimension is 1,1 (row count, column count)
            #its degree is 0
            o2 = self.builder.parameter("OutSpecMySpecific_spec_O2_" + \
                 spec.count.str(),"int",0,1,1,0)

            #List here newly created output parameters
            outputs = [o1, o2]
            spec.count = spec.count + 1
            spec.outputs = outputs
            return outputs
        else :
            #Replace strings by Parameter objects
            for i in range(len(outputs)):
            if isinstance(outputs[i], str) :

                #Build generic Parameter with default values,
                #its name is the value provided in outputs[i]
                #its type is "NoType" (generic type)
                #its initial value is 0
                #its dimension is 1,1 (row count, column count)
                #its degree is 0
                outputs[i] = self.builder.parameter(outputs[i],"NoType", 0, 1, 1, 0)
        return outputs
```

3.2.4 ope function

The ope function contains the list of operations performed by the new Specific algorithm.

```
    def ope(inputs, outputs = [], count = spec.count) :
        outputs = check_outputs(outputs,count)
        spec.inputs = inputs

        #Report your input parameters here
        #The input parameters for the new Specific are q1 and q2
        [q1, q2] = spec.inputs

        #Describe the specific algorithm here using Atomics and/or Specifics

        #Here the two outputs o1 and o2 are computed using Atomics

        #First output parameter o1 is the result of q1+q2
        #The Atomic add is used
        [outputs[0]] = add.ope([q1,q2], [outputs[0]])

        #Second output parameter o2 is the result of q1*q2
```

```
        #The Atomic mult is used
        [outputs[1]] = mult.ope([q1,q2], [outputs[0]])

        spec.outputs = outputs
        return spec.outputs

    spec.ope = ope
    return spec
```

3.2.5 Integrate your Specific in PAnTHErS library

Edit the `specificfunctioncreator.sage` file.

Register your Specific algorithm in the `SpecificFunctionCreator` class by adding the following line to the `make_all` function:

```
res = res + [self.make_MySpecific()]
```

Copy the code of your function `make_MySpecific`, and paste it after every other `make_function` functions.

```
class SpecificFunctionCreator(object):

    def __init__(self, flag = "HEBasic", file = "") :
    #...
    #...
    #...

    def make_all(self) :
        res = []
        res = res + [self.make_addTimes()]
        res = res + [self.make_distriLWE()]
        res = res + [self.make_doubleDistriLWE()]
        #...
        #...

        #Register your Specific to the SpecificFunctionCreator
        #Add following line:
        res = res + [self.make_MySpecific()]

        return res

    #...
    #...

    def make_addTimes(self) :
    #...
    def make_distriLWE(self) :
            #...
    #...
    def make_msbToPolynomial(self) :
        #...
    def make_msbToPolynomial(self) :
          #...
```

```
    #Here paste the code from the mySpecific.sage file
    def make_MySpecific(self) :
    #... content of mySpecific.sage file
```

3.2.6 Use your Specific in a HE scheme

To have access to your Specific in a HE scheme, you need to declare it by adding the following line in the `__init__` function of the HE scheme class:

```
self.mySpecific = self.finder.specific(self.heMult.allSpecifics, "mySpecific")
```

Here is an example for the integration of the Specific algorithm `mySpecific` in a scheme `SchemeName` class:

```
class SchemeName(HEScheme) :
    """ Describe your HE scheme here """

    def __init__(self, listOfParams, listOfSets) :
        self.builder = Builder()
        self.finder = Finder()
        #...
        #...

        #Definition of Atomics and Specifics available in PAnTHErS library
        self.add = self.finder.atomic(self.heKeyGen.allAtomics, "add")
        self.mult = self.finder.atomic(self.heKeyGen.allAtomics, "mult")
        self.sub = self.finder.atomic(self.heKeyGen.allAtomics, "sub")
        #...
        #...

        self.addTimes = self.finder.specific(self.heKeyGen.allSpecifics, "addTimes")
        self.distriLWE = self.finder.specific(self.heKeyGen.allSpecifics, "distriLWE")
        self.powersOf = self.finder.specific(self.heKeyGen.allSpecifics, "powersOf")
        #...
        #...

        #Declare here the new Specific algorithm to use it in SchemeName
        self.mySpecific = self.finder.specific(self.heMult.allSpecifics, "mySpecific")
        #...
        #...
```

You can now use and test your new Specific algorithm `MySpecific` with an existing PAnTHErS application. However, no theoretical analysis is associated with your algorithm yet. For now, you can only use it for practical executions.

3.3 Create the analysis models for a new Specific

To be able to get theoretical analysis for memory cost and execution time estimations, the Specific algorithm have to be adapted. The algorithm duplication is required and must be integrated with

some adaptations to perform the desired analysis process. This has to be done twice: once for memory consumption analysis and once for computational complexity analysis.

3.3.1 Specific: Memory consumption analysis

Here are described the steps required for providing memory consumption analysis for the Specific algorithm `MySpecific`.

1. Import the Specific algorithm code

The Specific algorithm code from your file `mySpecific.sage` must be imported in the file `specificfunctionmemorycreator.sage`. This file can be found on PAnTHErS source code:

```
/path/to/panthers/Memory/specificfunctionmemorycreator.sage
```

Now edit the file `specificfunctionmemorycreator.sage` to add your own code. Like for `specificfunctioncreator.sage` in section 3.2.5, you need to register your algorithm to the `SpecificFunctionMemory Creator` class by adding the following line to the `make_all` function:

```
res = res + [self.make_MySpecific()]
```

Then, copy the code of your function `make_MySpecific`, and paste it after every other `make_function` functions.

```
class SpecificFunctionMemoryCreator(object):

    def __init__(self) :
        #...
        #...

    def make_all(self) :
        res = []
        res = res + [self.make_addTimes()]
        res = res + [self.make_distriLWE()]
        res = res + [self.make_doubleDistriLWE()]
        #...
        #...

