Source code obfuscation by mean of evolutionary algorithms

Sébastien Martinez

SOURCE CODE OBFUSCATION BY MEAN OF EVOLUTIONARY ALGORITHMS

Sébastien Martinez

2012

Internship report
version 0.95

Tutor : Sébastien Varrette
Advisor : Benoît Bertholon

University of Luxembourg,
Faculty of Sciences, Technologies and Communications

Master Informatique Spécialité Recherche en Informatique

TELECOM Bretagne

Keywords : Evolutionary Algorithm, Multi Objective EA, Software Obfuscation, Source-to-source Compilation, NSGA-II, Pips, Python, C

Abstract

Usually, when talking about security, the matter is about protecting a computer from intrusions or malicious software. Software obfuscation allows to protect software from piracy by making it able to run without letting the user know its composition. To achieve this goal, distributing binaries instead of source code is not enough since debuggers and decompilers can be used to help get the secret algorithm or data structure one does not want to be known by the user. The solution would be having code that is impossible to understand for the user, but since this goal cannot be reached, the code will have to be complicated enough so that users wanting to get secret algorithms will either give up, either obtain the algorithm when it is obsolete (e.g. when a new, better version is available).

Unlike regular compilation, a good obfuscated program cannot be obtained by applying a fixed sequence of transformation to the given original program. One has to find the sequence of transformations that fits the given program. In this context, the shadobf framework has been designed. Based on Evolutionary Algorithms (EAs), shadobf is used to find that sequence of transformation for any given program written in C. More precisely, extensions calculating obfuscations complexities have been added to the source-to-source compiler Pips, obfuscating transformations were selected and the NSGA-II was implemented in order to optimize the six obfuscation complexities of programs.

The conducted validations showed that using Multi Objective Evolutionary Algorithms (MOEAs) to optimize all the obfuscation metrics gives better obfuscated programs than using single objective EAs to optimize one obfuscation metric. They also showed that considering the execution time of the program as a fitness parameter for the EAs helps obtaining an obfuscated program whose execution time is kept reasonable despite the obfuscating transformations that were applied to it.

Acknowledgements

The work presented in this Master dissertation took place in the University of Luxembourg (UL) in the yComputer Science and Communications (CSC) department. The presented validations were carried out using the HPC facility of the UL.

Thus, I would like to thanks the ULHPC team for their help in the use of the University clusters, `Gaia` and `Chaos`. I would also like to thank the Pips developers team for helping me with the Pips development parts of the project and also the members of the Computer Science and Communications Research Unit in the University of Luxembourg for their warm welcoming and their contribution to the very nice environment I worked in for five months.

And last but not the least, I want to thank Sébastien Varrette and Benoit Bertholon for their help and advices during all my internship.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Source code obfuscation is the action of deliberately making a program's source code non-understandable in some sense, with the constraint that the obtained program will produce the same results as the non obfuscated program for any given input. Its main purpose is to protect the software from malicious users, *e.g.* a company selling a software using a cutting edge algorithm would not like to have its competitors getting the algorithm by just buying their software. A program's obfuscated source code is the consequence of applying a sequence of transformations on its source code. In this sense, the obfuscation process is similar to the one of a source to source compiler.

Just like the sequence of transformation needs to be adapted to a given program to produce an optimized code, the obfuscation process needs to select dynamically a set of transformation, applied in a given order to produce an optimally obfuscated program. In the case of optimizing the generated code for a given execution platform, the metric that can be used to evaluate a given candidate is easy : the execution time on the platform. In the context of obfuscating a source code, this evaluation is more complex. We have identified six metrics in this context : the program length, the cyclomatic complexity, the nesting complexity, the data flow complexity, the fan-in/fan-out complexity and the data structure complexity. Each of these metrics qualify different aspects of the program, and combined together, they can give a fairly accurate quantification of a program obfuscation.

In the case of "standard" source to source compilation, the use of Evolutionary Algorithm (EA) to find the optimal sequence of transformations for a given program is possible, the goal being to reduce the execution time of the program. The same work can be done to find the optimal sequence of obfuscating transformation that would increase the six metrics listed above. EAs are used to solve an optimization problem and can give good results in less time than more deterministic algorithms. The main principle of evolutionary algorithms is to generate a population of solutions and to compare them to choose the best individuals that will be mutated or recombined to form a new population, continuing until an optimal solution is found.

This internship report presents the work that has been done in the context of the SHADOBF project in the Faculty of Sciences, Technologies and Communications in the University of Luxembourg. The goal of this project is to perform a Multi Objective Optimization to find the sequence of obfuscating transformations that will optimize the six obfuscations metrics for a given program.

This report is organized as follows : an a first step, the tools and the algorithms used in this project are presented in §2. For instance, an overview of EAs, projPips, but also of obfuscation technics and of classical compiler transformations is proposed. Then, the section 3 details the implementation operated within the SHADOBF project. §4 is dedicated to the validation of this work on a concrete program, namely a matrix multiplication application. The results given in this section are promising and demonstrate the potential of the proposed approach. We found interesting to present the problems met during this internship, together with a justication of the choices made. This is the purpose of the section 5. §6 expound the seen perspectives while the section 7 conclude this work.

2 Context and motivations

2.1 The University of Luxembourg

This internship took place in the University of Luxembourg (UL) which was founded in 2003, in the Computer Science and Communications (CSC) department. Many countries are represented among the members of this department, contributing to multilingual and multicultural exchanges. This project was conducted under the supervision of Dr S. Varrette, scientific collaborator at the UL and B. Bertholon, PhD student at the UL.

The CSC department is currently the largest research unit of the University of Luxembourg, its staff includes 22 professors and associate-professors, 11 scientific support staff members, 25 post-doc researchers and scientific collaborators and 40 PhD students. The research fields of the department range from the investigation of the theoretical foundations to the development of interdisciplinary applications.

The University of Luxembourg owns 3 clusters : Gaia, Chaos and Granduc, representing 151 nodes, 1556 cores and 14.26 TFlops. The last one is part of the project French Grid5000 which is a scientific instrument for the study of large scale parallel and distributed systems. These clusters have been used for the validation of this project.

2.2 Project objective

Our objective is to optimize the obfuscation of programs written in C using Evolutionary Algorithms (EAs). We use the source-to-source compiler Pips for static analysis of programs and for applying transformations on source code. In order to quantify the obfuscation level of a source code, we use six obfuscation metrics detailed in §3.1: the program length, the cyclomatic complexity, the nesting complexity, the data flow complexity, the fan-in/fan-out complexity and the data structure complexity. The figure 1 illustrates the evolutionary process of obfuscation optimization. We now review the basic

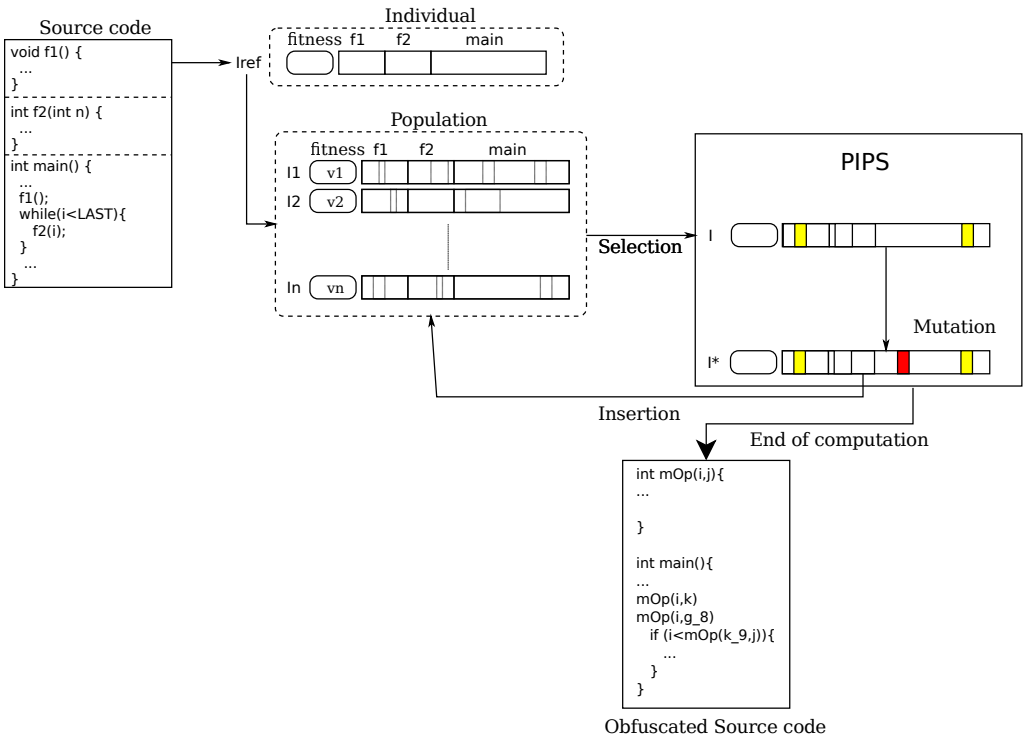


Figure 1: Illustration of the Evolutionary Algorithm (EA) proposed for Shadobf.

concepts used throughout this project.

2.3 Evolutionary Algorithms (EAs)

EAs use mechanisms inspired from the Darwinian theory of evolution to solve optimization problems by the mean of reproduction, mutation, recombination and selection. EAs involve generations of individuals, each individual being a potential solution of the problem to solve. Each generation is built from the previous one through mutation, recombination ... Then, the individuals are evaluated according to a fitness function which eventually involves multiple criterias. The algorithm halts when a stopping condition is met (typically, the fitness value has reached a given threshold or an optimal solution ha been found). Note that the convergence of EAs towards "good" solutions has been formally proven in [10]. Since the convergence of an EA can very be long one can chose to stop an EA when a fixed number of generation have been computed or when the fitness of the individuals does not change much between two consecutive generations.

Algorithm 1: General scheme of an EA.

```
t := 0;
Generation(X_t) // generate the initial population
Evaluation(X_t) // evaluate population
while Stopping criteria not satisfied do
    X̂_t = ParentsSelection(X_t); // select parents
    X'_t = Modification(X̂_t); // cross-over + mutation
    Evaluation(X'_t) // evaluate offspring
    X_{t+1} = Selection(X_t, X'_t) // select survivors for the next generation
    t := t + 1;
```

For single criteria based evolution, a simple EA was implemented. Its follows the general scheme presented in algorithm 1.

EA components in Shadobf

The EAs used in the SHADOBF project are formalized as follows.

Individual : An individual is a source code composed of modules (*i.e.* functions).

Original individual : The very first individual (from which all the individuals were mutated) which derives from the original source code we would like to obfuscate.

Degenerate individual : A degenerate individual is either an invalid individual (see **Validation of individuals**) or an individual whose source code cannot be compiled.

Population : A set of individuals

Mutation : A mutation is a sequence of transformations applied to a module of an individual. It produces a new offspring whose parent is the latter individual.

Selection : selection of individuals to form a new parent population that will produce a new children population through mutation. Depending on the used EA, individuals can by sorted by their fitness or by their domination rang (see NSGA-II *Algorithm*).

Validation of individuals : Every generated individual has to give the same output as the original individual for the given input. The validation of individuals is made by compiling them and running them on a predefined benchmark. Individuals whose output differs from the one given by the original individual and those whose execution time is considered too long are said *invalid individuals*. Note that as the benchmark does not obviously cover all possible input values, a valid individual may still be degenerate for some specific input such that we denote here a probabilistic validity.

Evaluation of individuals / fitness : The quantification of the quality of an individual. In this context, the quality of individuals is quantifies by their obfuscation complexity.

Note that *cross-over* is not considered in our work as we didn't find a satisfiable way to perform this evolutionary operation without degrading the obfuscation capability of our algorithm[1].

The NSGA-II Algorithm

Simple EAs cannot handle fitness composed of multiple criteria, the main problem being to sort a newly formed generation. Therefore, one has to use specific algorithm handling multiple criteria. These algorithms are called Multi Objective Evolutionary Algorithms (MOEAs) and the reference algorithm

[1] More precisely, a cross-over assumes a shared (common) pattern between the individuals.

in this context is NSGA-II [5]. This algorithm is an enhancement of the NSGA algorithm introducing elitism and reducing the complexity of the algorithm from $O(mN^3)$ to $O(mN^2)$ where N is the size of the population and m is the number of criteria. The side effect of this acceleration is the increasing of the storage from $O(N)$ to $O(N^2)$.

The order relation mainly used in Multi Objective Evolutionary Algorithms (MOEAs) is the domination relation. Assuming that the objective values to optimize belongs to $\mathbb{R}$, the n fitness values of a given individual form a vector in $\mathbb{R}^n$. The domination relation forms a partial order on $\mathbb{R}^n$. In our case, the objectives (obfuscation complexities, execution time) are real numbers.

Definition 1 (Domination)**.** *An vector x of $\mathbb{R}^n$ dominates a vector y of $\mathbb{R}^n$ if and only if $\exists i$ such that $x_i < y_i$ and $x_j \leq y_j \forall j \in [0..n-1]$*

The NSGA-II algorithm uses Pareto fronts to sort the individuals of a generation. Pareto fronts are sorted by rank : F_1 is the set of the individuals that are not dominated by any other individual, F_n is the set of the individuals that are dominated by exactly $n-1$ other individuals. In order to evaluate the density of solutions around a given point a *crowding distance* is calculated for each individual (or point) then the individuals are sorted according to the *crowded comparison*. The crowding distance of an individual estimates the distance between this individual and its closer neighbors.

Algorithm 2: Computing the crowding distance of individuals.

S = set of individuals
l = size of S
for *each criteria* **do**
  sort S according to the criteria
  $S[1] = S[l] = \infty$
  f_{min} = minimum value of criteriaă
  f_{max} = maximum value of criteriaă
  for $i = 2$ *to* $(l-1)$ **do**
    $S[i]_{distance} = S[i]_{distance} + (S[i+1]_{criteria} - S[i-1]_{criteria})/(f_{max} - f_{min})$

Definition 2 (Crowded Comparison)**.** *Let $\geq$ be the operator of* crowded comparison *and i and j individuals. $i \geq j$ if and only if $(i_{rank} < j_{rank}) or ((i_{rank} = j_{rank}) and (i_{crowdingdistance} > j_{crowdingdistance}))$*
Remark : The rank of an individual is the rank of the Pareto front it is in

When the individuals of a generation have been sorted, a children generation of the same size is built from that generation (called the parent generation) and the union of these two generations is sorted to create the next parent generation. The NSGA-II algorithm replaces the sorting method used in the `Selection` step in the algorithm detailed on algorithm 1 by its own method implying Pareto fronts.

Related work

Guelton and Varrette used EAs to optimize source code in [6], the fitness being the execution time. They used Pips for applying transformations on the source code and developed `pypsearch` in this context. Comparing the results brought by the evolutionary approach (compared to a complete approach and a glutton approach), they found that the evolutionary approach gave an optimal result like the complete approach in times comparable to the glutton approach that was unable to give an optimal result. This work makes use of the same idea yet on a different objective: rather than optimizing the code for an efficient execution on a given computing platform, we try here to obfuscate it. To make this last notion more clear, the next section provides an overview of the main obfuscating techniques.

2.4 Obfuscation techniques

Obfuscation transformations can take many forms, and can affect many parts of a program : the data structures used, the functions called or even the textual representation of the source code (e.g. Removing

comments). When compiling regular code, a fixed sequence of transformations is applied to the code and this sequence does not change with the code given to the compiler. When obfuscating code, the sequence of transformations applied to code will have an effect that depends on the code. Therefore, we can estimate an absolute quality of a transformation but we cannot find a transformation that is better than the other in all circumstances. Colberg et al. published a taxonomy of obfuscating transformations [4] and defined the *quality* of an obfuscating transformation.

Definition 3 (Obfuscating Transformation)**.** *Let $P \xrightarrow{\tau} P'$ be a transformation of a source program P into a target P'. $P \xrightarrow{\tau} P'$ is an* obfuscation transformation *if P and P' have the same* observable behavior*. More precisely, the following conditions are respected :*

- *If P fails to terminate or terminate with an error condition, then P' may or may not terminate*
- *Otherwise P' must terminate and produce the same output as P*

Observable behavior can be defined as being the behavior experienced by the user. *i.e.* everything the user can notice at first sight. Hence, if P' has side effects (new created files, network communications ...) that are not noticed by the user, it can still have the same observable behavior (provided it has the same user experienced effects as P).

