Università degli Studi di Verona

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Progetto corso Sicurezza del Software

Offuscatore per il linguaggio Python

Candidati:
Giovanni Bellorio
Matricola VR419955

Kevin Costa
Matricola VR424301

Alessandro Cosma
Matricola VR420117

Anno Accademico 2018-2019

INDICE

1 INTRODUZIONE

Lo scopo di questo progetto è realizzare un offuscatore per il linguaggio Python. Il programma che implementa l'offuscatore sarà realizzato anch'esso in Python e prenderà in considerazione le seguenti tipologie di offuscamento del codice:

- **Layout obfuscation**
- **Data obfuscation**
- **Control Flow obfuscation**

Nel capitolo 2 daremo una panoramica generale sui concetti legati all'offuscamento del codice, spiegando cos'è formalmente un offuscatore, quali sono le sue carateristiche, i suoi pregi e i suoi difetti, infine spiegheremo nel dettaglio le varie tecniche di offuscamento che andremo ad utilizzare.

Nel capitolo 3 verrà presentato il nostro programma per l'offuscamento del codice. Elencheremo le varie scelte progettuali effettuate in fase di progettazione (3.1) e successivamente presenteremo passo passo il codice che implementa l'algoritmo (3.2).

Infine, nel capitolo 4 mostreremo un semplice esempio del suo fuzionamento accompagnato da alcuni test.

2

BACKGROUND

Quando si considera un software sviluppato e distribuito, la maggior parte del suo "know-how" risiede nel prodotto stesso. Risulta quindi essenziale la protezione del suo codice e della sua proprietà intellettuale.

Attacchi di *reverse-engineering* hanno l'obiettivo di attuare analisi statiche e dinamiche del codice, al fine di capire il funzionamento interno di un programma proprietario di interesse. La protezione è necessaria ad ogni livello di funzionamento del programma, a partire dal codice sorgente, fino ad arrivare al codice macchina, passando anche per le informazioni di interesse che il programma da proteggere scambia con l'ambiente di esecuzione.

Esistono varie tecniche di protezione del codice con differenti obiettivi:

- *Misure legali*: leggi, copiright, patenti, brevetti e licenze ideate per proteggere la "forma" e l'idea di un programma.
- *Offuscamento*: rendere difficile il reverse-engineering del programma.
- *Watermarking del software*: inserire una firma o un riconoscimento contro il furto del programma.
- *Tamper-proofing*: rilevare violazioni contro l'integrità del codice o modifiche illegali.

Essendo il nostro progetto mirato alla realizzazione di un offuscatore per codice Python, descriveremo ora in dettaglio il concetto di offuscatore e presenteremo alcune delle principali tecniche utilizzate.

2.1 OFFUSCATORE

Sia:

$$P \xrightarrow{T} P'$$

una trasformazione di un programma sorgente P in un programma P' con $P, P' \in Programs$, cioè l'insieme dei programmi.

$P \xrightarrow{T} P'$ è una trasformazione di offuscamento se P e P' hanno lo stesso *comportamento osservabile*.

Più precisamente, affinchè $P \xrightarrow{T} P'$ sia una trasformazione di offuscamento legale, devono valere le seguenti condizioni:

1. Se P non riesce a terminare o termina in una condizione di errore, allora P' può o non può terminare.
2. Altrimenti, P' deve terminare e produrre lo stesso output di P.

Il *livello di sicurezza* rispetto al reverse-engineering che un offuscatore aggiunge ad un dato programma dipende da vari fattori: raffinatezza della trasformazione di offuscamento, potenza del deoffuscatore, quantità di risorse (tempo e memoria) disponibili al deoffuscatore.

2.1.1 *Impossibilità di offuscare un programma*

Una trasformazione di offuscamento non potrà mai proteggere completamente un programma. Fred Cohen, il primo ad introdurre la diversità dei programmi come difesa contro gli attacchi automatici ai sistemi operativi, affermava che:

"Any protection scheme other than a physical one depends on the operation of a finite state machine, and ultimately, any finite state machine can be examined and modified at will, given enough time and effort. The best we can ever do is delay the attack by increasing the complexity of making desired alterations (security through obscurity)"

[inserire risultati paper "On the impossibility of code obfuscation"]

2.1.2 *Valutare le trasformazioni offuscanti*

Una trasformazione offuscante è valutate in base alle seguenti proprietà:

- **Potenza**: quanto più difficile è comprendere il codice offuscato rispetto a quello originale.
- **Resilienza**: misura quanto una trasformazione regge l'attacco di un deoffuscatore automatico.
- **Costo**: misura quanto overhead computazionale (tempo e spazio) viene aggiunge all'applicazione offuscata.
- **Furtività**: misura quanto bene il codice nuovo introdotto con la trasformazione si integra con il codice originale.

POTENZA Sia T una trasformazione che preserva il comportamento, tale che $P \xrightarrow{T} P'$ trasfroma un programma P in un programma P'. Sia $E(P)$ la complessità di P.

$T_{pot}(P)$, la potenza di T rispetto ad un programma P, è una misura del grado con cui T cambia la complessità di P.

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

T è una *trasformazione di offuscamento potente* se $T_{pot}(P) > 0$.

RESILIENZA Sia T una trasformazione che preserva il comportamento, tale che $P \xrightarrow{T} P'$ trasfroma un programma P in un programma P'. $T_{res}(P)$ è la *resilienza* di T rispetto ad un programma P.

$T_{res}(P) = one - way$ se l'informazione è rimossa da P tale che P non può essere ricostruito da P'.

$$T_{res} = Resilience(T_{Deobfuscator\_effort}, T_{Programmer\_effort})$$

COSTO Il costo di una trasformazione offuscante è il tempo di esecuzione (o la quantità di spazio occupato) del programma offuscato rispetto a quello originale.