        #Register your Specific to the SpecificFunctionMemoryCreator
        #Add following line:
        res = res + [self.make_MySpecific()]

        return res

    def make_addTimes(self) :
    #...
    def make_distriLWE(self) :
           #...
    #...
    def make_msbToPolynomial(self) :
        #...
    def make_msbToPolynomial(self) :
          #...
```

```
    #Here paste the code from the mySpecific.sage file
    def make_MySpecific(self) :
    #... content of mySpecific.sage file
```

2. Update your Specific algorithm code to evaluate memory consumption

Now from the file `specificfunctionmemorycreator.sage`, edit the code of your `make_MySpecific` function in order to adapt it for memory analysis.

2.1 `make_MySpecific` function

```
    def make_MySpecific(self) :

        #Build a SpecificFunctionMemory object named "MySpecific"
        spec = self.builder.specificFunctionMemory("MySpecific")

        #Create a Memory Object used for memory consumption tracking
        spec.memory = Memory()

        #No more changes required for this function
        #...
```

2.2 `ope` function

The `ope` function also need to be updated.

The prototype of the function must be updated to accept a Memory object:

```
def ope(inputs, outputs = [], memory = spec.memory, count = spec.count) :
```

You also need to update all the operations performed by Atomics and Specifics. Replace all the calls by their memory analysis counterparts. For example:

```
[outputs[0]] = add.ope([q1,q2], [outputs[0]])
#Becomes
[outputs[0]] = add.ope([q1,q2], [outputs[0]], memory)
```

> ⚠ Ensure that your updated function call matches the parameter order as described in the file `Memory/atomicfunctionmemorycreator.sage`.

At the end of the `ope` function, you need to add the two following lines in oder to keep the Memory object sorted:

```
spec.memory.allMem = []
memory.sortSpec(spec, spec.name + " : " )
```

The final `ope` function:

```
    #Update the function prototype to accept a Memory object
    def ope(inputs, outputs = [], memory = spec.memory, count = spec.count) :
```

```
        #...

        #Update the add call for its memory analysis counterpart
        [outputs[0]] = add.ope([q1,q2], [outputs[0]], memory)

        #Update the mult call for its memory analysis counterpart
        [outputs[1]] = mult.ope([q1,q2], [outputs[0]], memory)

        #Add two lines sort parameters in Memory object
        spec.memory.allMem = []
        memory.sortSpec(spec, spec.name + " : " )

        spec.outputs = outputs
        return spec.outputs
```

⚠ **`ope` function: local variables** Another code update is required if you introduced local variables in your `ope` function. As they are not automatically tracked by the Memory object, you need to do it manually.

```
#Local variable example
tmpAdd = self.builder.parameter("TmpVar_add_" + spec.count.str(), "poly", R(0),1,1,0)
tmpMul = self.builder.parameter("TmpVar_mul_" + spec.count.str(), "poly", R(0),1,1,0)

#Register local variables after their creation
memory.add(tmpAdd)
memory.add(tmpMul)
memory.raise_memTmp(tmpAdd)
memory.raise_memTmp(tmpMul)

#...
#function operations
#...

#At the function end, add local variables to global memory
spec.memory.allMem = [tmpAdd, tmpMul]
#...
```

3.3.2 Specific: Computational complexity analysis

Here are described the steps required for providing computational complexity analysis for the Specific algorithm `MySpecific`.

1. Import the Specific algorithm code

The Specific algorithm code from your file `mySpecific.sage` must be imported in the file `specificfunctioncomplexitycreator.sage`. This file can be found on PAnTHErS source code:

```
/path/to/panthers/Complexity/specificfunctioncomplexitycreator.sage
```

Now edit the file `specificfunctioncomplexitycreator.sage` to add your own code. Like for `specificfunctioncreator.sage` in section 3.2.5, you need to register your algorithm to the `SpecificFunctionComplexity Creator` class by adding the following line to the `make_all` function:

```
res = res + [self.make_MySpecific()]
```

Then, copy the code of your function `make_MySpecific`, and paste it after every other `make_function` functions.

```
class SpecificFunctionComplexityCreator(object):

    def __init__(self) :
        #...
        #...

    def make_all(self) :
        res = []
        res = res + [self.make_addTimes()]
        res = res + [self.make_distriLWE()]
        res = res + [self.make_doubleDistriLWE()]
        #...
        #...

        #Register your Specific to the SpecificFunctionComplexityCreator
        #Add following line:
        res = res + [self.make_MySpecific()]

        return res

    def make_addTimes(self) :
    #...
    def make_distriLWE(self) :
           #...
    #...
    def make_msbToPolynomial(self) :
        #...
    def make_msbToPolynomial(self) :
          #...

    #Here paste the code from the mySpecific.sage file
    def make_MySpecific(self) :
    #... content of mySpecific.sage file
```

2. Update your Specific algorithm code to evaluate computational complexity

Now from the file `specificfunctioncomplexitycreator.sage`, edit the code of your `make_MySpecific` function in order to adapt it for computational complexity analysis.

2.1 `make_MySpecific` function

```
    def make_MySpecific(self) :

        #Build a SpecificFunctionComplexity object named "MySpecific"
```

```
        spec = self.builder.specificFunctionComplexity("MySpecific")

        #Create a Complexity Object used for computational complexity tracking
        spec.complexity = Complexity()

        #No more changes required for this function
        #...
```

2.2 `ope` function

The `ope` function also need to be updated.

The prototype of the function must be updated to accept a Complexity object:

```
def ope(inputs, outputs = [], complexity = spec.complexity, count = spec.count) :
```

At the beginning of the `ope` function, add the following line to reinitialize the `spec.complexity` object.

```
spec.complexity.reset()
```

You also need to update all the operations performed by Atomics and Specifics. Replace all the calls by their computational complexity analysis counterparts. For example:

```
[outputs[0]] = add.ope([q1,q2], [outputs[0]])
#Becomes
[outputs[0]] = add.ope([q1,q2], [outputs[0]], complexity)
```

> ⚠ Ensure that your updated function call matches the parameter order as described in the file `Complexity/atomicfunctioncomplexitycreator.sage`.

At the end of the `ope` function, you need to add the following line in oder to log computational complexity in output file:

```
complexity.printInFile(spec, spec.name + " ")
```

The final `ope` function:

```
    #Update the function prototype to accept a Complexity object
    def ope(inputs, outputs = [], complexity = spec.complexity, count = spec.count) :
        spec.complexity.reset()
        #...