In order to evaluate the quality of obfuscating transformations, we need to define several transformation properties and metrics. The three main properties being *Potency*, *Resilience* and *Cost*.

Potency can be considered as a measure of a transformation usefulness in its task of hiding the intent of the program coder. *Potency* can be seen as a measure of an obfuscation transformation efficiency toward human readers. *Resilience* can be seen as a measure of an obfuscation transformation efficiency toward automatic deobfuscators (as an opposition to potency). Transformation cost measures the penalty induced by the transformation : a transformation can make the program to use more memory or more time. These three measures compose the quality of a transformation.

We now formalize these definitions.

Definition 4 (Transformation Potency)**.** *Let τ be a behavior-conserving transformation, such that $P \xrightarrow{\tau} P'$ transforms a source program P into a target program P'. Let $E(P)$ be the complexity of P. $\tau_{pot}(P)$, the* potency *of τ with respect to a program P is a measure of the extent to which τ changes the complexity of P. It is defined as:*

$$\tau_{pot}(P) = E(P')/E(P) - 1$$

τ is said a potent obfuscating transformation if $\tau_{pot}(P) > 0$.

In this definition E a measure of complexity. Since there are many software complexity measures, one has to be chosen. Several metrics will be listed in the next subsection.

Software complexity metrics are often subjective and some transformation will increase the program complexity according to the metric in use while the deobfuscation of these transformations are really simple for a machine though uneasy for a human reader as we will see further. Hence, potency can be pictured as a measure of a transformation usefulness toward human readers.

To measure a transformation usefulness toward automatic deobfuscators, resilience has to be introduced. Resilience takes two parameters in consideration : *Programmer Effort* (the amount of time taken to build an automatic deobfuscator that will efficiently reduce the potency of τ) and *Deobfuscator Effort* (the execution time and the memory space required by the obfuscator to reduce efficiently the potency of τ).

Definition 5 (Transformation Resilience)**.** *Let τ be a behavior-conserving transformation, such that $P \xrightarrow{\tau} P'$ transforms a source program P into a target program P'. $\tau_{res}(P)$ is the* resilience *of τ with respect to a program P. $\tau_{res}(P) =$* one-way *if information is removed from P such that P cannot be reconstructed from P'. Otherwise, $\tau_{res} = Res(\tau_{Deobfuscator_{effort}}, \tau_{Programmer_{effort}})$, where Res, the Resilience is the function defined by the matrix defined in the matrix in Figure 2.*

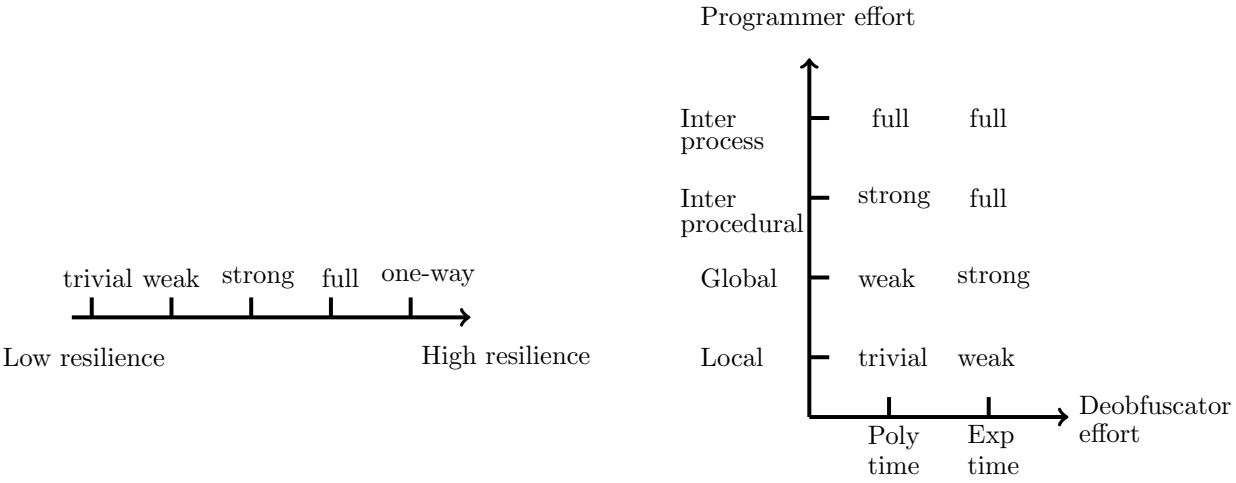


Figure 2: Resilience of obfuscating transformations : Scale of values (left) and resilience matrix (right).

Transformations often introduce some loss in the program. The program can need more memory space or more time to terminate after the application of a transformation. Transformation cost introduces this notion.

Definition 6 (Transformation Cost)**.** *Let τ be a behavior-conserving transformation, such that $P \xrightarrow{\tau} P'$ transforms a source program P into a target program P'. $\tau_{cost}(P)$ is the extra execution time/space of P' compared to P. $\tau_{cost}(P)$ is said :*

- dear *if executing P' requires* exponentially *more resources than P*
- costly *if executing P' requires $O(n^p), p > 1$, more resources than P*
- cheap *if executing P' requires $O(n)$ more resources than P*
- free *if executing P' requires $O(1)$ more resources than P*

Potency, resilience and cost compose the quality metric of obfuscating transformations.

Definition 7 (Transformation quality)**.** *$\tau_{qual}(P)$, the quality of a transformation τ, is defined as the combination of the potency, resilience, and cost of τ*

$$\tau_{qual}(P) = (\tau_{pot}(P), \tau_{res}(P), \tau_{cost}(P))$$

2.5 Classical compiler transformations

Pips provides several transformations aiming to optimize the parallelism of a source code. Some of these transformations can be used as obfuscating transformations. In this context, the following transformations have been selected :

Loop unrolling : Unrolls a loop by a given rate, reducing its range.

Inlining : Replaces a function call by the function content

Localize declarations : Relocate variable declarations the closer possible to their usage.

Outlining : Replaces a block of code by a call to a function that executes this block (defines this function if needed).

Index set splitting : Splits a loop in two by dividing its range.

Unfolding : Cascade code inlining.

Flatten code : Increases the scope of local variables by moving them up in the syntax tree of the program and unrolls loops in order to enlarge basic blocks in the code.

Loop fusion : Fusions two loops.

Coarse grain parallelization : Parallelize loops basing on convex array regions.

A perspective of this work, as outlined in §3.2 consists in defining new compiler transformations useful for an obfuscation process.

2.6 The Shadobf project

The SHADOBF project makes use of PIPS for static analysis of programs and applying transformations to it. The choice for the compiler to use was between the `Clang` fronted for the LLVM compiler and the source-to-source compiler PIPS. PIPS was chosen because of my past experience with this software and its development team. Moreover, PIPS brought PYPS (its Python frontend), transformations and `pypsearch` (a PYPS library for using genetic algorithms with PIPS) that were already implemented and this greatly facilitate the running of this internship toward concrete results.

2.6.1 Pips and its frontend Pyps

PIPS [1] (*Parallélisation Interprocédurale de Programmes Scientifiques*) is an interprocedural source to source compiler analyzing C and Fortran programs and transforming them to optimize parallel executions of these programs. Hence, the purpose of many of PIPS transformations is to accelerate loop executions. This can be achieved for instance by including OPENMP pragmas or replacing some operations by SSE instructions. However, some of PIPS transformations can be used for obfuscation like loop unrolling or function inlining. A PIPS process involves several passes each of them responsible for a given transformation. For a given program, its functions are called *modules*.

PIPS passes are called through its Python front-end PYPS. PYPS is linked to PIPS using **swig**[3] and its design is object oriented and modular. The `workspace` object models a program and offers special functionalities. Different types of workspaces are available in PYPS, each one offering different functionalities (*e.g.* benchmarking modules).

More details on PIPS will be provided in §3.3.1.

2.6.2 `Pypsearch`

`Pypsearch` is a PYPS module which implements searching algorithms for automatic optimisation of programs execution time using PIPS. Among the searching algorithms, a single criteria genetic algorithm has been implemented. `Pypsearch` uses Remote Procedure Calls to handle several workspaces (and therefore several individuals) at a time. Other details on `Pypsearch` will be given in §3.3.2.

2.6.3 Shadobf

Due to the previous work with PIPS and Genetic Algorithms in the framework of [6], PIPS has been chosen for syntax analysis, obfuscation complexities computing and most importantly, for applying in a flexible way the transformations on the source code, hence the choice to work on programs written in C. Moreover, several passes dedicated to the previous work on code optimization that were already implemented in PIPS like outlining or loop unrolling happen to have a fairly good obfuscating power.

The six most common obfuscation metrics (Summarized bellow) listed by Colberg et al.[4] have different semantical meaning and one cannot simply chose only one of these metrics to quantify the obfuscation of a program. Therefore, SHADOBF implements Multi Objective Evolutionary Algorithms (MOEAs) in order to optimize the obfuscation of programs along the six main obfuscation metrics.

μ_1 **Halstead's complexity (Program length) :** the μ_1 complexity of a program is the number of operator and operands of its source code.

μ_2 **MacCabe's complexity (Cyclomatic Complexity) :** the μ_2 of a program is the number of predicates in its source code.

μ_3 **Nesting complexity :** the μ_3 of a program is the maximum nesting depth of the predicates in its source code.

μ_4 **Data flow complexity :** the μ_4 of a block is the number of variable that are used but not declared in this block. The μ_4 of a program is the sum of the μ_4 of its blocks.

μ_5 **Fan-in/out complexity :** the fan-out of a function is the number of its write effects plus the number of functions called by that function. The fan-in of a function is the number of read effects plus the number of function that call that function. The $\mu 5$ of a function is the product of is fan-in by its fan-out. The $\mu 5$ of a program is the sum of the μ_5 of its functions.

μ_6 **Data structure complexity :** the μ_6 of a program is the sum of the size of its variables.

Increasing a program's obfuscation often comes along with increasing the program's execution time. For instance, the outlining transformations will increase the number of function calls, increasing the execution time of the program. On the other hand, applying transformations chosen at random generation after generation might create degenerate individuals that would not give the same output as the original individual when given the same input or that would simply crash while running. The goal being to obtain an optimal individual giving the exact same output as the original one within an execution time reasonably increased, the degenerate individuals must be removed and replaced by new ones so do the individuals whose execution time is too long.

The development operated ion this project used to comprise three major steps :

1. Implementing passes computing the obfuscation complexities in Pips
2. Selecting Pips transformations that could be used to increase a program's obfuscation
3. Writing a Python program using Pyps and implementing the evolutionary algorithms we want to experiment

3 Implementation details

3.1 Implementing the Metrics

For each one of the six obfuscation metrics, a dedicated pass was added in Pips. Each one of these passes computes an obfuscation complexity for a given module. Due to the C programming language specificities or to the internal representation of the code by Pips, the definition of some complexities has been revised, and for some complexities, only an estimation of its value is possible.

The computation of each metric relieves exclusively on the static analysis of the program. Using dynamic analysis to calculate obfuscation complexity, although being more precise in some cases, would have been more complex because it would have required the generation of enough input to represent all the possible inputs.

Obfuscation complexities are stored in the Pips database as an union type called `obfuscation_complexity` that can be either a `mu1_complexity`, a `mu2_complexity` ... All the obfuscation complexities are stored in a list of type `obfuscation_complexity_list`. Then, the Pips databases associates each `obfuscation_complexity_list` to a module.

For instance, the μ_3 complexity of the module `main` of a program is stored as an `obfuscation_complexity` in the `obfuscation_complexity_list` associated to the module `main`.

To access a module's obfuscation complexity in Pyps, a property was added to the `module` class. This property triggers the call to a function retrieving the complexity in the Pips database.

The obfuscations complexities are computed for each module of a program and the reduction operation we use to obtain the complexity of the program is the addition. Using the addition operation allows to take every module into account when calculating the obfuscation complexity of the program. Notice that other binding operations might be considered in the reduction of the complexities among all modules than the addition. The impact (and definition) of such other operations needs to be evaluated and is kept as future work.

3.1.1 The μ_1 complexity (Program length)

Defined as the number of operator and operands of a program, this complexity also called Halstead's complexity is directly linked to the program's code length. The internal representation of Pips being closer to a Fortran representation, the `compute_mu1_complexity` pass may not always give an exact value of a module's μ_1 complexity. For instance, the internal representation of Pips tends to replace *for* loops by Fortran *do* loops whenever it is possible. Yet we consider the returned value sufficiently close to the real number of operations and operands. In practice, the *compute_mu1_complexity passes* analyses the module's Abstract Syntax Tree (AST) to count the number of operators and operands of the module. The computation of the μ_1 complexity is illustrated in listing 1.

```
#include <stdio.h>

/* Computed mu1 complexity of this module: 16 */
int main(){
    int i;
    for (i=0;i<15;i++){
        printf("iteration %d\n",i);
    }
    return 0;
}
```

Listing 1: Example of μ_1 complexity.

3.1.2 The μ_2 complexity (Cyclomatic complexity)

McCabe's complexity [9] is computed from the Control Flow Graph (CFG) C associated to the considered program. Its value is $e - n + p$ where e stands for the number of edges of C, n is the number of vertices of C and p corresponds to the number of connected components of C. This can be fairly approximated by the number of predicates in the program plus one. Hence the `compute_mu2_complexity` pass counts the number of loops and tests in the program it is run on. In practice, the `compute_mu2_complexity` pass analyses the AST of the module to count the number of loops (*for* and *while*) and tests (*switch* are counted as N nested tests, where N is the number of cases in the *switch* statement). On the example of listing 2, the calculated μ_2 complexity of the `main` module is 6.

```
#include <stdio.h>

/* Computed mu2 complexity of this module: 4*/
/* Computed mu3 complexity of this module: 2*/
int main(){
    int i = 0;
    int j = 0;
    while (i<16){
        printf("iteration %d\n", i);
        if (i>2){
            j+=i;
        }
    }
    if (j >= 3){
        printf("j is greater than 3\n");
    }
```

```
    return 0;
}
```

Listing 2: Example of μ_2 and μ_3 complexity.

3.1.3 The μ_3 complexity (Nesting complexity)

The nesting complexity defined by Harrison [7] uses the CFG of the program to consider the nesting of nodes when measuring the complexity of a program. For each node of the control flow, the complexity of the program is incremented by the Halstead's complexity of this node plus the number of nodes within its range plus the Halstead's complexity of each of theses nodes.

Implementing the actual definition of Harrison's complexity would introduce redundancy with the Halstead's complexity (the μ_1 complexity) and McCabe's complexity as this complexity tries to replace both of these complexities. Therefore, we redefined the μ_3 complexity of a module as the maximum depth of predicate nesting in the module. This new definition will complement MacCabe's complexity. MacCabe's complexity will stimulate the appearance of predicates and our μ_3 complexity will stimulate their nesting (and only their nesting). Like in the `compute_mu1_complexity` and the `compute_mu2_complexity` passes, the AST of the module is analysed to find the maximum depth of the module's statements in the module. The internal representation of Pips representing `switch` statement as nested `test` statements, the depth of a `case` is its rank in the list of cases of the `switch` statement. On the example of figure 2, the calculated μ_3 complexity of the **main** module is 2.

3.1.4 The μ_4 complexity (Data flow complexity)

The μ_4 complexity reflects the data flow complexity by adding the distances between the declaration of variables and their actual usage. For instance, global variable are usually harder to manipulate than local one and more generally, the longer a variable's scope is, the more complicated it is for an attacker to track the variable manipulation. The goal of the μ_4 complexity is therefore to count the number of variables that are defined outside of the blocks where they are used in (e.g. global variables, same variables used in different blocks). Here, blocks are semantical blocks of the module, and although it would be more accurate for the splitting to be adapted to the given module semantic, we had to decide for a splitting strategy that would be the same for every module that can be met since we want the pass to be completely automatic.

In practice, the splitting strategy we selected operates as follows : loops (condition and body), tests branches and syntax blocks (parts of code between brackets) are considered as atomic blocks. Moreover, when a loop, test or black statement ends, a new block is opened. Loops, tests and syntax blocks constitute natural semantic blocks with their own coherence, and a variable used in two areas of the code separated by a block is harder to estimate as it could have been modified in that block. This splitting pattern is easily applied when given the module's AST and offers an acceptable solution, although not optimal (an optimal splitting could be found by adapting the splitting more precisely to the given program), to the splitting problem.