        #Update the add call for its computational complexity analysis counterpart
        [outputs[0]] = add.ope([q1,q2], [outputs[0]], complexity)

        #Update the mult call for its computational complexity analysis counterpart
        [outputs[1]] = mult.ope([q1,q2], [outputs[0]], complexity)

        #Log computational complexity in output file
        complexity.printInFile(spec, spec.name + " ")
```

```
        spec.outputs = outputs
        return spec.outputs
```

ope function: loops analysis optimization Analyzing loops can be time consuming. It's possible to optimize their computational complexity analysis process. To do so, you have to retrieve the analysis results for one loop, and then, manually multiply it by the amount of loops that would have been performed.

```
#Loop optimization example
for i in range(10) :
[a] = add.ope([a,b], [a], complexity)

#Becomes

#The factor 10 is provided on the last add operation parameter
[a] = add.ope([a,b], [a], complexity, 10)
```

3.4 Scheme integration using PAnTHErS template

3.4.1 Get the scheme template

A template for a new HE scheme is available in PAnTHErS source files.

Copy the template file and name it upon your own scheme:

```
$> cd /path/to/panthers/
$> cp Templates/template_HE_Scheme_Class.sage myScheme.sage
```

Now edit the file `myScheme.sage` and adapt it to your own scheme.

3.4.2 Class name

```
#Change class name with your own Scheme name
class MyScheme(HEScheme) :
    """ Describe your HE scheme here """
```

3.4.3 `__init__` function

```
    def __init__(self, listOfParams, listOfSets) :
        self.builder = Builder()
        self.finder = Finder()

            #List your input sets here
        #Here the new scheme have a set named R
        [self.R] = listOfSets
```

```
        #List your input parameters here
        #Here the new scheme have two parameters p1 and p2
        [self.p1, self.p2] = self.defineInParams(listOfParams, listOfSets)

        #Put back your input parameters list here
            #Here put back the two parameters p1 and p2
        HEScheme.__init__(self, [self.p1, self.p2], listOfSets)

        # ...
        # ... list of Atomics and Specifics available in PAnTHErS ...
        # ...
```

3.4.4 defineInParams function

```
    def defineInParams(self, listOfParams) :
        """
        Build Parameter objects with input parameters values given in listOfParams
        See Parameter class for a list of all possible build values
        """

        #Build two Parameter objects for the parameters p1 and p2

        #Build p1 Parameter from listOfParams[0] (value of p1),
        #its name is "p1"
        #its type is "int"
        #its dimension is 1,1 (row count, column count)
        p1 = self.builder.parameter("p1", "int", listOfParams[0], 1, 1)

        #Build p2 Parameter from listOfParams[1] (value of p2),
        #its name is "p2"
        #its type is "matrixPoly"
        #its dimension is 2,3 (row count, column count)
        p2 = self.builder.parameter("p2", "matrixPoly", listOfParams[1], 2, 3)

        #Return newly created Parameters
        return [p1, p2]
```

3.4.5 keyGen function

```
    def keyGen(self) :
        """ Defines HEKeyGen object which has :
            - a list of Parameter inputs
            - a function containing operation of key generation.
            Outputs (keys generated) are put in self.inputs of the HEScheme class.
        """

        self.heKeyGen = self.builder.heKeyGen()

        #Here update the type and dimensions of the generated Keys for the scheme

        # Private key is a 'poly' of size 1,1 (row count, column count) with initial
```

```
    # value of 0
    # Public key is a 'listPoly' of size 1,2 (row count, column count) with
    # initial value of [] (empty list)
    # Evaluation key is a 'listPoly' of size 1,2 (row count, column count) with
    # initial value of [] (empty list)
    self.heKeyGen.inputs = self.inputs
    sk = self.builder.key("PrivateKey", "poly",0,1,1)
    pk = self.builder.key("PublicKey", "listPoly",[],1,2)
    rlk = self.builder.key("EvaluationKey", "listPoly",[],1,2)
    self.heKeyGen.outputs = [sk,pk,rlk]
```

3.4.6 keyGen.ope function

The `keyGen.ope` function describes the keys generation for the new HE scheme. The key generation must be described using exclusively Atomic and Specific algorithms. See the definition of ope function of a new Specific algorithm in section 3.2.4 for more examples.

```
def ope(inputs = self.inputs, sets = self.sets, count = self.heKeyGen.count) :
    #Here define all the steps required for keys generation
    #These steps must be described using Atomics and Specifics available
    #in PAnTHErS

    #if a temporary variable is required,
    #you can use the following name generator to avoid name collisions :
    # "TmpHEBasicKeyGen_" + self.heKeyGen.count.str()

    #Private Key generation
    [sk] = addTimes.ope([q1,q2,q2], [sk])
    #Put more operations here
    #...

    #Public Key generation
    [pk] = addTimes.ope([q1,q2,q2], [pk])
    #Put more operations here
    #...

    #Evaluation Key generation
    [rlk] = addTimes.ope([q1,q2,q2], [rlk])
    #Put more operations here
    #...

    #The 3 keys (sk, pk and rlk) are added to the general inputs of the class
    self.inputs = self.inputs + [sk,pk,rlk]
    self.heKeyGen.count = self.heKeyGen.count + 1

self.heKeyGen.ope = ope
```

3.4.7 enc, dec, addHE and multHE functions

Describe the `enc`, `dec`, `addHE` and `multHE` functions using the same approach as for the `keyGen` function.

```
    #The same work has to be done for the following functions
    #using Atomics and Specifics available in PAnTHErS
    def enc(self) :
    #describe the encrypt function
    #...
    def dec(self) :
    #describe the decrypt function
    #...
    def addHE(self) :
    #describe the Homomorphic Addition function
    #...
    def multHE(self) :
    #describe the Homomorphic Multiplication function
    #...
```

3.4.8 depth function

```
    def depth(self) :
        """Optional: calculates multiplicative depth of the scheme (thanks to a
        pre-calculated equation) """
        res = 0

        #Describe depth calculation here (without using Atomics or Specifics)

        return res
```

3.4.9 __repr__ function

```
        #Here return the Name of the class
    def __repr__(self):
        return "MyScheme"
```

3.5 Create the analysis models for a new HE Scheme

To be able to get theoretical analysis for memory cost and execution time estimations, the new HE Scheme have to be adapted. The scheme duplication is required and must be adaptated to perform the desired analysis process. This has to be done twice: once for memory consumption analysis and once for computational complexity analysis.

3.5.1 HE scheme: Memory consumption analysis

Here are described the steps required for providing memory consumption analysis for the HE Scheme `MyScheme`, defined in previous section (seciton 3.4).

1. Duplicate the HE Scheme source file

Your HE scheme source file `myScheme.sage` must be duplicated. Here we duplicate it under the name of `mySchemeMemory.sage`, the duplicated file is placed in PAnTHErS source `Memory` folder.

```
$> cd /path/to/panthers/
$> cp myScheme.sage Memory/mySchemeMemory.sage
```

Now edit the file `mySchemeMemory.sage` to adapt the code for memory analysis.

2. Update your HE scheme code to evaluate memory consumption

Now from the file `mySchemeMemory.sage`, edit the code of your HE scheme in order to adapt it for memory consumption analysis.

2.1 Class name

Update your class name to `MySchemeMemory`, also, make it inherit from `HESchemeMemory`.

```
#Change class name
class MySchemeMemory(HESchemeMemory) :
```

2.2 `__init__` function

At the beginning of the `__init__` function, you must declare a `Memory` object that will be used to track memory consumption.

```
self.memory = Memory(flag, file)
```

You have to change the call to `HEScheme.__init__` to `HESchemeMemory.__init__`.

```
HEScheme.__init__(self, [self.p1, self.p2], listOfSets)
#Becomes
HESchemeMemory.__init__(self, [self.p1, self.p2], listOfSets, self.memory)
```

At last, you must declare the module of your HE scheme to the `Memory` object. The module is taken from the input parameters. Here the parameter `p1` is declared as the module for the HE scheme.

```
self.memory.module = self.p1
```

Here is the resulting `__init__` function.

```
    def __init__(self, listOfParams, listOfSets) :
        self.builder = Builder()
        self.finder = Finder()

        #Add a Memory object to track memory consumption
        self.memory = Memory(flag, file)

            #...

        #Change HEScheme to HESchemeMemory
        HESchemeMemory.__init__(self, [self.p1, self.p2], listOfSets, self.memory)
```

```
        #Define scheme module in Memory object
        #The module is taken from input parameters
        self.memory.module = self.p1

        # ...
```

2.3 keyGen function

You need to change the line responsible of the creation of the `heKeyGen` object. The type of this object changes from `HEKeyGen` to `HEKeyGenMemory`.

```
self.heKeyGen = self.builder.heKeyGen()
#Becomes
self.heKeyGen = self.builder.heKeyGenMemory()
```

Rigth after that line, you need to build a `Memory` object for the new `HEKeyGenMemory` object.

```
self.heKeyGen.memory = Memory(flag, file)
```

Here is the resulting `keyGen` function.

```
    def keyGen(self) :

        #Update the builder to create a HEKeyGenMemory object
        self.heKeyGen = self.builder.heKeyGenMemory()
        #Create the associated Memory object
        self.heKeyGen.memory = Memory(flag, file)

        #...
```

2.4 keyGen.ope function

Retrieve the `Memory` object at the beginning of the `ope` function.

```
memory = self.memory
```

Like for the Specific algorithm memory analysis function (section 3.3.1), the Specific and Atomic algorithm calls used in the `keyGen.ope` function need to be replaced by their memory consumption analysis couterparts. For example:

```
[sk] = addTimes.ope([q1,q2,q2], [sk])
#Becomes
[sk] = addTimes.ope([q1,q2,q2], [sk], memory)
```

> ⚠ Ensure that your updated function call matches the parameter order as described in the files `Memory/atomicfunctionmemorycreator.sage` and `Memory/specificfunctionmemorycreator.sage`.

At the end of the function, add some lines of code to keep track of memory consumption. You first need to report there the ouput parameters (as the `keyGen.ope` function *stores* its outputs in the inputs, the inputs are referenced here).

```
memory.sortHEBasic(self.heKeyGen, "KeyGen : ", self.inputs)
```

Then, add the two following lines to keep memory object updated.

```
self.heKeyGen.memory = memory
self.memory.update(memory)
```

Here is the resulting `keyGen.ope` function.

```
    def ope(inputs = self.inputs, sets = self.sets, count = self.heKeyGen.count) :
        #Retrieve the Memory object
        memory = self.memory


        #Private Key generation
        #Update addTimes with its memory consumption analysis couterpart
        [sk] = addTimes.ope([q1,q2,q2], [sk], memory)
        #...

        #Public Key generation
        #Update addTimes with its memory consumption analysis couterpart
        [pk] = addTimes.ope([q1,q2,q2], [pk], memory)
        #...

        #Evaluation Key generation
        #Update addTimes with its memory consumption analysis couterpart
        [rlk] = addTimes.ope([q1,q2,q2], [rlk], memory)
        #...


        #Add this line to keep memory object sorted
        #For the keyGen.ope only, report the inputs on last parameter
        memory.sortHEBasic(self.heKeyGen, "KeyGen : ", self.inputs)

        #Add the two following lines to keep track of memory consumption
        self.heKeyGen.memory = memory
        self.memory.update(memory)
```

⚠ **`keyGen.ope` function: local variables** Another code update is required if you introduced local variables in your `keyGen.ope` function. As they are not automatically tracked by the Memory object, you need to do it manually.

```
#Local variable example
tmpAdd = self.builder.parameter("TmpVar_add_" + spec.count.str(), "poly", R(0),1,1,0)
tmpMul = self.builder.parameter("TmpVar_mul_" + spec.count.str(), "poly", R(0),1,1,0)

#Register local variables after their creation
memory.add(tmpAdd)
memory.add(tmpMul)
memory.raise_memTmp(tmpAdd)
```

```
memory.raise_memTmp(tmpMul)

#...
#function operations
#...

#At the function end, add local variables to the existing inputs
memory.sortHEBasic(self.heKeyGen, "KeyGen : ", self.inputs + [tmpAdd, tmpMul])
#...
```

2.5 enc, dec, addHE and multHE functions

Describe the `enc`, `dec`, `addHE` and `multHE` functions using the same approach as for the `keyGen` function.

However, be sure to adapt the process for each function. For example, in the **`enc`** function, create a **`heEnc`** object of type **`HEEncMemory`**.