When the module has been split into blocks, its μ_4 complexity is the sum of the μ_4 complexity of its blocks. The μ_4 complexity of a block is the number of variables used but not defined in this block. If a same variable is used more than once in a block, it will we counted every time it is used without being defined in the current block. *Remark: This definition of the data flow complexity considers variable affectation as variable usage and not as variable definition. Also, other binding operations might be considered in the reduction of the μ_4 complexity* In the example on listing 3, the computed μ_4 complexity of the **main** module is 15.

3.1.5 The μ_5 complexity (Fan-in/out complexity)

Henry and kafura defined a software measure based on information flow [8] as $length*(fan_{in}*fan_{out})**2$, *length* being the μ_1 complexity or McCabe complexity of the module. The fan_{in} of a module M is the number of local flow into M plus the number of data structures read by M, The fan_{out} of a module M is the number of local flow from M plus the number of data structures in which M writes.

To avoid redundancy and to keep the complexity linear (the power of 2 only being a weight), the μ_5 complexity has been redefined has the product $fan_{in} * fan_{out}$. In C, the usage of pointers making the

```c
#include <stdio.h>

/* Computed mu4 complexity of this module: 15*/
int main(){
    int size = 10;
    int i,k;
    int a[size][size];
    int b[size][size];
    for(i=0;i<size;i++){
        for(k=0;k<size;k++){
            int j = 5*k;
            a[i][k] = i+k;
            b[k][i] = j;
        }
    }
    return 0;
}
```

Listing 3: Example of μ_4 complexity.

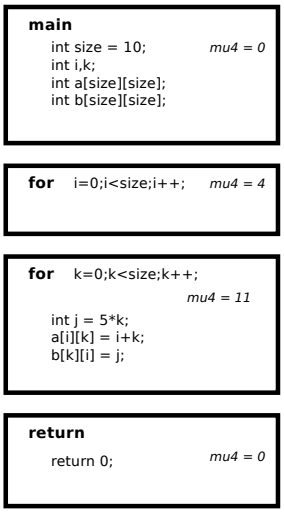


Figure 3: Illustration of the splitting strategy.

notion of variable less sensible than the notion of memory slot the fan_{in} and fan_{out} definitions can be understood this way : The fan_{in} of a module M is calculated as the number of modules calling M plus the number of read effects of M. The fan_{out} of a module M is calculated as the number of modules called by M plus the number of write effects of M. In the example on listing 4, the calculated μ_5 complexity of the module **foo** is 3 (*foo* is called once and reads the variables b and matrix writes, it writes to the variable matrix, its fan-in is 3, its fan-out is 1) and the μ_5 of the module **main** is 2. The total μ_5 of the program is 5.

```c
#include <stdio.h>

/* Computed mu5 complexity of this module: 3*/
int matrix[10][10];
int foo(int b){
    int i,j;
    for(i=0;i<10;i++){
        for(j=0;j<10;j++){
        matrix[i][j]=matrix[i][j]*b;
        }
    }
}
/* Computed mu5 complexity of this module: 2*/
int main(){
    int i,j;
    int k = 4;
    for(i=0;i<10;i++){
        for(j=0;j<10;j++){
        matrix[i][j]=i+j;
        }
    }
    foo(k);
    return 0;
}
```

Listing 4: Example of μ_5 complexity.

3.1.6 The μ_6 complexity (data structure complexity)

The regular definition of the μ_6 complexity of a variable is the dimension of this variable (e.g. the μ_6 complexity of a vector is 1, the μ_6 complexity of a matrix is 2). Since in C, the dimension of a data

structure does not have the same semantic value as in other languages. The μ_6 complexity of a variable has been redefined as its size (in bytes). Yet, this poses a problem with pointers. For instance : whereas a picture is often modelled as a two dimension matrix of pixels, it could be stored in a one dimension array where the pixel (i,j) will be at the $j + width * i$ position. Moreover, if an array has been declared with no dimension and its memory is dynamically allocated later in the code its dimension cannot be known statically. Dynamic allocation should be taken into account as a variable whose size is dynamically allocated is more complex that statically sized variable from an obfuscation point of view. It is handled using the μ_1 complexity : the number of operands and operators inside the arguments of the called allocation function (e.g. `malloc`, `calloc`) is added to the mu_6 complexity of the variable on which the allocation function is applied.

In order to deal with dynamic sized arrays (e.g. `int v[n];`) the `approximate_eval` pass is called before calculating the program's μ_6 complexity to replace variable by an upper-bound of their values. If the size of a variable cannot be calculated statically and an upper-bound of its size cannot be calculated then the variable size is assumed to be 1. Finally, for the full module, its μ_6 complexity sums the μ_6 complexity of each involved variable. In the example on listing5, the calculated μ_6 complexity of the module **main** is 63 (in the declaration of k, i has been approximated to 3 by the pass *approximate_eval*).

```
#include <stdio.h>
#include <stdlib.h>

typedef struct { int champ1; int champ2; long champ3; } t;
/* Computed mu6 complexity of this module: 63*/
int main(){
    char * s = "ma variable";
    char x[] = "hello world ha ha";
    char r[5] = "abcde";
    int i;
    t myt = {2,3,4};
    if (myt.champ1 > 2){
        i = 3;
    }else{
        i = 1;
    }
    int k[i];
    return 0;
}
```

Listing 5: Example of μ_6 complexity.

3.2 Selecting the Pips transformations

Although the main purpose of Pips transformations is to enhance parallelism of a program, several transformations can be used for source code obfuscation. Apart from the `remove_comments` and `scramble_variable_names` passes that were developed in the context of the Shadobf project, all the presented transformations were already implemented in Pips. The table 1 page 23 summarizes the chosen transformations, their quality (in an obfuscation point of view) and their impact of the six obfuscation complexities.

3.2.1 Loop unrolling

The `unroll` pass unrolls a loop by a given rate. The figure 4 shows an example of loop unrolling for a rate of 4. Loop unrolling increases highly the μ_1 complexity (the higher the rate, the higher the μ_1 increasing) and has a small impact on the μ_2, μ_4 and μ_6 complexities because of the added loop and variables.

Considering the μ_1 complexity, the loop unrolling transformation has a high potency. This transformation is also free since it requires $O(1)$ more resources than the original form. But loop unrolling has a trivial resilience : reversing the transformation only implies polynomial time and local work.

```c
#include <stdio.h>

int main(){
    int i;
    int a[15];
    for(i=0;i<15;i++){
        a[i]=i;
    }
    return 0;
}
```

(a) original program

```c
#include <stdio.h>

int main(){
   int i;
   int a[15];
   //PIPS generated variable
   int LU_NUB0, LU_IB0, LU_IND0;
l99999:    LU_NUB0 = 15;
   LU_IB0 = 3;
   for(LU_IND0 = 0; LU_IND0 <= LU_IB0-1; LU_IND0 += 1) {
      a[LU_IND0*1+0] = LU_IND0*1+0;
l99998:       ;
   }
   for(LU_IND0 = LU_IB0; LU_IND0 <= LU_NUB0-1; LU_IND0 += 4) {
      a[(LU_IND0+0)*1+0] = (LU_IND0+0)*1+0;
      a[(LU_IND0+1)*1+0] = (LU_IND0+1)*1+0;
      a[(LU_IND0+2)*1+0] = (LU_IND0+2)*1+0;
      a[(LU_IND0+3)*1+0] = (LU_IND0+3)*1+0;
l99997:       ;
   }
   i = 0+MAX0(LU_NUB0, 0)*1;
   return 0;
}
```

(b) transformed program with a rate of 4

Figure 4: Example of loop unrolling.

3.2.2 Inlining

The `inlining` pass includes a function's code into a function that calls the latter. The figure 5 shows an example of function inlining. Function inlining highly increases the μ_1 complexity of the modules were the function has been inlined, it also increases the μ_6 complexity of these modules. On the other hand, it reduces the μ_5 complexity of the inlined function since it will be called in less functions. Moreover, the other complexities can be increased for the modules where the function has been inlined as the inlined function will bring new predicates, increase nesting levels, ...

Considering the μ_1 or μ_6 complexity, the inlining transformation has a high potency provided the inlined function is long enough and its cost is free (it does not add many more operations in the program). Its resilience is strong because reversing the transformation would require polynomial time and inter procedural analysis of the program. If the definition of the inlined function is removed, then the resilience is one-way.

```c
#include <stdio.h>

int foo(int b){
    if (b>5)
        return 3;
    else
        return 5;
}
int main(){
    int i = 2;
    int j;
    j = foo(i);
    return 0;
}
```

(a) original program

```c
#include <stdio.h>

int main(){
    int i;
    //PIPS generated variable
    int _return0;
    i = 2;
    int j;
    {
        //PIPS generated variable
        int b0 = i;
        if (b0>5)
            _return0 = 3;
        else
            _return0 = 5;
    }
    j = return0;
    return 0;
}
```

(b) transformed program

Figure 5: Example of function inlining.

3.2.3 Unfolding function calls

The `unfolding` pass inlines function calls recursively in order to remove every function call from a given module. Its properties are the same as the `inline` pass with higher influences on the complexities and its resilience is "full" (the time necessary to revert the transformation becomes exponential).

3.2.4 Localizing declarations

The `localize_declaration` pass moves variables declarations closer to their uses. The figure 6 shows an example of declaration localization. Declaration localization increases the μ_6 complexity by replacing long scope global variables by many small scope local variables. The other complexities usually stay unchanged.

Considering the μ_6 complexity, the `localize_declaration` transformation has a medium potency and it has a free cost. Its resilience is weak : it requires a polynomial time and an analysis of the global data flow to reverse this transformation.

```c
#include <stdio.h>

int main(){
    int i,j;
    int a[15][15];

    for(i=0;i<15;i++){
        for(j=0;j<15;j++){
            a[i][j] = i+j;
        }
    }

    for(i=0;i<15;i++){
        for(j=0;j<15;j++){
            a[i][j] = i+j;
        }
    }

    return 0;
}
```

(a) original program

```c
int main(){
    int i;
    int a[15][15];

    for(i = 0; i <= 14; i += 1) {
        //PIPS generated variable
        int j;
        for(j = 0; j <= 14; j += 1)
            a[i][j] = i+j;
    }
    for(i = 0; i <= 14; i += 1) {
        //PIPS generated variable
        int j;
        for(j = 0; j <= 14; j += 1)
            a[i][j] = i+j;
    }
    return 0;
}
```

(b) transformed program

Figure 6: Example of `localize_declarations`.

3.2.5 Removing Comments

The `remove_comments` pass remove every commentaries from the code. Its does not have any impact of the six complexities. Therefore, its potency is zero. However, its cost is free and its resilience is one-way.

3.2.6 Scrambling variable names

The `scrambe_variable_names` pass aims to remove any semantic content in the name of variables by renaming them with random generated names. This pass is actually under development and was not in the pool of transformations when running the experiments presented in this report.

3.2.7 Outlining

The `outling` pass defines a function which body is a loop or a loop nest selected in the given module. That loop (or loop nest) is then replaced by a call to the created function. The figure 7 shows an example of function outlining. This transformation increases the μ_5 complexity.

Considering the μ_5 complexity, the outlining transformation has a high potency and has a free cost. Moreover, it has a strong resilience as inverting the transformation would require inter procedural analysis.

```c
#include <stdio.h>

int main(){
    int i,j;
    int a[15][15];

    for(i=0;i<15;i++){
        for(j=0;j<15;j++){
            a[i][j] = i+j;
        }
    }

    return 0;
}
```

(a) original program

```c
#include <stdio.h>

void foo(int a[15][15]){
    //PIPS generated variable
    int i, j;

l99998:
    for(i = 0; i <= 14; i += 1)
l99999:
        for(j = 0; j <= 14; j += 1)
            a[i][j] = i+j;
}
int main(){
    int i, j;
    int a[15][15];
l99998:    foo(a);
    return 0;
}
```

(b) transformed program

Figure 7: Example of function outlining.

3.2.8 Index set splitting

The `index_set_splitting` splits a loop into two loops. The fist loop ends at a given iteration and the second starts on the next iteration. The figure 8 shows an example of index set splitting on the seventh iteration. This transformation increases the μ_2 and the μ_1 complexity.

Considering the μ_2 complexity, the index set splitting transformation has a low potency, a free cost and a weak resilience (reverting the transformation only require global analysis).

```c
#include <stdio.h>

int main(){
    int i;
    int a[15];

    for(i=0;i<15;i++){
        a[i] = i;
    }

    return 0;
}
```

(a) original program

```c
#include <stdio.h>

int main(){
    int i;
    int a[15];
    //PIPS generated variable
    int i0;
l99999:
    for(i = 0; i <= MIN(7, 14); i += 1)
        a[i] = i;
l99998:
    for(i0 = MIN(7, 14)+1; i0 <= 14; i0 += 1)
        a[i0] = i0;
    return 0;
}
```

(b) transformed program

Figure 8: Example of index set splitting at iteration 7.

3.2.9 Code flattening

The `flatten_code` pass aims to enlarge basic blocks of the given module. It moves declarations to a higher level in the AST, making global variable out of local variables and unrolls completely loops with constant bounds. The figure 9 shows an example of code flattening. This transformations increases the μ_4 complexity and the loop unrolling increases the μ_1 complexity.

Considering the μ_4 complexity, Code flattening has a medium potency and a free cost. Its resilience is strong since reversing the transformation would require inter procedural analysis of the variable lifetime.

```c
#include <stdio.h>

int main(){
    int i;
    int a[4];

    for(i=0;i<4;i++){
        int k = i+5;
        a[i] = 5;
    }

    if (a[0] == 7){
        int k = a[1];
    }
    return 0;
}
```

(a) original program

```c
#include <stdio.h>

int main() {
    int i;
    int a[4];
    //PIPS generated variable
    int k, k_0;
    k = 0+5;
    a[0] = 5;
    k = 1+5;
    a[1] = 5;
    k = 2+5;
    a[2] = 5;
    k = 3+5;
    a[3] = 5;
    if (a[0]==7)
        k_0 = a[1];
    return 0;
}
```

(b) transformed program

Figure 9: Example of code flattening.

3.2.10 Loop fusion

The `loop_fusion` pass tries to merge a given loop with the following loop provided both loops have the same index and iteration set. Loop fusion is not an obfuscation transformation so to speak, but it is interesting as it could break the semantic unity of two loops by merging them together and create blocks of instructions that could be operated on by other transformations.

3.2.11 Parallel loop generation

`Parallel loop generation` is a sequence of transformations used in `pypsearch` to generate parallel loop and apply OpenMP pragmas on them. The sequence of transformation is composed of the following passes : `privatize_module`, `coarse_grain_parallelization` and `ompify_code`.

The `privatize_module` pass marks local variables of loops as private, preparing the module for the passes `coarse_grain_parallelization` and `ompify_code` that will add OpenMP pragmas on the code when possible.

This transformation introduces parallelism in the code but has a very low obfuscating power. Since pragmas are not counted for the calculation of the μ_1, this transformations does not change the complexities. Reverting this transformation is quite easy, hence its resilience is trivial. However, OpenMP introducing reduction can be harder to understand for readers who are not familiar with the OpenMP standards, giving to this transformation a weak resilience.

On the other hand, introducing OpenMP pragma in the code can accelerate its execution time for a reasonable cost, which can be interesting when applying many execution slowing obfuscating transformations.

```
int foo(int a [15][15], b [15][15]){
    int i,j;
    int c [30];

    for (i=1;i<14;i++){
        for (j=1;j<14;j++){
            c[i+j]=a[i-1][j]+b[i][j]*a[i][j+
        }
    }

    return 0;
}
```

(a) original program

```
int foo(int a [15][15],  int b [15][15]){
   int i, j;
   int c [30];
   for(i = 1; i <= 13; i += 1)
#pragma omp parallel for
   for(j = 1; j <= 13; j += 1)
      c[i+j] = a[i-1][j]+b[i][j]*a[i][j+1];
   return 0;
}
```

(b) transformed program

Figure 10: Example of parallel loop generation.

When running multi-objective evolutionary algorithms, adding transformations that do not change the complexities is an interesting thing. The new individuals formed with these transformations will be compared to other individuals whose complexities have been changed. They might even be preferred to individuals "more obfuscated" if they stay on a low-ranked Pareto front.