```
    def enc(self) :
        #Update the builder to create a HEEncMemory object
        self.heEnc = self.builder.heEncMemory()
        #Create the associated Memory object
        self.heEnc.memory = Memory(flag, file)
```

And so on for each following functions.

```
    def enc(self) :
    #describe the encrypt function
    #...
    def dec(self) :
    #describe the decrypt function
    #...
    def addHE(self) :
    #describe the Homomorphic Addition function
    #...
    def multHE(self) :
    #describe the Homomorphic Multiplication function
    #...
```

⚠ At the end of each `functionHE.ope` function, take care to report the ouput parameters and every local variable you may have created. For example for the `enc` function:

```
memory.sortHEBasic(self.heEnc, "MyFunction: ", self.outputs + [tmp1, tmp2, \
    tmpN, ...])
```

2.6 __repr__ function

```
    #Here return the Name of the class
    def __repr__(self):
        return "MySchemeMemory"
```

3.5.2 HE scheme: computational complexity analysis

Here are described the steps required for providing computational complexity analysis for the HE Scheme `MyScheme`, defined in section 3.4.

1. Duplicate the HE Scheme source file

Your HE scheme source file `myScheme.sage` must be duplicated. Here we duplicate it under the name of `mySchemeComplexity.sage`, the duplicated file is placed in PAnTHErS source `Complexity` folder.

```
$> cd /path/to/panthers/
$> cp myScheme.sage Complexity/mySchemeComplexity.sage
```

Now edit the file `mySchemeComplexity.sage` to adapt the code for computational complexity analysis.

2. Update your HE scheme code to evaluate computational complexity

Now from the file `mySchemeComplexity.sage`, edit the code of your HE scheme in order to adapt it for computational complexity analysis.

2.1 Class name

Update your class name to `MySchemeComplexity`, also, make it inherit from `HESchemeComplexity`.

```
#Change class name
class MySchemeComplexity(HESchemeComplexity) :
```

2.2 `__init__` function

You must update the `__init__` function prototype to allow a `Complexity` object as input parameter.

```
def __init__(self, listOfParams, listOfSets, flag = "HEBasic", file = "", \
    complexity = Complexity()) :
```

You have to change the call to `HEScheme.__init__` to `HESchemeComplexity.__init__`.

```
HEScheme.__init__(self, [self.p1, self.p2], listOfSets)
#Becomes
HESchemeComplexity.__init__(self, [self.p1, self.p2], listOfSets, complexity)
```

Here is the resulting `__init__` function.

```
    def __init__(self, listOfParams, listOfSets, flag = "HEBasic", file = "", \
        complexity = Complexity()) :
        self.builder = Builder()
        self.finder = Finder()

            #...

        #Change HEScheme to HESchemeComplexity
        HESchemeComplexity.__init__(self, [self.p1, self.p2], listOfSets, complexity)

        # ...
```

2.3 keyGen function

You need to change the line responsible of the creation of the heKeyGen object. The type of this object changes from HEKeyGen to HEKeyGenComplexity.

```
self.heKeyGen = self.builder.heKeyGen()
#Becomes
self.heKeyGen = self.builder.heKeyGenComplexity()
```

Rigth after that line, you need to build a Complexity object for the new HEKeyGenComplexity object.

```
self.heKeyGen.complexity = Complexity(flag, file)
```

Here is the resulting keyGen function.

```
    def keyGen(self) :

        #Update the builder to create a HEKeyGenComplexity object
        self.heKeyGen = self.builder.heKeyGenComplexity()
        #Create the associated Complexity object
        self.heKeyGen.complexity = Complexity(flag, file)

        #...
```

2.4 keyGen.ope function

You must update the keyGen.ope function prototype to allow a Complexity object as input parameter.

```
def ope(inputs = self.inputs, sets = self.sets, complexity = self.heKeyGen.complexity) :
```

At the beginning of the keyGen.ope function add the following line to reset the complexity object:

```
self.heKeyGen.complexity.reset()
```

Like for the Specific algorithm memory analysis function in section 3.3.1, the Specific and Atomic algorithm calls used in the keyGen.ope function need to be replaced by their computational complexity analysis couterparts. For example:

```
[sk] = addTimes.ope([q1,q2,q2], [sk])
#Becomes
[sk] = addTimes.ope([q1,q2,q2], [sk], complexity)
```

> ⚠ Ensure that your updated function call matches the parameter order as described in the files Complexity/atomicfunctioncomplexitycreator.sage and Complexity/specificfunctioncomplexitycreator.sage.

At the end of the function, add the following line to log computational complexity information in output file.

```
complexity.printInFile(self.heKeyGen, "KeyGen ")
```

Here is the resulting `keyGen.ope` function.

```
    def ope(inputs = self.inputs, sets = self.sets, \
        complexity = self.heKeyGen.complexity) :
        #Reset complexity object
        self.heKeyGen.complexity.reset()


        #Private Key generation
        #Update addTimes with its computational complexity analysis couterpart
        [sk] = addTimes.ope([q1,q2,q2], [sk], complexity)
        #...

        #Public Key generation
        #Update addTimes with its computational complexity analysis couterpart
        [pk] = addTimes.ope([q1,q2,q2], [pk], complexity)
        #...

        #Evaluation Key generation
        #Update addTimes with its computational complexity analysis couterpart
        [rlk] = addTimes.ope([q1,q2,q2], [rlk], complexity)
        #...


        #Add this line to log computational complexity data to output file
        complexity.printInFile(self.heKeyGen, "KeyGen ")
```

2.5 `enc`, `dec`, `addHE` and `multHE` functions

Describe the `enc`, `dec`, `addHE` and `multHE` functions using the same approach as for the `keyGen` function.

However, be sure to adapt the process for each function. For example, in the **`enc`** function, create a **`heEnc`** object of type **`HEEncComplexity`**.

```
    def enc(self) :
        #Update the builder to create a HEEncComplexity object
        self.heEnc = self.builder.heEncComplexity()
        #Create the associated Complexity object
        self.heEnc.complexity = Complexity(flag, file)
```

And so on for each following functions.