Transformation	**Quality**			**Impact on complexities**					
Name	**Potency**	**Resilience**	**Cost**	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6
Loop unrolling	high(μ_1)	trivial	free	high	low	none	low	none	low
Inlining	high(all)	strong one-way	free	high	medium	medium	medium	decreases	high
Localize declarations	medium(μ_6)	weak	free	none	none	none	none	none	medium
Remove comments	zero	one-way	free	none	none	none	none	none	none
Scramble variable names	zero	one-way	free	none	none	none	none	none	none
Outlining	medium(μ_5)	strong	free	none	none	decreases	none	medium	none
Index set splitting	low(μ_2)	weak	free	none	low	none	none	none	none
Unfolding	high(all)	full	free	high	medium	medium	medium	decreases	high
Flattening code	medium(μ_4)	strong	free	medium	none	none	medium	none	none
Loop fusion	zero	weak	free	none	none	none	none	none	none
Parallel loop	zero	trivial weak	free	none	none	none	none	none	none

Table 1: Selected transformations in PIPS and their relative impact on the chosen complexities.

3.2.12 Other transformations

The presented transformations feature the most common obfuscating transformations like function inlining or loop unrolling and new transformations should be added to get better results when running evolutionary algorithms. For example, transformations increasing the μ_3 complexity like inserting dead code or increasing the μ_4 complexity like replacing all variables by a global array of pointers to these variables would have high resilience and great potency for a free cost.

3.3 Shadobf

3.3.1 Workspace management in Pyps

PIPS passes are called through its Python front-end PYPS. The three main classes provided by PYPS are the `workspace` class, the `module` class and the `loop` class. The `workspace` class represents a program and the `module` class represents a function. The PIPS passes are methods of the `module` and `loop` classes (depending on whether the passes is applied on a loop or on a function).

PYPS and its modules provide several different `workspace` classes, each one providing new functionality. To combine the functionalities of several workspaces, the user has to define a new `workspace` class that will use multi inheritance to inherit the methods and attributes of its parents (see figure 11).

Special workspaces used in the project

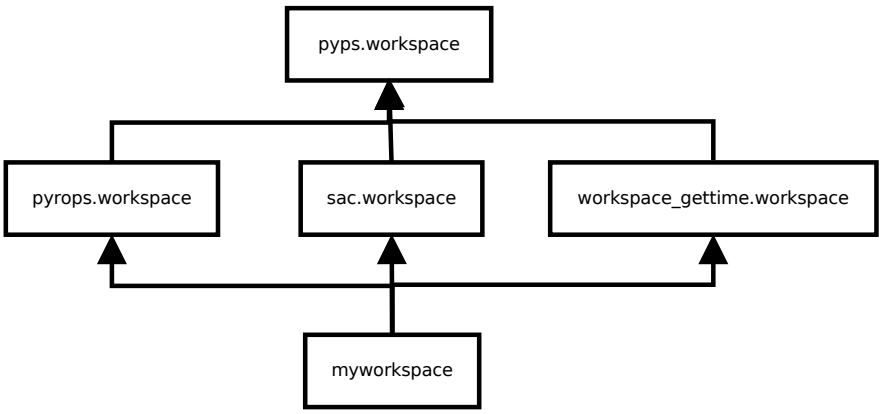


Figure 11: Exemple of workspace inheritance tree.

```
import pyps
import workspace_gettime
import pyrops

class workspace(workspace_gettime.workspace,pyrops.workspace):
    pass

w = workspace("source.c")
w.props.UNROLL_RATE=4
w.fun.main.loops()[0].unroll()
w.fun.main.benchmark_module() #Method inherited from workspace_gettime
w.display()
execution_times = w.benchmark(iterations=3)['main'] #Method inherited from workspace_gettime
print(execution_times)
w.save()
w.close()
```

Figure 12: Example of workspace manipulation.

The most used workspaces in this projects are `pyrops.workspace` and `workspace_gettime.workspace`.

PYROPS is a module of PYPS defining a workspace that uses PYRO 3[2][2]. PYRO stands for **PY**thon **R**emote **O**bject and is a python library implementing Remote Procedure Call for python 2. So far, using PYROPS workspaces is the only way to work on several workspaces at a time. In the `pyrops.workspace`, the usual attribute `cpypips` storing the python/c interface is replaced by a proxy to a RPC server.

Therefore, when calling a method or manipulating an attribute of a PYROPS workspace like with any regular workspace, the method is invoked or the attribute is accessed on the RPC server started on the workspace creation.

`Workspace_gettime` is a module of PYPS defining a workspace enabling the user to benchmark a program's modules execution times. The `workspace_gettime` workspace includes new functions and pragmas in the program's code in order to be able to measure the execution time of a module only.

An example of program benchmarking is seen on the example of figure 12.

3.3.2 The architecture of `pypsearch`

`Pypsearch` uses PYROPS and `workspace_gettime` workspaces in order to work with several workspace at a time and to be able to estimate their execution time. In `pypsearch`, each individual is modelled by a class

[2]Although the PYRO developers advise to switch to PYRO 4, PYRO 3 is still the version packaged in Debian, hence the older version

named `Individual` which is strongly linked to a workspace inheriting from `workspace_gettime.workspace` and `pyrops.workspace`.

The main attributes of the `Individual` class are `ws`: the workspace associated with each instance of the class which is opened in the class constructor, `mutation` : the list of the mutations that have been applied, `execution_time` : the mean execution time of the individual and `living` which indicates if the individual is still "alive" (i.e. if its workspace `ws` is still opened).

Mutations are applied to an individual by calling its `push` method and its `execution_time` is updated by calling the `rate` method.

A simplified view of the class Individual is shown on the figure 13

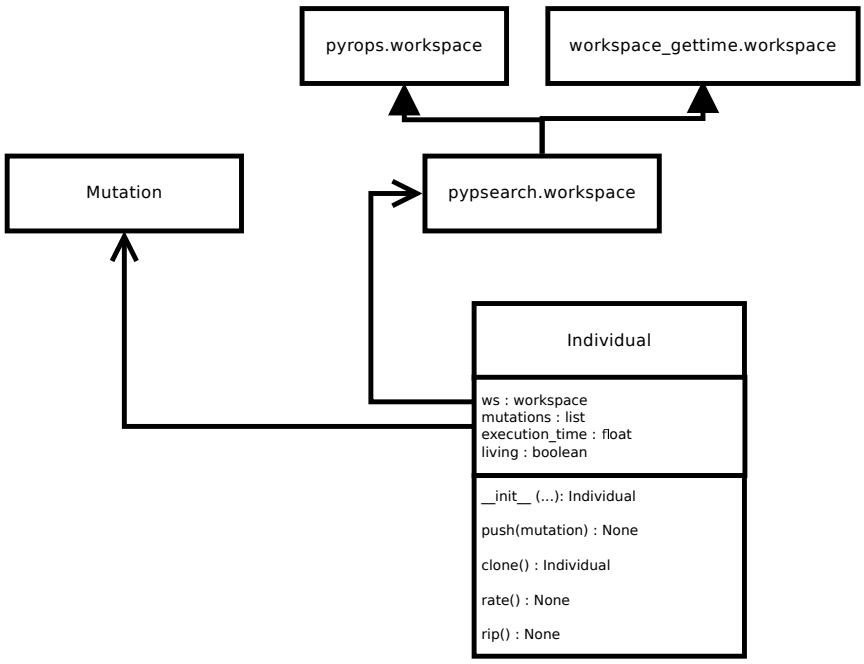


Figure 13: The `Individual` class in `pypsearch` (simplified view).

The genetic algorithm implemented in `pypsearch` is a classic simple criteria algorithm that does not allow identical individuals in a same generation. It is described in algorithm 3

3.3.3 The Shadobf engine

The SHADOBF project includes a python program that implements EAs and uses PYPS to run PIPS passes and manipulate programs. This Python program is divided in two pieces :

shadobf implements the EA used in experiences presented in this report.

pompe is a python program called from the command line to manipulate PIPS workspaces and pilot the obfuscation process on a given individual.

The figure 14 summarizes the architecture of the whole SHADOBF engine. The transformations and the obfuscation complexity passes are implemented in PIPS which is linked with PYPS using *swig*. PYPS access PIPS results by reading them in the PIPS database with special C functions linked to Python with swig.

PYPS classes and functions are imported in the `pompe` program witch opens workspaces, applies transformations, rates individuals and validate them. Let's recall that `pompe` is a standalone program that is called from the command line.

`Shadobf` is the program running the EAs, it uses the `subprocess` Python module to call `pompe` and reads the outputs of the subprocess to get the results.

We will now detail each of these components.

Algorithm 3: Genetic algorithm implemented in `pypsearch`.

```
inputs r = renewal rate of each generation
l = size of the population
n = number of generations to compute
output i = 1
while i < n do
    H = ∅
    //Building a new children population
    K = set of 2r samples of P_i
    sort(K)
    K = best r elements of K
    for each individual of K do
        M = {mutations available for individual}
        for each mutation of M do
            clone = individual.clone()
            clone.push(mutation)
            if clone is not in Q_i or in P_i then
                Q_i = Q_i ∪ {clone}
                breakă
    //Building the next generation
    for each individual of Q_i do
        individual.rate()
    P_{i+1} = best (n − i) elements of P_i
    H = best r elements of Q_i
    P_{i+1} = P_{i+1} ∪ H
```

3.3.4 `pompe` : the Pyps interface

`Pompe` is a command-line tool that uses PYPS workspaces to manipulate individuals. Its purpose is to separate the source code manipulations from the global evolution in a difference process and has been developed to answer three major problems met with PIPS.

- Precondition calculation are very costly for a computation time point of view
- Errors or warning would systematically kill the process using PYPS
- Opening and closing PIPS workspaces in a same process would make its memory consumption grow out of reasonable bounds (and eventually cause the process to be killed for `out of memory` reason)

After applying several transformations, the calculation of preconditions for a single individual could take more than a week. Therefore, we need to be able to stop this computation and continue the evolution.

When PIPS would meet an error or too many warnings while running, it would just stop running with a `sigabort` or `sigkill` message that cannot be caught, killing PYPS (or the program than called the PYPS methods) as a consequence. Therefore, we cannot have the EA and the workspace handling be done in the same process or the evolutionary run would be killed on the first degenerate individual.

In practice, when a PIPS workspace is closed, the memory it used is not completely freed. Opening and closing several PIPS workspaces in a same process would make its memory consumption grow without reducing and lead inevitably to being killed by the operating system Out-of-memory killer. Moving all the PIPS related operations in a new process is the better solution to theses three problems. When calling `pompe` through a subprocess, SHADOBF waits for the results and kills the subrocess if it has not finished its work in a fixed time length. If PIPS stops running because of an error, only the subprocess is killed and the evolutionary run can continue. Finally, a new subprocess being called for each individual manipulation the memory is freed when the process is finished.

From a practical point of view, `pompe` handles three main tasks : (1) generating the benchmark used to validate each produced individual, (2) rating (calculating the obfuscation complexities of) the individuals and (3) applying transformations on them.

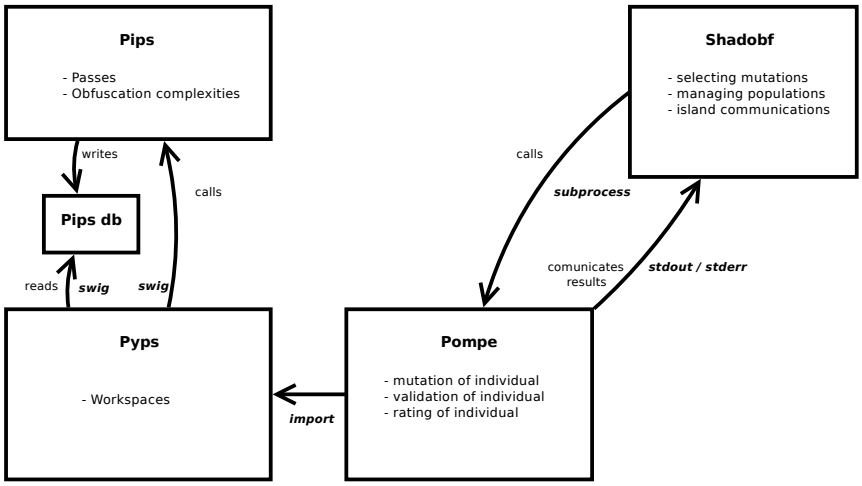


Figure 14: The SHADOBF architecture.

The validation of individuals is handled by a *Validation* class with the four following methods :

generate : generates a benchmark and storing references outputs and execution times (obtained by running the original individual on the generated benchmark).

validate : validates an individual by running it on the given benchmark and checking the output is the same as the reference.

printBench : print the benchmark, the output and execution time references to a text file

parseBench : reads the benchmark and the references from a text file

In the `Validation` class implemented in `pompe`, theses methods are empty. The user has to create his or her own Python program that will import `pompe` and will implement his or her own `Validation` class inheriting the four methods of the `Pompe.Validation` class.

Remark : If `pompe` is used with its empty `Validation` class, the program will still work but individuals will not be validated, therefore allowing degenerate individuals in the population

`Pompe` takes paths to the sources files of the individual it has to work on in addition to several parameters depending on the task to be completed and outputs paths to new individual generated (when the generation of a new individual is asked). One typical call to `pompe` could be :

```
pompe.py --sources matrix_mul_matrix.c --flags -D_BITS_BYTESWAP_H --fitness mu1 \\
--mutation "localizeDeclaration;main" --benchmark benchmark --work mutate
```

This example illustrates the following command line options of `pompe`.

–**sources matrix_mul_matrix.c** gives the path to source code of the individual to work on.

–**flags -D_BITS_BYTESWAP_H** is a `Cflag` that should be given to **gcc** when compiling the code .

–**fitness mu1** gives the fitness that must be rated.

–**mutation "localizeDeclaration;main"** indicates the transformation that must be applied and precises the module on which it must be applied.

–**benchmark benchmark** gives the path to the benchmark generated from the original individual ad printed in a text file.

–**work mutate** indicates that `pompe` must generate a new individual from the given sources

3.3.5 `Shadobf` : the evolution manager

SHADOBF implements EAs and handles evolutionary runs, letting `pompe` handle operations on individuals. The two main evolutionary algorithms implemented are : a simple single objective EA and the NSGA-II algorithm. Both of these algorithms can be used in a ring island model as defined in §3.3.5.

In addition of optimizing a given source code following the obfuscation complexities previously summarized, SHADOBF guarantees that the produced individuals are not degenerate. This means that the individuals produced by the evolutionary runs have a high probability to give the same output as the original individual for the same input and without taking too long time. To do so, a benchmark is generated, the original individual is compiled and this benchmark is run. Its output and execution time are then saved for reference. During the evolutionary run, every new individual is compiled and run against the benchmark. If the output is not strictly equal to the reference or if the execution time is too long (the factor being set by the user) compared to the execution time of the original individual then the new individual is declared invalid and a new random one is generated to take its place (see the algorithms 6 and 5).

The ring island model

The ring island model introduces parallelism in an EA by allowing several running EAs to communicate their best individuals to each other at some point of their computation. Each running instance of EA is an island and the communicated individuals are migrants. In `shadobf`, the islands send their whole population to the next island when a fixed generation is reached, leading to an asynchronous model which is more efficient from a computation point of view. When an island receives individuals from another island, the individuals are added to the current children population that is being built. They will eventually be integrated in the next generation if they can compete with the other individuals. A list of generation can be given to an island in order to make the island send its population when each one of these generations is reached.

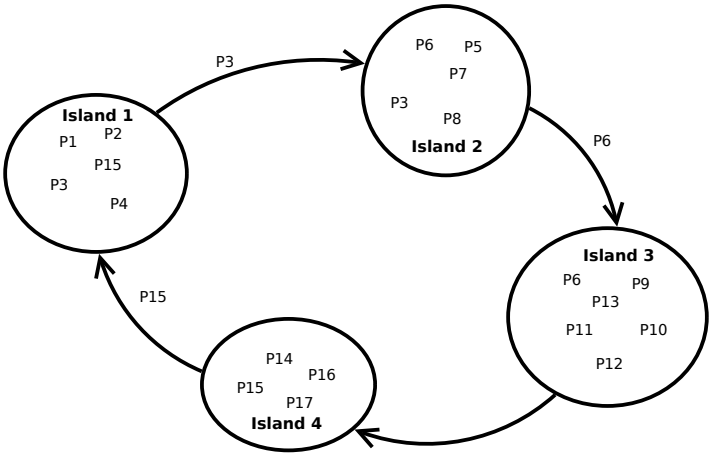


Figure 15: Example of ring island model featuring 4 islands.

The shadobf mainloop

The `shadobf` mainloop is the same regardless of the EA heuristic which is considered in the sens that the used EA will change the *sort* function : the single objective algorithm will sort the individuals of a population by their complexity using a regular sorting algorithm (since only one complexity is used). When running multi-objective mode, the NSGA-II algorithm will sort the individuals of a population using the *non domination sorting algorithm* detailed in §2.3.

Remark : when using the NSGA-II algorithm, the renewal rate has the same value as the parent population size

Algorithm 4: SHADOBF general evolutionary run.

```
G_1 = {originalindividual}
g = 1
generate_benchkmark(G_1[1])
rate(G_1[1])
generate first children population Q_1 sort(Q_1) // sort the population according to the EA in use
G_1 = G_1 ∪ Q_1
sort(G_1)
g = 2
while g <= max generation do
    Q_g = make_new_population(G_{g-1})
    sort(Q_g)
    G_g = G_{g-1}[0 : renewal_rate] ∪ Q_g[0 : renewal_rate]
    sort(G_g)
```

The first children population is composed of mutated copies of the original individual, each one being the result of a different mutation.

Algorithm 5: Creation the first children population Q_i from a parent individual I.

```
Q_1 = ∅
M = {mutations available for I }
shuffle M
for each mutation in M do
    i = evolve(I,mutation) // new individual, result of the application of mutation to I
    calulate obfuscation metrics for i
    if i is valid then
        Q_1 = Q_1 ∪ {i}
        if size(Q_1) = (population size -1) then
            break
```

The next children populations are composed of mutated copies of individuals of the parent populations. Notice that the parent population being assumed sorted, if the renewal rate is inferior to the parent population size then only the best individuals from the parent population will be copied and mutated. One could chose to shuffle the parent population to add even more randomness to the process of building the children population, but this will cost a new sorting of the parent population.

4 Validation on a concrete program

4.1 Matrix mul matrix

The presented results were obtained by running SHADOBF on a program multiplying two matrix together which source code is presented in listing D.1. This program was chosen as an example because its source code is easy to understand, making its more obfuscated versions more obvious to distinguish. Moreover this program contains many loops and most of the selected transformations only apply to loops.

The table 2 shows the obfuscation complexities for each modules of the `matrix_mul_matrix program`, namely : `matrix_mul_matrix`, `parse_matrix` and `main`.

The validation implemented for this program generates a benchmark of randomly generated matrices that are given to the original program to obtain the output and execution time references. The number of matrices, their dimensions and values are chosen randomly at the benchmark generation.

Algorithm 6: Creation of a children population Q_i from a parent population P_i.

```
Q_i = ∅
for each individual of P_i do
    M = {mutations available for individual}
    shuffle M
    for each mutation in M do
        i = evolve(individual,mutation) // new individual, result of the application of mutation to individual
        calulate obfuscation metrics for i
        if i is valid then
            Q_i = Q_i ∪ {i}
            if size(Q_i) = renewal rate then
                return Q_i
            break
```

Module	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6
matrix_mul_matrix	41	4	3	50	5	12
parse_matrix	83	5	4	30	2	21
main	89	2	1	71	30	24
TOTAL	213	11	8	151	65	57

Table 2: Obfuscation complexities of matrix_mul_matrix program.

4.2 Single objective runs

In this section, we will show some results obtained by running a simple objective EA on `matrix_mul_matrix`. For some complexities, the computation of the EA was to long and therefore we could not have the results in time to summarize them in this report at the moment of writting.

All theses runs were launched with the same parameters :

- The population size is 30 individuals
- The renewal rate is 15 individuals per generation.
- The limit of generations to compute is 30.
- The pool of transformations is the one described in §3.2.
- The used EA is the single objective version of the algorithm presented in §3.3.5.

4.2.1 μ_3 complexity

We present here a single objective run for the μ_3 complexity. Figure 16 shows the evolution of the μ_3 complexity through generations. The table 3 shows the obfuscation complexities of each modules of the obfuscated version of `matrix_mul_matrix` obtained at the end of the run.

The execution time of the obfuscated program produced by this evolutionary run was 388 s (the reference time being 643 s).

4.2.2 μ_5 complexity

We present here a single objective run for the μ_5 complexity. Figure 17 shows the evolution of the μ_5 complexity through generations. The table 4 shows the obfuscation complexities of each modules of the obfuscated version of `matrix_mul_matrix` obtained at the end of the run.

The execution time of the obfuscated program produced by this evolutionary run was 259 s (the reference time being 257 s).

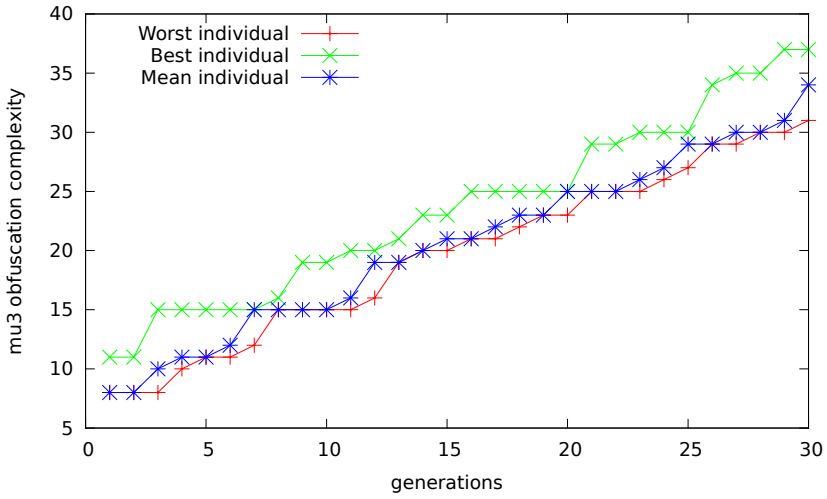


Figure 16: μ_3 complexity through generations.

Module	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6
matrix_mul_matrix	211	8	3	63	2	44
parse_matrix	90	5	4	31	1	21
main	449	10	3	229	117	24
GjWmbANbwh	31	3	2	13	5	8
ImRejokYGN	112	5	4	42	1	16
CWFBbnBrsZ	85	5	4	36	8	16
aUkajyWUdu	85	5	4	36	4	16
VuupXQdQca	85	5	4	36	8	16
YLHnwSgipR	85	5	4	36	4	16
MWrlmcgoRg	85	5	4	36	4	16
cXThqskROZ	33	2	1	11	7	4
TOTAL	1351	58	37	457	74	197

Table 3: Obfuscation complexities after μ_3 single objective run.

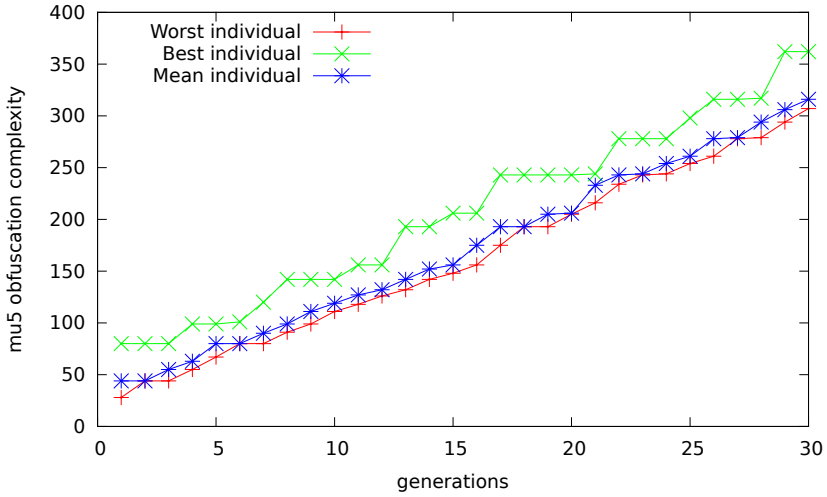


Figure 17: μ_5 complexity through generations.

Module	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6
matrix_mul_matrix	20	2	1	7	4	12
parse_matrix	39	1	0	4	2	21
main	327	1	0	2	35	178
gOjEiJbvnj	348	14	4	121	8	16
foyYVBeiUk	9	1	0	6	12	4
ebBjtXfsWi	21	2	1	11	7	4
rChpVULMVe	37	1	0	10	10	16
BmHOgcXNvH	177	5	4	53	6	28
ftzMiVIjtb	22	2	1	8	4	8
zOnOMKRqfG	9	1	0	6	16	4
dWKYWskAdD	40	1	0	9	4	16
cSojkLClcF	17	2	1	5	12	4
eJaeCRWoam	104	5	4	36	4	16
TOTAL	1170	38	16	284	124	327

Table 4: Obfuscation complexities after μ_5 single run.

4.3 Multiple objective runs

In this section we will show several results obtained by running different versions of MOEAs. We will begin by presenting Pareto fronts drawn for runs implying two complexities, then we will give the result obtained by running the NSGA-II algorithm with the six obfuscation complexities in addition to the execution time. Finally, we will show a run obtained applying the ring island model on the NSGA-II algorithm.

4.3.1 Pareto fronts

We present here Pareto fronts drawn for the optimization of two objective. More precisely, the combinations μ_1/μ_2, μ_4/μ_5 and μ_5/execution time are presented. These run happened to produce individuals having the exact same complexities (or execution time) very soon in the evolution process. Therefore, we drawn the Pareto fronts fro the generation before all the individual have the same complexities.

All theses runs were launched with the same parameters, *i.e.* :

- The population size is 30 individuals
- The renewal rate is 30 individuals per generation (imposed by the NSGA-II algorithm).
- The limit of generations to compute is 30.
- The pool of transformations is the one described in §3.2.
- The used EA is the NSGA-II algorithm.

μ_1 and μ_2

This run compares the sultanate evolution of the μ_1 and μ_2 complexities. We observed that on the 5th generation, nearly all the individuals had the same complexities. And on the 6th generation, all the individuals had the exact same complexities (*i.e.* $\mu_1 : 378$, $\mu_2 : 22$).

The figure 18 show the Pareto front for the 4th generation.

The obfuscated program obtained at the end of this run had the following complexities:

μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	execution time	time reference
378	22	11	371	25	57	272	260

μ_4 and μ_5

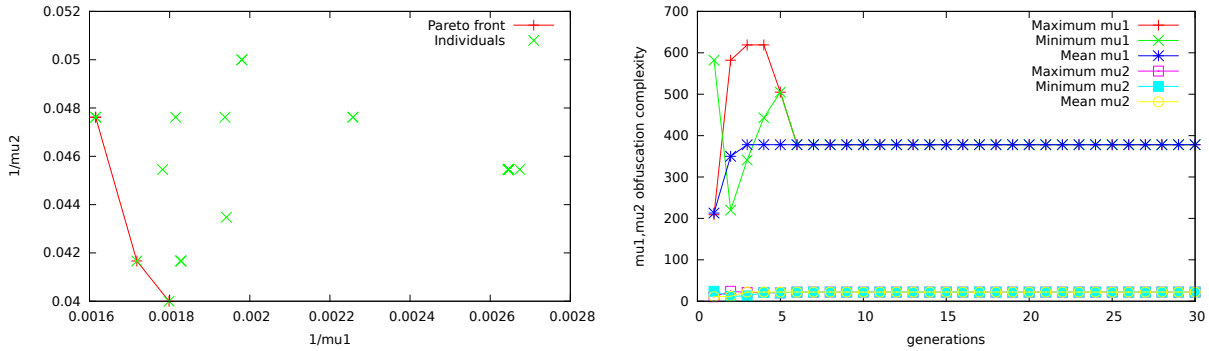


Figure 18: Pareto front for μ_1,μ_2 run with NSGA-II, 4th generation (left) and evolution of the complexities (right).

This run compares the sultanate evolution of the μ_4 and μ_5 complexities. We observed that on the 8th and the 9th generations, nearly all the individuals had the same complexities. And on the 10th generation, all the individuals had the exact same complexities (*i.e.* μ_4 : 223, μ_5 : 80).

The figure 19 show the Pareto front for the 7th generation.

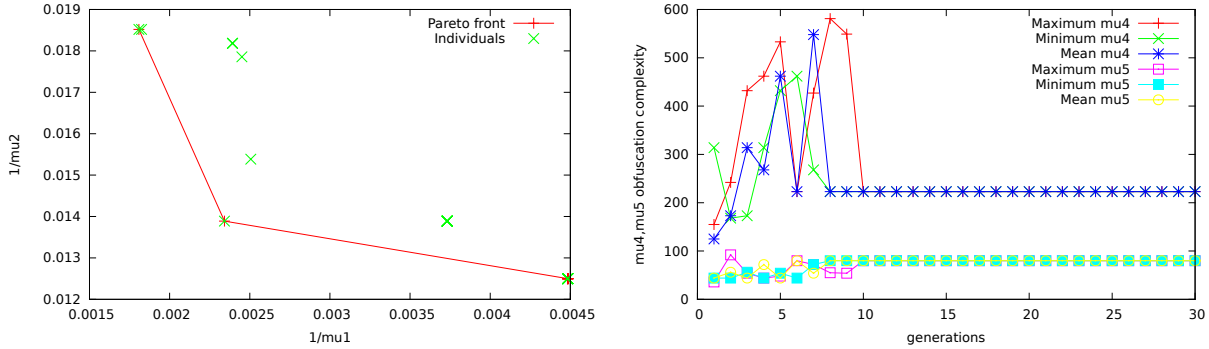


Figure 19: Pareto front for μ_4,μ_5 run with NSGA-II, 7th generation (left) and evolution of the complexities (right).

The obfuscated program obtained at the end of this run had the following complexities:

μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	**execution time**	**time reference**
284	14	8	223	80	69	971	1571

μ_5 and execution time

This run compares the sultanate evolution of the μ_5 complexity and the execution time. We observed that on the 2d and the 3rd generations, nearly all the individuals had the same complexities. And on the 4th generation, all the individuals had the exact same complexities (*i.e.* μ_5 : 44, execution time : 121s).

The figure 20 show the Pareto front for the 1st generation.

The obfuscated program obtained at the end of this run had the following complexities:

μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	**execution time**	**time reference**
246	13	8	211	44	53	121	120

4.3.2 NSGA-II run on the six obfuscation complexities

We show here the results obtained by running the NSGA-II algorithm for the six obfuscation complexities and the execution time.

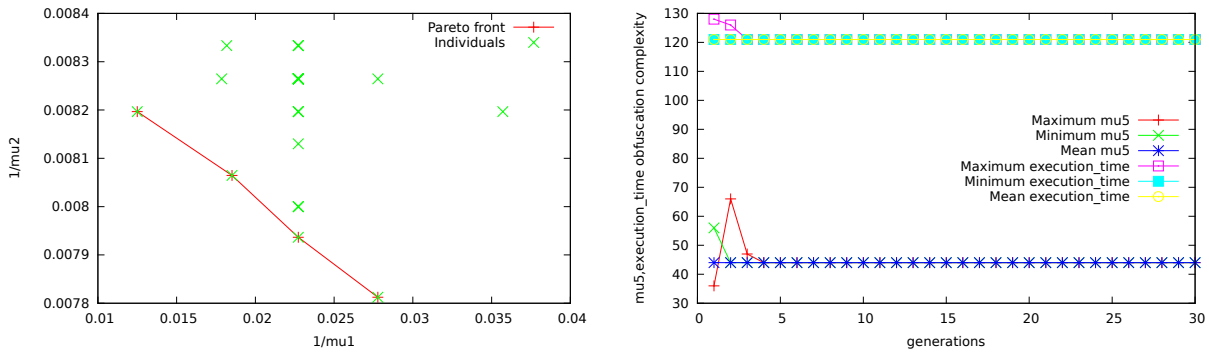


Figure 20: Pareto front for μ_5,execution time run with NSGA-II, 1st generation (left) and evolution of the complexities (right).

The μ_6 complexity was compute without calling the `approximative_eval` pass. Therefore, no upper bound was calculated for variable sized variables. This choice has been made to reduce the time necessary to compute the evolutionary run by removing the need to compute the preconditions of the modules before computing their μ_6 complexity.

The figure 21 show the evolution of the objective values throughout the generations. The table 5 shows the transformations that were applied on `matrix_mul_matrix` to obtain the best obfuscated program from it.