```
    def enc(self) :
    #describe the encrypt function
    #...
    def dec(self) :
    #describe the decrypt function
    #...
    def addHE(self) :
```

```
    #describe the Homomorphic Addition function
    #...
    def multHE(self) :
    #describe the Homomorphic Multiplication function
    #...
```

⚠ At the end of each `functionHE.ope` function, take care to report the related function call to log the results. For example for the `enc` function:

```
complexity.printInFile(self.heEnc, "Enc ")
```

2.6 `__repr__` function

```
    #Here return the Name of the class
    def __repr__(self):
        return "MySchemeComplexity"
```

3.6 Adding the new scheme to PAnTHErS graphical interface

To use your new HE scheme on PAnTHErS graphical interface, you need to manually add it in the source files.

3.6.1 In `const_id.py` file

Add a global identifier for your scheme in the file `const_id.py`.

⚠ Beware not to choose an identifier already affected to an existing scheme. It is recommended to use the value of the highest known scheme ID and increment it by one for your own scheme ID.

For example, for the application `MyScheme`, create a new global identifier called `MYSCHEME_ID`:

```
MYSCHEME_ID = 3
```

In the `getSchemeName` function, add the code required to find the name of the new scheme from its global identifier:

```
        elif scheme == MYSCHEME_ID :
                schemeName = "myScheme"
```

3.6.2 In `Interface/interface.py` file

In the file `Interface/interface.py`, add to the `__init__` function:

```
# "MyScheme" = scheme name
# MYSCHEME_ID = global identifier for MyScheme
#2 = nbOfParams (for the two params p1 and p2)
```

```
#["p1", "p2"] = varying input parameters names of current scheme
self.myScheme = SchemeInterface(self.schemeFrame, self.paramFrame, "MyScheme", \
    MYSCHEME_ID, 2, ["p1", "p2"], self.nbOfRow)
self.updateNbOfRow(self.myScheme)
```

Also add your scheme to the list `schemeList`:

```
#Add self.myScheme in self.schemeList
self.schemeList = [self.fv, self.yashe, self.fntru, self.myScheme]
```

In the function `defaultValues`, add the default values for the varying input parameters of your scheme. These values will be used when clicking on the `Default Value` button of the graphical interface.

```
if analyzeOrExplo == ANALYZE_ID:
    #Default values for analysis
    #....
    self.myScheme.params[0].forDefaultButton = [100,300,10,100] #p1
    self.myScheme.params[1].forDefaultButton = [2,40,2,2] #p2
else:
    #Default values for exploration
    #...
    self.myScheme.params[0].forDefaultButton = [2,1000,1,2] #p1
    self.myScheme.params[1].forDefaultButton = [2,1000,1,2] #p2
```

3.6.3 In `Analyse/analyse.py` file

At the beginning of the file `Analyse/analyse.py`, add the lines:

```
load("../myScheme.sage")
load("../Memory/mySchemeMemory.sage")
load("../Complexity/mySchemeComplexity.sage")
```

3.6.4 In every `Analyse/appli_*.sage` files

In every function named `computeApplication*`, add the following condition to test if the given scheme is an instance of `myScheme`, and then define the type of parameter used for ciphertexts :

```
    #Here MyScheme ciphertexts are matrixPoly
    if isinstance(scheme, MyScheme) or isinstance(scheme, MySchemeComplexity) or \
        isinstance(scheme, MySchemeMemory) :
        typeCipher = "matrixPoly"
```

3.6.5 In `Analyse/parameterschoice.sage` file

Create a function named `chooseMySchemeParameter (p1, p2, secu = 80)`. Here we provided two input parameters (p1 and p2).

⚠ Only provide input parameters that can be varied (during an analysis).

The purpose of this function is to generate/create the parameters `listOfParams` and `listOfSets`. Parameters that can be varied are provided as input parameters. Other fixed value parameters required by the HE scheme have to be generated/initialized here.

You can see examples in other `choose*Parameter` functions already available in this file.

In the function named `chooseParameter`, add a condition with your own scheme identifier:

```
elif scheme == MYSCHEME_ID :
    #Here MyScheme takes 2 input parameters
    if len(params) != 2 :
        raise Exception("chooseParameter : there is not 2 parameters (but {}) in \
            params".format(len(params)))

    #Retrieve scheme parameters here
    p1 = params[0].currentValue
    p2 = params[1].currentValue

    #call chooseMySchemeParameter with p1 and p2
    return chooseMySchemeParameter(p1,p2,secu)
```

In the function `createSchemeObject`, add a conditional branch for your scheme. The three lower conditional branches are required to generate an appropriate HE scheme object depending on the kind analysis chosen.

```
elif scheme == MYSCHEME_ID :
    if whichAnalysis == EXECUTION_ID : #execution
        return MyScheme(params, sets, "HEBasic", "")
    elif whichAnalysis == COMPLEXITY_ANALYSIS_ID : #complexity
        return MySchemeComplexity(params, sets, "HEBasic", "")
    elif whichAnalysis == MEMORY_ANALYSIS_ID : #memory cost
        return MySchemeMemory(params, sets, "HEBasic", "")
```

Chapter 4

Adding a new Homomorphic application to PAnTHErS

PAnTHErS is provided with some applications. They show some examples of how it is possible to evaluate homomorphic encryption usage. You can also integrate your own application within PAnTHErS with a few steps described in this chapter.

4.1 Write your Application code and convert it to PAnTHErS format.

You first need to get your own application working, written in C++, using Cingulata tool.

Then convert your application to BLIF format. A detailed example of application convertion to BLIF is presented on Cingulata website.

From your converted BLIF file, convert it using PAnTHErS embedded tool. The convertion tool is located on `/path/to/panthers/Templates/Template_Appli/Tools`.

To convert your application you need:

- `parse_mapped_blif.rb`: The convertion tool, written in Ruby.
- `appli.blif`: Your application, converted to BLIF format using Cingulata.

From a terminal, run PAnTHErS convertion tool. Here is an example of convertion for a sample `appli.blif` file:

```
$> ./parse_mapped_blif.rb -i appli.blif -o appli.sage
Module "CIRCUIT"
 Inputs:      8
 Outputs:     8
 Gates:      94
  buf:        1
  inv:       27
  and:       14
  xor:       53
$>
```

The tool will create a file called `appli.sage`, containing your converted application.

Here is an example of the content for a sample `appli.sage` file:

```
i_2 = ENC(i_2_input);
i_3 = ENC(i_3_input);
i_4 = ENC(i_4_input);
i_5 = ENC(i_5_input);
i_6 = ENC(i_6_input);
i_7 = ENC(i_7_input);
i_8 = ENC(i_8_input);
i_9 = ENC(i_9_input);

m_6 = XOR(i_9, i_8);
n106 = NOT(i_9);
n104 = NOT(i_8);
n20 = AND(i_9, i_8);
n22 = NOT(i_7);
m_7 = i_9;
n40 = NOT(i_5);
n19 = NOT(i_3);
...
```

Keep your `appli.sage` file for later PAnTHErS application integration steps.

4.2 Add your application code to PAnTHErS

4.2.1 Get application template files

Get 2 template files from `/path/to/panthers/Templates/Template_Appli`:

- appli_TOREPLACE.py
- exec_TOREPLACE.py

Copy them to `/path/to/panthers/Analyse` and rename it with your app name.

Here is an example for a sample application named `MyApp`.

```
$> cd /path/to/panthers
$> cp Templates/Template_Appli/appli_TOREPLACE.py Analyse/appli_MyApp.py
$> cp Templates/Template_Appli/exec_TOREPLACE.py Analyse/exec_MyApp.py
```

Edit the code from the new created files.

First edit the file `appli_MyApp.py` and `exec_MyApp.py`, replace every instance of the string `TOREPLACE` with the name of your application. You can do it using the following `sed` command:

```
$> cd /path/to/panthers/Analyse
$> sed -i 's/TOREPLACE/MyApp/g' appli_MyApp.py exec_MyApp.py
```

4.2.2 Create a global identifier for your application

Edit the file `const_id.py`, and add a global identifier for your application.

> ⚠ Beware not to choose an identifier already affected to an existing application. It is recommended to use the value of the highest known application ID and increment it by one for your own application ID.

For example, for the application `MyApp`, create a new global identifier called `MYAPP_ID`:

```
MYAPP_ID = 5
```

Also, you have to update the constant `MAX_APP_ID`. Its value must be equal to one above the maximum application ID value.

For example, for the application `MyApp`, as we added one identifier, the `MAX_APP_ID` as to be updated :

```
MAX_APP_ID = 5
#BECOMES
MAX_APP_ID = 6
```

Update the `getApplicationName` function. This function is used to easily retrieve the application name from its global identifier value.

Here is an example for the application `MyApp`.

```
    elif appId == MYAPP_ID :
        return MYAPP_APP_NAME
```

Update the `getApplicationDepth` function. This function is used to easily retrieve the application multiplicative depth from its global identifier value.

Here is an example for the application `MyApp`.

```
    elif appliName == MYAPP_APP_NAME:
        return 2
```

Here is a full example of the `const_id.py` file after adding a global identifier for the application`MyApp`:

```
#SCHEMES
FV_ID = 0
YASHE_ID = 1
FNTRU_ID = 2

#APPLICATIONS
MEDICAL_ID = 1
TOY_ID = 2
FIVEHB_ID = 3
CROISSANT_ID = 4
#ADD HERE THE NEW APPLICATION GLOBAL IDENTIFIER
MYAPP_ID = 5
#INVALID MODE, only here to now the max app id
#UPDATE THE MAX_APP_ID VALUE TO FIT THE MAX APP_ID VALUE +1
MAX_APP_ID = 6

#APPLICATION NAMES
MEDICAL_APP_NAME = "Medical"
TOY_APP_NAME = "Toy"
FIVEHB_APP_NAME = "FiveHB"
CROISSANT_APP_NAME = "Croissant"
#Add MyApp application name
MYAPP_APP_NAME = "MyApp"
```

```
#ANALYSIS_MODES / EXPLORATION_MODES
ANALYZE_ID = 0
EXPLORATION_ID = 1

EXECUTION_ID = 1
COMPLEXITY_ANALYSIS_ID = 2
MEMORY_ANALYSIS_ID = 3
EXPLORATION_ID = 4
#INVALID MODE, only here to know the max mode value
MAX_MODE_ID = 5

def getApplicationName(appli) :
    #...
    elif appli == MYAPP_ID :
        appliName = MYAPP_APP_NAME
    else :
        raise Exception("getApplicationName: appli of ID {} is not in PAnTHErS library"\
            .format(scheme))
    return appliName

#...
def getApplicationDepth(appli) :
    #...
    elif appliName == MYAPP_APP_NAME :
        depth = 6
    return depth
```

4.2.3 Update the templates with your application global identifier

Edit the template files `appli_MyApp.py` and replace the occurences of `APPNUMBER_ID` with your application identifier.

```
$> cd /path/to/panthers/Analyse
$> sed -i 's/APPNUMBER_ID/MYAPP_ID/g' appli_MyApp.py
```

4.2.4 Integrate your application code in the template

Edit the file `appli_MyApp.py` and paste your converted application code from the file `appli.sage` at the end of the function named `computeApplicationMyApp`, between the `KEYGEN()` function call and the `return` statement.

```
KEYGEN()
#PASTE YOUR CODE HERE
return
```

Before the code of your application, you need to assign all inputs parameters. For the example application the following line is used:

```
[i_2_input, i_3_input, i_4_input, i_5_input, i_6_input, i_7_input, i_8_input,\
        i_9_input] = plaintexts
```

At the end of the application code, the outputs also need to be gathered to be returned from the function. For this example application, the following lines are used:

```
#aggregate bits in one byte
res = int(R2(m_0_output.value))*128 + int(R2(m_1_output.value))*64 + \
    int(R2(m_2_output.value))*32 + int(R2(m_3_output.value))*16 + \
    int(R2(m_4_output.value))*8 + int(R2(m_5_output.value))*4 + \
    int(R2(m_6_output.value))*2 + int(R2(m_7_output.value))

    #return decrypted byte
    return res
```

Here is an extract of the resulting `computeApplicationMyApp` for the `MyApp` application:

```
def computeApplicationMyApp(scheme, plaintexts, R, progressBar = 0, step = 0) :
    #...
    KEYGEN()
    #PASTE YOUR APPLICATION CODE HERE
    #Add the list of your input parameters here
    [i_2_input, i_3_input, i_4_input, i_5_input, i_6_input, i_7_input, i_8_input, \
        i_9_input] = plaintexts

    i_2 = ENC(i_2_input);
    i_3 = ENC(i_3_input);
    i_4 = ENC(i_4_input);
    i_5 = ENC(i_5_input);
    i_6 = ENC(i_6_input);
    i_7 = ENC(i_7_input);
    i_8 = ENC(i_8_input);
    i_9 = ENC(i_9_input);

    m_6 = XOR(i_9, i_8);
    n106 = NOT(i_9);
    n104 = NOT(i_8);
    n20 = AND(i_9, i_8);
    n22 = NOT(i_7);
    m_7 = i_9;
    n40 = NOT(i_5);
    n19 = NOT(i_3);
    #...