- The population size is 30 individuals
- The renewal rate is 30 individuals per generation (imposed by the NSGA-II algorithm).
- The limit of generations to compute is 30.
- The pool of transformations is the one described in §3.2.
- The used EA is the NSGA-II algorithm.
- The computing of the μ_6 complexity was done without calling the `approximative_eval` pass.

Transformation	**Parameters**	**Target**
Loop unrolling	rate = 4	l99997
Index set splitting	bound = 18	l99986
Loop unrolling	rate = 4	l99998
Index set splitting	bound = 18	l99995
Outlining	none	loop l99996
Loop unrolling	rate = 4	l99999
Parallel loop making	none	matrix_mul_matrix
Outlining	none	loop l99974
Outlining	none	loop l99996
Index set splitting	bound = 18	l99988
Loop unrolling	rate = 4	l99979
Outlining	none	loop l99980
Index set splitting	bound = 18	l99977
Inlining	inline : parse_matrix	main
Outlining	none	loop l99973
Loop unrolling	rate = 4	l99996
Removing comments	none	bzanWsdQPT

Table 5: Transformations applied on NSGA-II run (ordered).

The obfuscated program obtained at the end of this run had the following complexities values :

μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	**execution time**	**time reference**
1400	41	18	2161	108	257	634	631

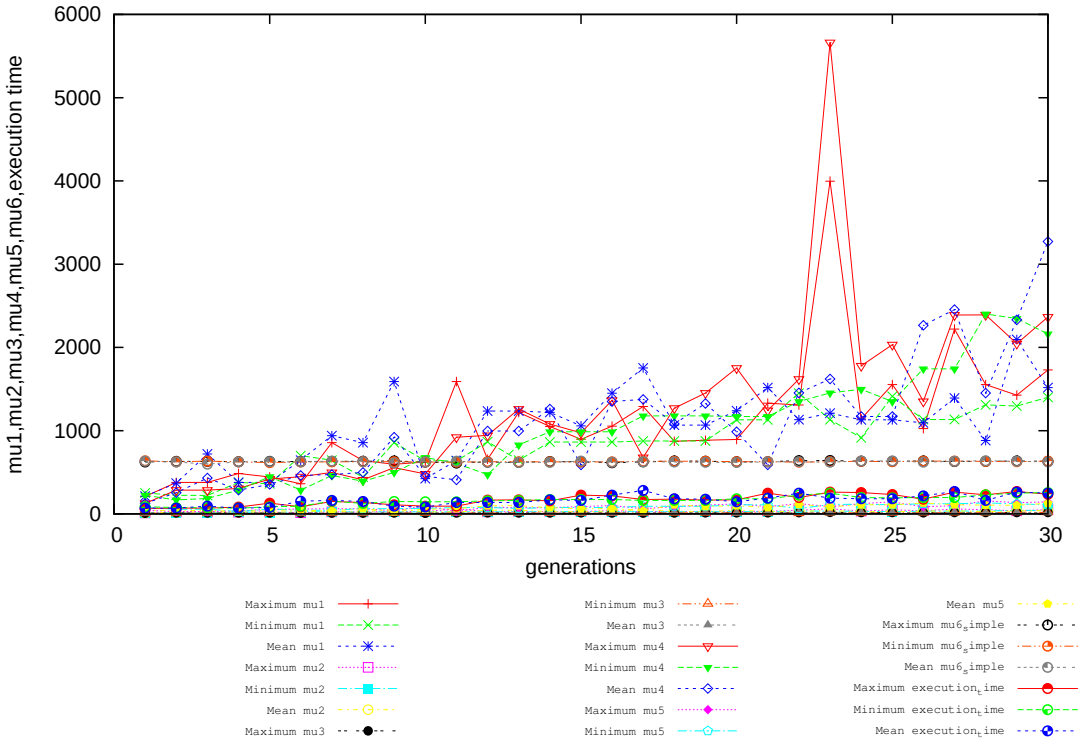


Figure 21: NSGA-II run on `matrix_mul_matrix` with the μ_1, μ_2, μ_3, μ_4, μ_5, μ_6 and execution time.

4.3.3 Island ring model

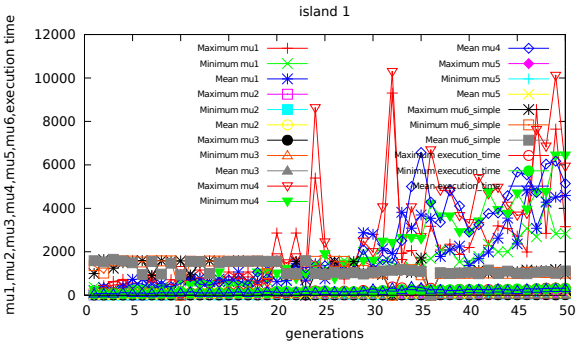
We present here the results obtained by applying the ring island model to the NSGA-II algorithm. We used 16 islands to optimize the six obfuscation complexities and the execution time of `matrix_mul_matrix`.

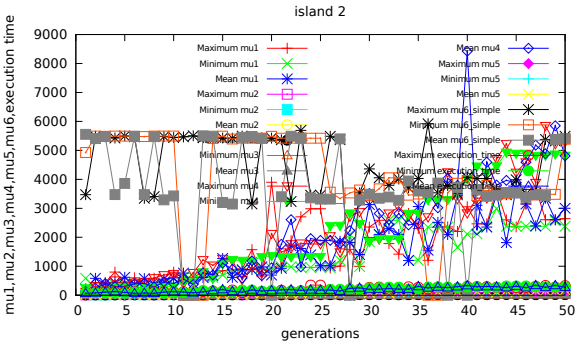
When the computation of the evolutionary run was finished, we used the sorting techniques used in the NSGA-II algorithm to find the best obfuscated program among the best individuals generated by each island. We then found that the island 2 had produced the best obfuscated version of `matrix_mul_matrix`. When the computation was over, the island 8 and the island 14 had stopped. The island 8 was stopped because its computation was still not finished when the time allocated on cluster for this run was over and the island 14 stopped because it produced too many degenerate individuals and had not enough valid individuals to continue its computation.

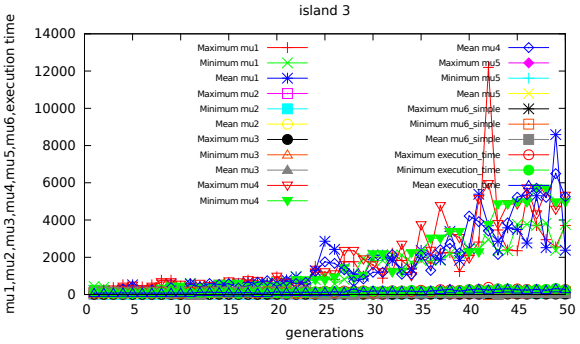
This run was configured as follows :

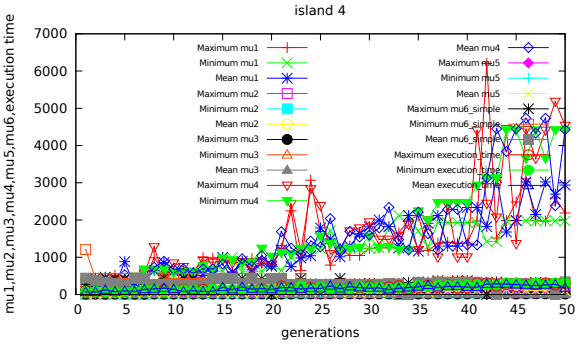
- The population size is 30 individuals
- The renewal rate is 30 individuals per generation (imposed by the NSGA-II algorithm).
- The limit of generations to compute is 50.
- The pool of transformations is the one described in §3.2.
- The used EA is the NSGA-II algorithm cobined with the ring island model.
- 16 islands were running at the same time.
- The 16 Islands sent their population to their neighbors on the 10th,20th,30th and 40th generations.
- The island n sends its populations to the island $n+1 \mod 16$ and receives the populations sent by the island $n-1 \mod 16$.
- The computing of the μ_6 complexity was done without calling the `approximative_eval` pass.

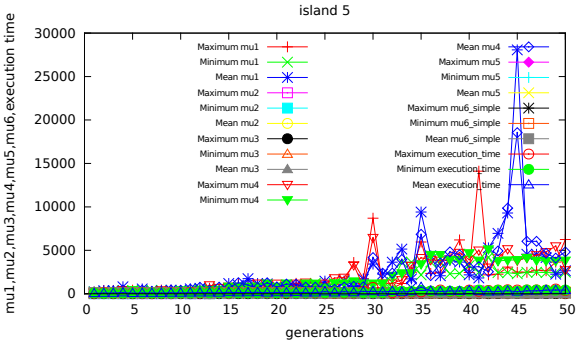
The following graphics show the evolutions of the complexities values and of the execution time of the individuals throughout the generations for each island. The table 7 summarizes the complexities values and the execution time of the best individuals produced by each island.

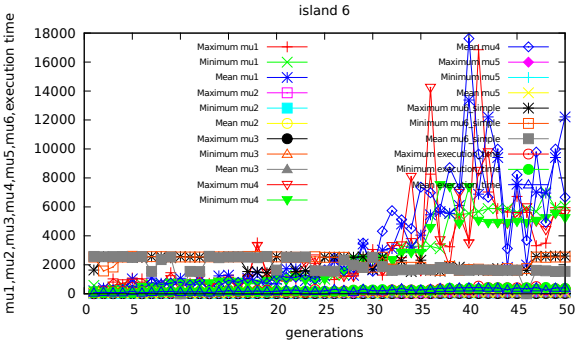


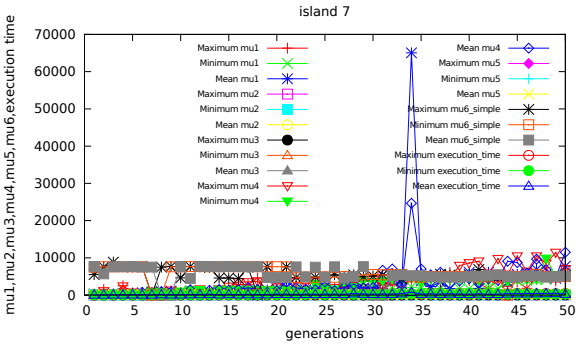


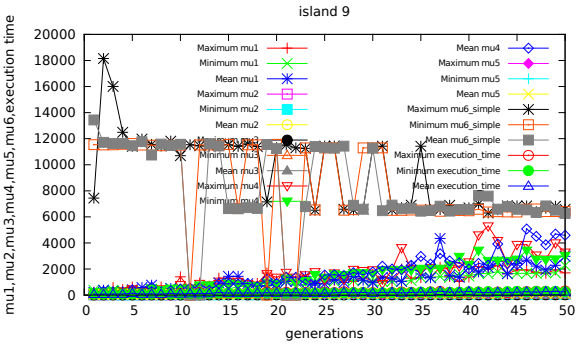


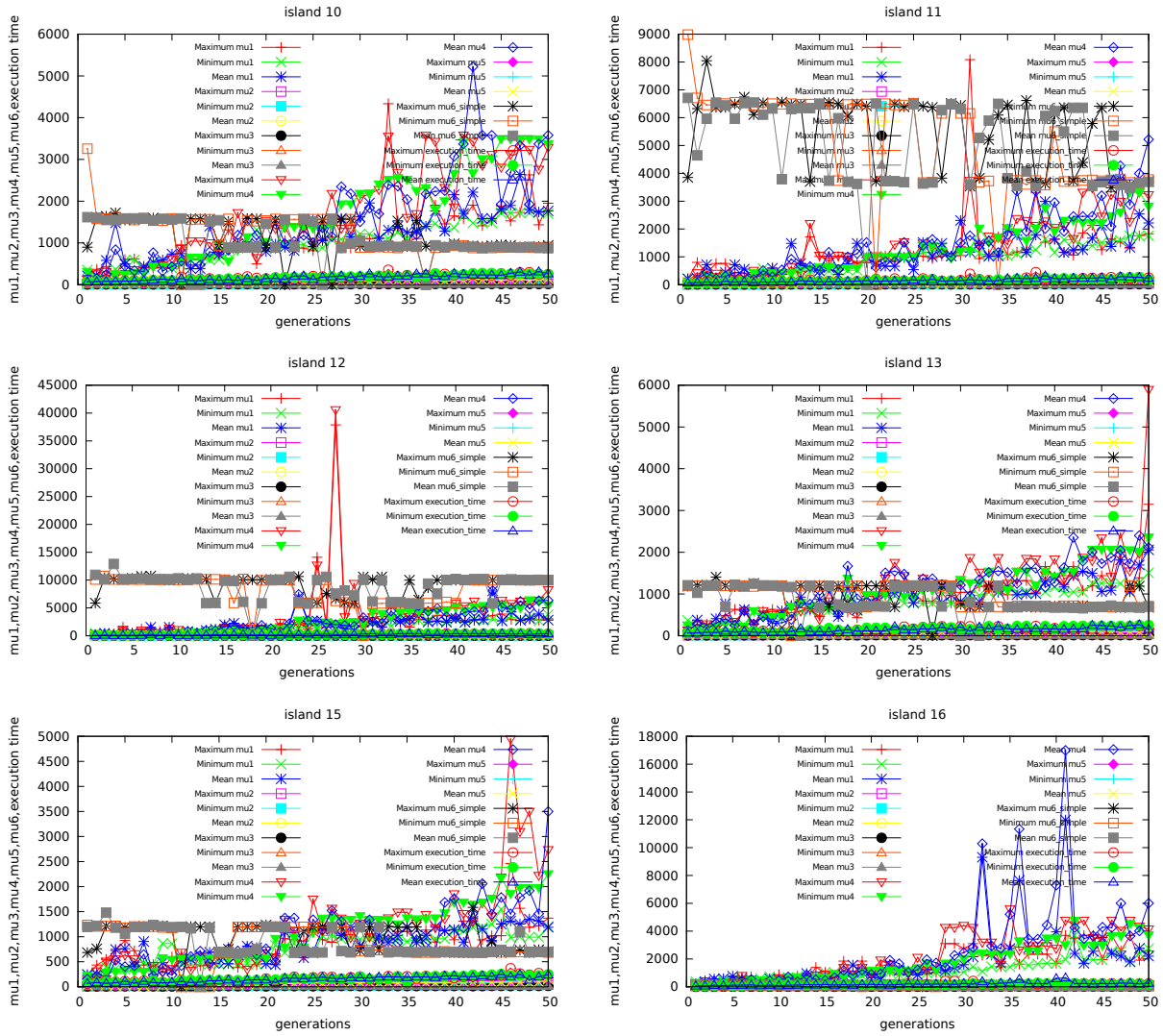












Island	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	execution time	time reference
Island 1	2831	65	26	6439	176	317	1018	1576
Island 2	2379	74	22	4821	146	289	5385	5489
Island 3	3768	81	20	4960	136	249	127	124
Island 4	1976	62	18	4416	135	313	291	435
Island 5	2485	85	19	3837	122	437	98	94
Island 6	6055	75	22	5232	113	369	2533	2551
Island 7	5458	61	23	5297	121	341	4948	7648
Island 9	2276	64	15	3033	104	261	6420	125335
Island 10	1706	47	15	3369	102	249	905	1667
Island 11	1760	50	19	2816	166	185	3777	6480
Island 12	3782	65	19	5502	136	361	10006	10099
Island 13	1507	44	14	2360	106	257	707	1194
Island 15	1205	41	17	2254	139	249	687	1199
Island 16	2740	76	20	3720	125	205	74	126

Table 7: Best individuals in the 16 islands of the ring island model.

The Island 2 has produced the best obfuscated version of `matrix_mul_matrix`. The source code of this program D.2 can be seen in the appendices.

4.4 Summary of the obtained results

We summarize the previous results in the table 8. For each complexity, the cell containing the highest value is colored in green. For the time execution, the cell containing the best ration execution time / time reference is colored in green.

	Task	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	execution time	time reference
S.O.	single μ_3	1351	58	37	457	74	197	388	643
	single μ_5	1170	38	16	284	124	327	259	257
M.O.	μ_1,μ_2	378	22	11	371	25	57	272	260
	μ_4,μ_5	284	14	8	223	80	69	971	1571
	μ_5,execution time	246	13	8	211	44	53	121	120
	every complexity	1400	41	18	2161	108	257	634	631
	ring island model	2379	74	22	4821	146	289	5385	5489

Table 8: Recapitulation of the presented results (**S.O.** : Single Objective, **M.O.** : Multiple Objective). The cells in green highlight the best found results.