    #decrypt each encrypted bit
    m_0_output = DEC(m_0);
    m_1_output = DEC(m_1);
    m_2_output = DEC(m_2);
    m_3_output = DEC(m_3);
    m_4_output = DEC(m_4);
    m_5_output = DEC(m_5);
    m_6_output = DEC(m_6);
    m_7_output = DEC(m_7);
    #Return your application outputs

    #aggregate bits in one byte
    res = int(R2(m_0_output.value))*128 + int(R2(m_1_output.value))*64 + \
        int(R2(m_2_output.value))*32 + int(R2(m_3_output.value))*16 + \
```

```
        int(R2(m_4_output.value))*8 + int(R2(m_5_output.value))*4 + \
        int(R2(m_6_output.value))*2 + int(R2(m_7_output.value))

    #return decrypted byte
    return res
```

4.2.5 Create your application input parameters

Then you need to provide plaintext input for your application. The function `setPlaintextsAppliMyApp` is responsible for generating plaintext input, and store them in a list called `plaintexts`.

> 💡 Cingulata converted application usually takes input parameters split in single bits.

> ⚠ Take care of the order of the output parameters. The order as to be the same as used when you retrieve the parameters in the function `computeApplicationMyApp`.

In this example, the function takes input parameters and split them in single bits:

```
#Here the application input takes a single integer value
def setPlaintextsAppliMyApp(intInputValue) :
    #Here the integer value is split in single bit values
    plaintexts = Integer(intInputValue).digits(2, padto = 8)
    plaintexts.reverse()
    #return the list of bits
    return plaintexts
```

> 💡 **Optional**: The input parameters are then processed by the `createSchemeWithPlaintextsAppliMyAppli` function to convert them to `HEMessage` objects. The code provided in the template can usually be reused for Cingulata converted applications. If not, you can edit it manually to fit your application needs.

> 💡 You can see other application examples from applications already integrated in PAnTHErS (files named `appli_*.py`).

4.3 Add your application to PAnTHErS graphical interface

4.3.1 Get the template file

Copy the `TOREPLACEScript.sagescript` template file located in `Templates/Template_Appl`, rename it with your application name and put it in the `Interface` folder.

Here is an example for the sample application `MyApp`:

```
$> cd /path/to/panthers
$> cp Templates/Template_Appli/TOREPLACEScript.sagescript Interface/MyAppScript.sagescript
```

Edit the newly created `MyAppScript.sagescript` file, and replace all instances of `TOREPLACE` with your app name.

Here is an example for the sample application `MyApp` using a single `sed` command

```
$> sed -i 's/TOREPLACE/MyApp/g' MyAppScript.sagescript
```

4.3.2 Update graphical interface source files

`Analyse/parameterchoice.sage` file

In the `parameterchoice.sage` file, update the `executePracticalAnalysis` function.

Add a `elif` statement matching your application global identifier. The purpose of this function is to launch the application analysis process.

Here is an example for the sample application `MyApp`:

```
    elif appli == MYAPP_ID  :
        print("executePracticalAnalysis : MyApp")
        out = check_output(["sage", "-c", "os.system('sage < MyAppScript.sagescript')"])
```

`Interface/interface.py` file

Update the `showApp` function. The purpose of this function is to create graphical elements for the graphical interface (buttons, labels,...).

You have to add some lines of code to add buttons for your application.

Here is an example for the sample application `MyApp`:

```
    #Create a radio button for MyApp
    self.appFrame.boutonAppMyApp = Radiobutton(self.appFrame, variable=self.valueBtnApp, \
        value =MYAPP_ID, text=MYAPP_APP_NAME+" Application", command=self.askForDepth)
    #Position the new button on the interface
    self.appFrame.boutonAppMyApp.grid(row=fromRow+MYAPP_ID-1, column=1, columnspan=7)
```

`Analyse/analyse.py` file

At the beginning of the `analyse.py` file, add an `attach` statement to load your application.

The statement for the application `MyApp` should be:

```
attach("../Analyse/appli_MyApp.py")
```

Next, update the `bench` function. Add a new `elif` statement to allow starting new analysis for your application.

Here is the resulting code for the `MyApp` application.

```
    #If MyApp is selected
    elif appli == MYAPP_ID :
        print(MYAPP_APP_NAME)
        #Start MyApp
        graphsName = appliMyApp(execOrAnalyse, scheme, params, progressBar)
```

Update the `panthersAppli` function. You need to add a new `elif` statement matching your application global identifier. The added code is responsible for starting the analysis and retrieving the results.

Here is the resulting code for the MyApp application:

```
    #MyApp is selected
    elif appli == MYAPP_ID :
        #Start complexity analysis for MyApp
        setParam.compPanthers = panthersAppliMyAppOnce(scheme, schemeParams, \
            COMPLEXITY_ANALYSIS_ID, fileComp, setParam.depth)
        #Start memory cost analysis for MyApp
        setParam.memPanthers = panthersAppliMyAppOnce(scheme, schemeParams, \
            MEMORY_ANALYSIS_ID, fileMem, setParam.depth)
        #If an execution is required
        if execOrAnalyze == EXECUTION_ID :
            #Start a practical execution of MyApp
            setParam.memCalibrated, setParam.compCalibrated = \
                executePracticalAnalysis(appli, EXPLORATION_ID, "MiB_mem_tmp", scheme, \
                schemeParams)
```