The ring island model applied to the NSGA-II algorithm brought the best obfuscated program without augmenting the execution time of the program on the generated benchmark. We can see that optimizing the six obfuscation complexities altogether brought a better obfuscated program than when optimizing only two complexities. Moreover, using the six complexities kept a diversity of complexity values among the individuals throughout the generations whereas the individuals tend quickly to have the same complexity when using only two complexities.

5 Problems met, and choices made

5.1 pypsearch **and** pompe

At first, the Shadobf was supposed to be an enhancement of `pypsearch`. But unfortunately too many problems were encountered with Pips and Pyro.

Pyro's RPC servers tended to keep many unused files open. That caused many `file descriptors out of range` errors (this error happened on Debian systems, whereas on ArchLinux systems, the error was `Too many opend files`, no matter the value of `ulimit -n`) that were fatal for the execution of `pypsearch`. Pyps not being able to open several workspaces at the same time, the only way to work on more than one individual at a time was to use `pyrops` workspaces, and meet the problems mentioned above. Moreover, when Pips would encounter an error, it would most of the time send a `sigabort` or `sigkill` message that would kill `pypsearch`.

`Pypsearch` and Pyrops being unmaintained for three years at least and Pyro 3 being deprecated by its developers, we decided to make a new scalable program that would be more resistant to degenerate individuals and to Pips errors.

This program was named `shadobf` after the name of the project and used some parts of `pypsearch` for managing the transformations. Its implementation is sequential in the sense that it works on individuals one by one. But using a sequential implementation in order to remove the Pyro dependencies was not enough because the errors caused by Pips would still kill `shadobf`. Therefore, we decided to divide the program in two parts : `shadobf` and `pompe`.

`Shadobf` kept the high level management of the EAs (*i.e.* management of populations, of the pool of mutations ...) and `pompe` would handle all the operation that require Pips (*i.e.* evaluation of obfuscation complexities, benchmarking, application of transformations ...). `Shadobf` would call `pompe` from the command line through the `subprocess` python module.

5.2 The passes left behind

Some passes were implemented in Pips but unfortunately we were not able to make them stable in time to use them in our transformation pool.

The pass `scramble_variable_names` aimed to change the name of every variable a name generated at random. Implementing this pass was supposed to lead to a way of rename the functions of the given

program. This pass deletes every variable declaration and adds new one, declaring the variables with their new name. This step requires the pass `split_variable_declarations` to separate every variable declaration in the AST (since PIPS has trouble replacing multiple variables that are declared in a same statement.

So far, the `split_variable_declarations` cannot handle the splitting of global variable. It splits the declarations and place them in a block, transforming them into local variable. Therefore the generated code cannot be compiled as it contains variables referenced with being declared.

The `scramble_variable_names`, although providing a high resilient obfuscating transformation, does not increase any of the six obfuscation metrics we used. Therefore, we decided to keep the development of the `scramble_variable_names`, `split_variable_declarations` pass, along with the pass changing functions names for a future work.

5.3 Pips and Clang

At the beginning of this project, we had to chose a tool for the static analysis and the transformation of source codes. We chose PIPS over CLANG because of my past experience with PIPS and because PIPS already brought several transformations. Moreover, SHADOBF was supposed to be an enhancement of `pypsearch` which is brought by PIPS as a PYPS module.

6 Future work

This reports details a work that is currently still in progress, a final version of this report will be distributed at the end of the internship, including some results and short term objective that would have been reached by then. These results will be presented at the final presentation of this internship in September.

Also, it is planned to submit a paper presenting our work to the French conference *ComPAS'13*.

6.1 Short term objectives

In §4, only one program was used to validate our work, a short term objective could be to chose other programs with different properties to obfuscate with SHADOBF.

So far, only the ring island model has been implemented in SHADOBF. A less deterministic island model where island could send their populations to any other island could be implemented. This could increase the diversity among the populations of islands.

6.2 Long term objectives

As seen in the results presented in in §3.2, the transformations we selected for our evolutionary runs is essentially composed of transformations that were already implemented in PIPS. Implementing new obfuscating transformations in PIPS would enable us to enlarge our pool of transformations and introduce more diversity among the individuals. Moreover, a larger pool of transformation would enable us to use heterogeneous islands when running an island model (*i.e.* each island would have a different pool of transformations) and justify extending the number of generations to be computed in the evolutionary runs.

The following transformations could be implemented in a long term :

- including dead code (increases all the complexities except the μ_5 complexity)
- turning simple predicates into opaque predicates (a highly resilient transformation that could increase the μ_1 complexity)
- replacing all variables by a single global array of pointers, each pointer being treated as pointer to one of the variable that was removed (a one-way resilient transformation that would increase the μ_4 complexity)

7 Conclusion

We showed how combining MOEAs with a source-to-source compiler can optimize the obfuscation of a given program. As for optimizing the execution time of programs, EAs can search the space of available transformations automatically and in a more efficient way.

The validation seen in §4 showed that optimizing the values of the six obfuscations complexities at the same time brings a better obfuscated program than focusing on one or two of theses complexities. Moreover, a higher diversity of individuals is observed throughout the generations : during the evolutionary runs optimizing the values of only two obfuscation complexities, the individuals tended to reach the same complexities in less than ten generations whereas a diversity of values was kept among individual when optimizing the six obfuscation complexities.

Adding the execution time as an objective in the multiple objective evolutionary runs helped keeping a tolerable execution time for the obfuscated program. However, it should be noticed that all the transformations implied in our evolutionary runs had at most a cheap cost.

Enlarging the pool of transformations would surely bring better obfuscated program depending of the quality of the added transformations. A higher number of available transformations would increase the diversity of individuals met throughout the generations of the evolutionary runs. Transformations more "obfuscation-oriented" would have a greater impact on the obfuscation complexities than most of the Pips transformations we used. Including dead code with good opaque predicates would help increasing almost all the obfuscation complexity we considered.

As the evaluation of the potency of an obfuscating transformation requires the choice of an obfuscation metric, obfuscating transformations are often specialized in increasing a few complexities (the number of transformation increasing all the complexity being small). Even though some synergies between two or more obfuscation complexities, using single objective EAs favorites some transformation without guaranteeing the optimality of the obfuscated program according to all the obfuscation metrics. Using MOEAs, an optimally obfuscated program according all the complexities can be found without privileging any complexity. Moreover, the increasing of its execution time can be controlled by adding this criteria in the fitness of the MOEA.

References

[1] Parallélisation interprocédurale de programmes scientifiques (pips). http://pips4u.org.

[2] Python remote object. http://irmen.home.xs4all.nl/pyro3/.

[3] Simplified wrapper and interface generator. http://www.swig.org/.

[4] Clark Thomborson Christian Collberg and Douglas Low. A taxonomy of obfuscating transformations. 1997.

[5] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast elitist multiobjective genetic algorithm : Nsga-ii. 2002.

[6] Serge Guelton and Sébastien Varrette. Une approche génétique et source à source de l'optimisation de code. 2009.

[7] Warren A. Harrison and Keneth I. Magel. A complexity measure based on nesting level.

[8] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. 1981.

[9] Thomas McCabe. A complexity measure. 1976.

[10] Günter Rudolph. Convergence of evolutionary algorithms in general search spaces. 1996.

A Acronym used

AST Abstract Syntax Tree
CFG Control Flow Graph
CSC Computer Science and Communications
dEA distributed Evolutionary Algorithm
EA Evolutionary Algorithm
EAs Evolutionary Algorithms
MOEA Multi Objective Evolutionary Algorithm
MOEAs Multi Objective Evolutionary Algorithms
GA Genetic Algorithm
GP Genetic Programming
UL University of Luxembourg

B Notations

Do a table of notations
Ex:

- P = Program
- P'= Program obfuscate
- $\tau_{pot}(P)$ = potency of the transformation from P to P'
- $\tau_{res}(P)$ = resilience of the transformation from P to P'
- $\tau_{cost}(P)$ = cost of the transformation from P to P'
- $\tau_{qual}(P)$ = quality of the transformation from P to P'
- E(P) = the complexity of P,

C Contributions statistics during this internship

Task	Lines of code	Commits
Pips	2297	29
Pompe	804	1
Shadobf	707	4

Table 9: Quantified contributions in Pips, Shadobf and Pompe.

D Code listings

D.1 The original code of matrix_mul_matrix

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void matrix_mul_matrix(size_t N, double C[4*N][4*N], double A[4*N][4*N], double B[4*N][4*N]) {
    size_t i,j,k;
    for (i=0; i<4*N; i++) {
        for (j=0; j<4*N; j++) {
            for(k=0;k<4*N;k++)
```

```
            {
                C[i][j]+=A[i][k] * B[k][j];
            }
        }
    }
}

void parse_matrix(int n, char * s, double a[4*n][4*n]){
    int sl = strlen(s);
    char buffer[687];
    int i=-1;
    int j = 0;
    int p = 0;
    int k = 0;
    for (k =0;k<sl;k++){
        if (s[k] == '['){
            j = 0;
            i++;
        }
        else if (s[k] == ' '){
            buffer[p] = '\0';
            a[i][j] = atof(buffer);
            p =0;
            j++;
        }
        else if (s[k] == ']'){
            buffer[p] = '\0';
            a[i][j] = atof(buffer);
            p =0;

        }else{
            buffer[p] = s[k];
            p++;
        }
    }
}

int main(int argc, char ** argv) {
    int n = atoi(argv[1]);
    double (*a)[4*n][4*n],(*b)[4*n][4*n],(*c)[4*n][4*n];
    a=malloc(sizeof(double)*16*n*n);
    b=malloc(sizeof(double)*16*n*n);
    c=calloc(sizeof(double),16*n*n);
    int i=0;
    int j = 0;

    parse_matrix(n,argv[2],(*a));
    parse_matrix(n,argv[3],(*b));

    matrix_mul_matrix(n,*c,*a,*b);

    for(i=0;i<4*n;i++)
        printf("%f ",(*c)[i][i]);
    free(a);
    free(b);
    free(c);
    return 0;
}
```

D.2 Obfuscated source code of matrix_mul_matrix produced by combination of the ring island model and NSGA-II

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void yBsjWVaOsv(size_t N, long int LU_IND0, long int LU_IND2, double A[4*N][4*N], double B[4*N][4*N], double
    C[4*N][4*N]);
```

```
6  void fUjmZFhOPb(size_t N, long int LU_IND0, long int LU_IND2, double A[4*N][4*N], double B[4*N][4*N], double
       C[4*N][4*N]);
   void RwZXjQnHhy(int n0, char buffer[687], int i_0[1], int p_0[1], int sl, double P_0[4*n0][4*n0], char *s0);
   void PtIXJLXyOh(size_t N, size_t j1, double A[4*N][4*N], double B[4*N][4*N], double C[4*N][4*N], long int
       LU_IND0);
9  void wYZYlYYhNv(size_t N, long int LU_IND0, double A[4*N][4*N], double B[4*N][4*N], double C[4*N][4*N]);
   void WwZLXIuStG(size_t N, long int LU_IND0, double A[4*N][4*N], double B[4*N][4*N], double C[4*N][4*N]);
   void matrix_mul_matrix(size_t N, double C[4*N][4*N], double A[4*N][4*N], double B[4*N][4*N]);
12 void parse_matrix(int n, char *s, double a[4*n][4*n]);
   int main(int argc, char **argv);
   void KmVcvdqGyh(size_t N, long int LU_IND0, long int LU_IND2, double A[4*N][4*N], double B[4*N][4*N],
       double C[4*N][4*N])
15 {
     //PIPS generated variable
     size_t k03;
18 l99989:
     for(k03 = 0; k03 < 4*N; k03 += 1)
       C[LU_IND0*1+0][(LU_IND2+1)*1+0] += A[LU_IND0*1+0][k03]*B[k03][(LU_IND2+1)*1+0];
21 }
   void yBsjWVaOsv(size_t N, long int LU_IND0, long int LU_IND2, double A[4*N][4*N], double B[4*N][4*N], double
       C[4*N][4*N])
   {
24   //PIPS generated variable
     size_t k01;
   l99986:
27   for(k01 = 0; k01 < 4*N; k01 += 1)
       C[LU_IND0*1+0][(LU_IND2+3)*1+0] += A[LU_IND0*1+0][k01]*B[k01][(LU_IND2+3)*1+0];
   }
30 void fUjmZFhOPb(size_t N, long int LU_IND0, long int LU_IND2, double A[4*N][4*N], double B[4*N][4*N], double
       C[4*N][4*N])
   {
     //PIPS generated variable
33   size_t k00;
   l99999:
     for(k00 = 0; k00 < 4*N; k00 += 1)
36     C[LU_IND0*1+0][LU_IND2*1+0] += A[LU_IND0*1+0][k00]*B[k00][LU_IND2*1+0];
   }
   void RwZXjQnHhy(int n0, char buffer[687], int i_0[1], int p_0[1], int sl, double P_0[4*n0][4*n0], char *s0)
39 {
     //PIPS generated variable
     int j, k, i, p;
42   //PIPS generated variable
     int LU_NUB0, LU_IB0, LU_IND0;
     p = *p_0;
45   i = *i_0;
   l99995:   LU_NUB0 = (sl-1-0+1)/1;
     LU_IB0 = MOD(LU_NUB0, 4);
48 l99999:
     for(LU_IND0 = 0; LU_IND0 <= LU_IB0-1; LU_IND0 += 1) {
       if (s0[LU_IND0*1+0]=='[') {
51       j = 0;
         i++;
       }
54     else if (s0[LU_IND0*1+0]==' ') {
         buffer[p] = '\0';
         P_0[i][j] = atof(buffer);
57       p = 0;
         j++;
       }
60     else if (s0[LU_IND0*1+0]==']') {
         buffer[p] = '\0';
         P_0[i][j] = atof(buffer);
63       p = 0;
       }
       else {
66       buffer[p] = s0[LU_IND0*1+0];
         p++;
       }
69 l99977:       ;
     }
```

```
l99998:
72    for(LU_IND0 = LU_IB0; LU_IND0 <= LU_NUB0-1; LU_IND0 += 4) {
        if (s0[(LU_IND0+0)*1+0]=='[') {
          j = 0;
75        i++;
        }
        else if (s0[(LU_IND0+0)*1+0]==' ') {
78        buffer[p] = '\0';
          P_0[i][j] = atof(buffer);
          p = 0;
81        j++;
        }
        else if (s0[(LU_IND0+0)*1+0]==']') {
84        buffer[p] = '\0';
          P_0[i][j] = atof(buffer);
          p = 0;
87      }
        else {
          buffer[p] = s0[(LU_IND0+0)*1+0];
90        p++;
        }
        if (s0[(LU_IND0+1)*1+0]=='[') {
93        j = 0;
          i++;
        }
96      else if (s0[(LU_IND0+1)*1+0]==' ') {
          buffer[p] = '\0';
          P_0[i][j] = atof(buffer);
99        p = 0;
          j++;
        }
102     else if (s0[(LU_IND0+1)*1+0]==']') {
          buffer[p] = '\0';
          P_0[i][j] = atof(buffer);
105       p = 0;
        }
        else {
108       buffer[p] = s0[(LU_IND0+1)*1+0];
          p++;
        }
111     if (s0[(LU_IND0+2)*1+0]=='[') {
          j = 0;
          i++;
114     }
        else if (s0[(LU_IND0+2)*1+0]==' ') {
          buffer[p] = '\0';
117       P_0[i][j] = atof(buffer);
          p = 0;
          j++;
120     }
        else if (s0[(LU_IND0+2)*1+0]==']') {
          buffer[p] = '\0';
123       P_0[i][j] = atof(buffer);
          p = 0;
        }
126     else {
          buffer[p] = s0[(LU_IND0+2)*1+0];
          p++;
129     }
        if (s0[(LU_IND0+3)*1+0]=='[') {
          j = 0;
132       i++;
        }
        else if (s0[(LU_IND0+3)*1+0]==' ') {
135       buffer[p] = '\0';
          P_0[i][j] = atof(buffer);
          p = 0;
138       j++;
        }
        else if (s0[(LU_IND0+3)*1+0]==']') {
```

```
141             buffer[p] = '\0';
                P_0[i][j] = atof(buffer);
                p = 0;
144         }
            else {
                buffer[p] = s0[(LU_IND0+3)*1+0];
147             p++;
            }
    l99976:     ;
150     }
        k = 0+MAX0(LU_NUB0, 0)*1;
        *i_0 = i;
153     *p_0 = p;
    }
    void PtIXJLXyOh(size_t N, size_t j1, double A[4*N][4*N], double B[4*N][4*N], double C[4*N][4*N], long int
        LU_IND0)
156 {
        //PIPS generated variable
        size_t k1;
159     l99987:
        for(k1 = 0; k1 < 4*N; k1 += 1)
            C[(LU_IND0+3)*1+0][j1] += A[(LU_IND0+3)*1+0][k1]*B[k1][j1];
162 }
    void wYZYlYYhNv(size_t N, long int LU_IND0, double A[4*N][4*N], double B[4*N][4*N], double C[4*N][4*N])
    {
165     //PIPS generated variable
        size_t j2;
        //PIPS generated variable
168     long int LU_NUB0, LU_IB0, LU_IND1;
        //PIPS generated variable
        size_t k20, k21, k22, k23, k24;
171 l99988:    LU_NUB0 = (4*N-1-0+1)/1;
        LU_IB0 = MOD(LU_NUB0, 4);
    l99996:
174     for(LU_IND1 = 0; LU_IND1 <= LU_IB0-1; LU_IND1 += 1) {
            //PIPS generated variable

177     l99997:
            for(k20 = 0; k20 < 4*N; k20 += 1)
                C[(LU_IND0+2)*1+0][LU_IND1*1+0] += A[(LU_IND0+2)*1+0][k20]*B[k20][LU_IND1*1+0];
180     l99999:     ;
        }
    l99991:
183     for(LU_IND1 = LU_IB0; LU_IND1 <= LU_NUB0-1; LU_IND1 += 4) {
            //PIPS generated variable

186     l99995:
            for(k24 = 0; k24 < 4*N; k24 += 1)
                C[(LU_IND0+2)*1+0][(LU_IND1+0)*1+0] += A[(LU_IND0+2)*1+0][k24]*B[k24][(LU_IND1+0)*1+0];
189
    l99994:
            for(k23 = 0; k23 < 4*N; k23 += 1)
192             C[(LU_IND0+2)*1+0][(LU_IND1+1)*1+0] += A[(LU_IND0+2)*1+0][k23]*B[k23][(LU_IND1+1)*1+0];

    l99993:
195         for(k22 = 0; k22 < 4*N; k22 += 1)
                C[(LU_IND0+2)*1+0][(LU_IND1+2)*1+0] += A[(LU_IND0+2)*1+0][k22]*B[k22][(LU_IND1+2)*1+0];

198 l99992:
            for(k21 = 0; k21 < 4*N; k21 += 1)
                C[(LU_IND0+2)*1+0][(LU_IND1+3)*1+0] += A[(LU_IND0+2)*1+0][k21]*B[k21][(LU_IND1+3)*1+0];
201 l99998:     ;
        }
        j2 = 0+MAX0(LU_NUB0, 0)*1;
204 }
    void WwZLXIuStG(size_t N, long int LU_IND0, double A[4*N][4*N], double B[4*N][4*N], double C[4*N][4*N])
    {
207     size_t j1, k1;
    l99986:
        for(j1 = 0; j1 < 4*N; j1 += 1)
```

```
l99987:       PtIXJLXyOh(N, j1, A, B, C, LU_IND0);
}
void matrix_mul_matrix(size_t N, double C[4*N][4*N], double A[4*N][4*N], double B[4*N][4*N])
{
  size_t i, j, k;
  //PIPS generated variable
  long int LU_NUB0, LU_IB0, LU_IND0;
  //PIPS generated variable
  size_t j0, k0, j1, k1, j2, k2, j3, k3, j4, k4;
  //PIPS generated variable
  long int LU_NUB1, LU_IB1, LU_IND1;
  //PIPS generated variable
  long int LU_NUB2, LU_IB2, LU_IND2;
  //PIPS generated variable
  size_t k00, k01, k02, k03, k04;
  //PIPS generated variable
  long int LU_NUB3, LU_IB3, LU_IND3;
  LU_NUB0 = (4*N-1-0+1)/1;
  LU_IB0 = MOD(LU_NUB0, 4);
l99996:
#pragma omp parallel for private(LU_IB2, LU_IND2, LU_NUB2, j0)
  for(LU_IND0 = 0; LU_IND0 <= LU_IB0-1; LU_IND0 += 1) {
l99998:       LU_NUB2 = (4*N-1-0+1)/1;
      LU_IB2 = MOD(LU_NUB2, 4);
l99993:
#pragma omp parallel for
      for(LU_IND2 = 0; LU_IND2 <= LU_IB2-1; LU_IND2 += 1) {
l99999:         fUjmZFhOPb(N, LU_IND0, LU_IND2, A, B, C);
l99997:         ;
      }
l99984:
#pragma omp parallel for private(LU_IB3, LU_IND3, LU_NUB3, k02, k04)
      for(LU_IND2 = LU_IB2; LU_IND2 <= LU_NUB2-1; LU_IND2 += 4) {
l99991:         LU_NUB3 = (4*N-1-0+1)/1;
        LU_IB3 = MOD(LU_NUB3, 4);
l99979:
        for(LU_IND3 = 0; LU_IND3 <= LU_IB3-1; LU_IND3 += 1) {
          C[LU_IND0*1+0][(LU_IND2+0)*1+0] += A[LU_IND0*1+0][LU_IND3*1+0]*B[LU_IND3*1+0][(LU_IND2
              +0)*1+0];
l99983:           ;
        }
l99978:
        for(LU_IND3 = LU_IB3; LU_IND3 <= LU_NUB3-1; LU_IND3 += 4) {
          C[LU_IND0*1+0][(LU_IND2+0)*1+0] += A[LU_IND0*1+0][(LU_IND3+0)*1+0]*B[(LU_IND3+0)*1+0][(
              LU_IND2+0)*1+0];
          C[LU_IND0*1+0][(LU_IND2+0)*1+0] += A[LU_IND0*1+0][(LU_IND3+1)*1+0]*B[(LU_IND3+1)*1+0][(
              LU_IND2+0)*1+0];
          C[LU_IND0*1+0][(LU_IND2+0)*1+0] += A[LU_IND0*1+0][(LU_IND3+2)*1+0]*B[(LU_IND3+2)*1+0][(
              LU_IND2+0)*1+0];
          C[LU_IND0*1+0][(LU_IND2+0)*1+0] += A[LU_IND0*1+0][(LU_IND3+3)*1+0]*B[(LU_IND3+3)*1+0][(
              LU_IND2+0)*1+0];
l99980:           ;
        }
        k04 = 0+MAX0(LU_NUB3, 0)*1;
l99989:         KmVcvdqGyh(N, LU_IND0, LU_IND2, A, B, C);
l99987:
        for(k02 = 0; k02 < 4*N; k02 += 1)
          C[LU_IND0*1+0][(LU_IND2+2)*1+0] += A[LU_IND0*1+0][k02]*B[k02][(LU_IND2+2)*1+0];
l99986:         yBsjWVaOsv(N, LU_IND0, LU_IND2, A, B, C);
l99994:         ;
      }
      j0 = 0+MAX0(LU_NUB2, 0)*1;
  }
l99985:
#pragma omp parallel for private(j3, j4)
  for(LU_IND0 = LU_IB0; LU_IND0 <= LU_NUB0-1; LU_IND0 += 4) {
l99992:
#pragma omp parallel for private(k4)
      for(j4 = 0; j4 < 4*N; j4 += 1)
l99995:
```

```
            for(k4 = 0; k4 < 4*N; k4 += 1)
276             C[(LU_IND0+0)*1+0][j4] += A[(LU_IND0+0)*1+0][k4]*B[k4][j4];
    l99990:
    #pragma omp parallel for private(LU_IB1, LU_IND1, LU_NUB1, k3)
279         for(j3 = 0; j3 < 4*N; j3 += 1) {
            LU_NUB1 = (4*N-1-0+1)/1;
            LU_IB1 = MOD(LU_NUB1, 4);
282 l99982:
            for(LU_IND1 = 0; LU_IND1 <= LU_IB1-1; LU_IND1 += 1)
                C[(LU_IND0+1)*1+0][j3] += A[(LU_IND0+1)*1+0][LU_IND1*1+0]*B[LU_IND1*1+0][j3];
285 l99981:
            for(LU_IND1 = LU_IB1; LU_IND1 <= LU_NUB1-1; LU_IND1 += 4) {
                C[(LU_IND0+1)*1+0][j3] += A[(LU_IND0+1)*1+0][(LU_IND1+0)*1+0]*B[(LU_IND1+0)*1+0][j3];
288             C[(LU_IND0+1)*1+0][j3] += A[(LU_IND0+1)*1+0][(LU_IND1+1)*1+0]*B[(LU_IND1+1)*1+0][j3];
                C[(LU_IND0+1)*1+0][j3] += A[(LU_IND0+1)*1+0][(LU_IND1+2)*1+0]*B[(LU_IND1+2)*1+0][j3];
                C[(LU_IND0+1)*1+0][j3] += A[(LU_IND0+1)*1+0][(LU_IND1+3)*1+0]*B[(LU_IND1+3)*1+0][j3];
291         }
            k3 = 0+MAX0(LU_NUB1, 0)*1;
        }
294 l99988:      wYZYlYYhNv(N, LU_IND0, A, B, C);
        WwZLXIuStG(N, LU_IND0, A, B, C);
    }
297 i = 0+MAX0(LU_NUB0, 0)*1;
}
void parse_matrix(int n, char *s, double a[4*n][4*n])
300 {
    int sl;
    //arbitrary long variable
303 char buffer[687];
    int i;
    int j;
306 int p;
    int k;
    sl = strlen(s);
309 i = -1;
    j = 0;
    p = 0;
312 k = 0;
l99995:
    for(k = 0; k <= sl-1; k += 1)
315     if (s[k]=='[') {
            j = 0;
            i++;
318     }
        else if (s[k]==' ') {
            buffer[p] = '\0';
321         a[i][j] = atof(buffer);
            p = 0;
            j++;
324     }
        else if (s[k]==']') {
            buffer[p] = '\0';
327         a[i][j] = atof(buffer);
            p = 0;
        }
330     else {
            buffer[p] = s[k];
            p++;
333     }
}
int main(int argc, char **argv)
336 {
    int n = atoi(argv[1]);
    double (*a)[4*n][4*n], (*b)[4*n][4*n], (*c)[4*n][4*n];
339 //PIPS generated variable
    int LU_NUB0, LU_IB0, LU_IND0;
    //PIPS generated variable
342 int LU_NUB1, LU_IB1, LU_IND1;
    a = malloc(sizeof(double)*16*n*n);
    b = malloc(sizeof(double)*16*n*n);
```

```
345     c = calloc(sizeof(double), 16*n*n);
        int i = 0;
        int j = 0;
348     {
            //Parse first matrix
            //PIPS generated variable
351         int n0 = n;
            //PIPS generated variable
            char *s0 = argv[2];
354         //PIPS generated variable
            double (*P_0)[4*n0][4*n0] = *a;
            {
357             int sl;
                //arbitrary long variable
                char buffer[687];
360             int i;
                int j;
                int p;
363             int k;
                sl = strlen(s0);
                i = -1;
366             j = 0;
                p = 0;
                k = 0;
369 l99995:            RwZXjQnHhy(n0, buffer, &i, &p, sl, *P_0, s0);
            }
        }
372     {
            //parse second matrix
            //PIPS generated variable
375         int n1 = n;
            //PIPS generated variable
            char *s1 = argv[3];
378         //PIPS generated variable
            double (*P_1)[4*n1][4*n1] = *b;
            {
381             int sl;
                //arbitrary long variable
                char buffer[687];
384             int i;
                int j;
                int p;
387             int k;
                sl = strlen(s1);
                i = -1;
390             j = 0;
                p = 0;
                k = 0;
393 l99998:            LU_NUB1 = (sl-1-0+1)/1;
            LU_IB1 = MOD(LU_NUB1, 4);
l99990:
396         for(LU_IND1 = 0; LU_IND1 <= LU_IB1-1; LU_IND1 += 1) {
                if (s1[LU_IND1*1+0]=='[') {
                    j = 0;
399                 i++;
                }
                else if (s1[LU_IND1*1+0]==' ') {
402                 buffer[p] = '\0';
                    ((*P_1)[i])[j] = atof(buffer);
                    p = 0;
405                 j++;
                }
                else if (s1[LU_IND1*1+0]==']') {
408                 buffer[p] = '\0';
                    ((*P_1)[i])[j] = atof(buffer);
                    p = 0;
411             }
                else {
                    buffer[p] = s1[LU_IND1*1+0];
414                 p++;
```

```
            }
l99992:             ;
        }
l99989:
        for(LU_IND1 = LU_IB1; LU_IND1 <= LU_NUB1-1; LU_IND1 += 4) {
            if (s1[(LU_IND1+0)*1+0]=='[') {
                j = 0;
                i++;
            }
            else if (s1[(LU_IND1+0)*1+0]==' ') {
                buffer[p] = '\0';
                ((*P_1)[i])[j] = atof(buffer);
                p = 0;
                j++;
            }
            else if (s1[(LU_IND1+0)*1+0]==']') {
                buffer[p] = '\0';
                ((*P_1)[i])[j] = atof(buffer);
                p = 0;
            }
            else {
                buffer[p] = s1[(LU_IND1+0)*1+0];
                p++;
            }
            if (s1[(LU_IND1+1)*1+0]=='[') {
                j = 0;
                i++;
            }
            else if (s1[(LU_IND1+1)*1+0]==' ') {
                buffer[p] = '\0';
                ((*P_1)[i])[j] = atof(buffer);
                p = 0;
                j++;
            }
            else if (s1[(LU_IND1+1)*1+0]==']') {
                buffer[p] = '\0';
                ((*P_1)[i])[j] = atof(buffer);
                p = 0;
            }
            else {
                buffer[p] = s1[(LU_IND1+1)*1+0];
                p++;
            }
            if (s1[(LU_IND1+2)*1+0]=='[') {
                j = 0;
                i++;
            }
            else if (s1[(LU_IND1+2)*1+0]==' ') {
                buffer[p] = '\0';
                ((*P_1)[i])[j] = atof(buffer);
                p = 0;
                j++;
            }
            else if (s1[(LU_IND1+2)*1+0]==']') {
                buffer[p] = '\0';
                ((*P_1)[i])[j] = atof(buffer);
                p = 0;
            }
            else {
                buffer[p] = s1[(LU_IND1+2)*1+0];
                p++;
            }
            if (s1[(LU_IND1+3)*1+0]=='[') {
                j = 0;
                i++;
            }
            else if (s1[(LU_IND1+3)*1+0]==' ') {
                buffer[p] = '\0';
                ((*P_1)[i])[j] = atof(buffer);
                p = 0;
```

```
                j++;
            }
            else if (s1[(LU_IND1+3)*1+0]==']') {
                buffer[p] = '\0';
                ((*P_1)[i])[j] = atof(buffer);
                p = 0;
            }
            else {
                buffer[p] = s1[(LU_IND1+3)*1+0];
                p++;
            }
l99991:             ;
        }
        k = 0+MAX0(LU_NUB1, 0)*1;
    }
  }
  //multiply matrix
  matrix_mul_matrix(n, *c, *a, *b);


l99996:   LU_NUB0 = (4*n-1-0+1)/1;
  LU_IB0 = MOD(LU_NUB0, 4);
l99994:
  for(LU_IND0 = 0; LU_IND0 <= LU_IB0-1; LU_IND0 += 1) {
    printf("%f ", ((*c)[LU_IND0*1+0])[LU_IND0*1+0]);
l99999:     ;
  }
l99993:
  for(LU_IND0 = LU_IB0; LU_IND0 <= LU_NUB0-1; LU_IND0 += 4) {
    printf("%f ", ((*c)[(LU_IND0+0)*1+0])[(LU_IND0+0)*1+0]);
    printf("%f ", ((*c)[(LU_IND0+1)*1+0])[(LU_IND0+1)*1+0]);
    printf("%f ", ((*c)[(LU_IND0+2)*1+0])[(LU_IND0+2)*1+0]);
    printf("%f ", ((*c)[(LU_IND0+3)*1+0])[(LU_IND0+3)*1+0]);
l99997:     ;
  }
  i = 0+MAX0(LU_NUB0, 0)*1;
  free(a);
  free(b);
  free(c);
  return 0;
}
```