$T_{cost}(P)$, cioè il costo di un programma P, può essere classificato su una scala di quattro punti come segue:

- **caro**: se l'esecuzione di P' richiede esponenzialmente più risorse di P.

- **costoso**: se l'esecuzione di $P^{'}$ richiede $O(n^p), p > 1$ risorse più di P.
- **economico**: se l'esecuzione di $P^{'}$ richiede $O(n)$ risorse più di P.
- **gratuito**: se l'esecuzione di $P^{'}$ richiede $O(1)$ risorse più di P.

FURTIVITA Mentre una trasformazione resiliente potrebbe non essere suscettibile agli attacchi di deoffuscatori automatici, potrebbe comunque essere suscettibile agli attacchi degli umani.

In particolare, se una trasformazione introduce un nuovo codice che differisce ampiamente da ciò che è presente nel programma originale, sarà facile individuare per un reverse engineer. Tali trasformazioni non sono furtive (a.k.a. *unstealthy*).

La furtività è una nozione sensibile al contesto. Il codice può essere nascosto in un programma ma estremamente visibile in un altro.

2.2 TECNICHE DI OFFUSCAMENTO DEL CODICE

In questa sezione andremo a descrivere a livello teorico le tecniche di offuscamento che saranno utilizzate successivamente all'interno del nostro offuscatore.

2.2.1 *Layout Obfuscation*

L'obiettivo del *Layout Obfuscation* è cambiare o rimuove le informazioni utili dal codice senza influire sulle istruzioni reali.

Tecniche principali utilizzate per questo tipo di offuscamento:

- **Rimozione commenti**: vengono rimossi i commenti e i docstring presenti nel codice.
- **Rinominazione degli identificatori** : i nuovi nomi possono utilizzare diversi schemi, numeri, caratteri non stampabili o caratteri invisibili. Inoltre i nomi possono essere riutilizzati, purchè usati in ambiti diversi.

Queste sono trasformazioni *one-way* in quanto la formattazione originale non può essere ripristinata.

Vantaggi:

- potente contro l'attacco umano
- per quanto riguarda i costi è gratuito.
- è molto utile per offuscare *authorship* del codice.

Svantaggi:

- Ha una bassa potenza, in quanto nella formattazione c'è poco contenuto semantico.

2.2.2 *Control Flow Obfuscation*

Il *Control Flow Obfuscation* trasforma un control-flow ben strutturato di un programma eseguibile in una struttura difficile da analizzare da un programma che rileva le trasformazioni del control-flow. Tale offuscamento può ostruire e fermare gli attacchi statici al control-flow del codice eseguibile sotto attacchi di tipo white-box. Questa tecnica costringe l'attaccante ad eseguire attacchi dinamici per ottenere il control-flow, in questo modo diventa molto più difficile da ottenere e aumenta significativamente lo sforzo dell'attacco.
Tecniche principali utilizzate per questo tipo di offuscamento:

- **Predicati opachi**: sono espressioni di valore booleano per le quali il difensore sa se restituiranno true, false o talvolta true e talvolta false
- **CFG Riducibili**: sono grafi che possono essere ridotti a un singolo nodo mediante una sequenza di applicazioni di due regole
- **Predicati opachi dinamici**: una famiglia di predicati opachi correlati, che valutano tutti lo stesso risultato ad ogni esecuzione, ma in diverse esecuzioni possono valutare risultati diversi

Vantaggi:

- potente contro l'analisi statica
- difficili da valutare dall'attaccante per la complessità esponenziale
- è molto utile per offuscare *authorship* del codice.

Svantaggi:

- Vulnerabile all'analisi dinamica

2.2.3 *Data Obfuscation*

Per offuscare una variabile v in un programma, la si converte dalla rappresentazione scelta inizialmente dal programmatore ad un rappresentazine più difficile da analizzare per l'attaccante. Qualsiasi valore che una variabile v può assumere deve essere rappresentato nella nuova variabile offuscata. Si ha bisogno di due funzioni:

$$\text{encode } E : T \to T'$$

$$\text{decode } D : T' \to T$$

e le operazioni sulle variabili devono essere interpretate nella nuova rappresentazione.
Tecniche principali utilizzate per questo tipo di offuscamento:

- **Encoding Integers**: gli interi possono essere codificati con "trucchi" della teoria dei numeri o criptandoli.
- **Encoding Booleans**: si usa lo stesso approccio per l'*Encoding Integers*, ma è molto più semplice perchè il range dei loro valori è conosciuto
- **Encoding Literal Data**: possibili soluzioni possono essere scomporre i dati in parti più piccole e nasconderle nel programma, fare lo *xor* di una stringa con un valore costante conosciuto o usare macchine di Mealy.

- **Encoding Arrays**: si possono riordinare o ristrutturare gli elementi dell'array

Vantaggi:

- potente contro l'attacco umano.
- per quanto riguarda i costi è gratuito.
- i *Literal Data* contengono molte informazioni semantiche.

Svantaggi:

- rallentamento delle prestazioni per le continue decodifiche.

3 OFFUSCATORE PER PYTHON

3.1 SCELTE PROGETTUALI

3.1.1 *Assunzioni*

Per l'offuscatore che abbiamo crato sono state fatte le seguenti assunzioni:

- Essendo python un linguaggio di programmazine non tipato, quando faccio l'encoding in una macchina di Mealy subito dopo faccio il decoding perché se in una condizione *if* entro nel ramo *else* e viene cambiato tipo di una variabile (da stringa a intero), non riesco più a trattarla come stringa, ma dorò trattarla come array permutation.
- Nell'array permutation assumiamo che l'array contenga dei valori e non dei nomi di variabili, altrimenti bisognerebbe andare in profondità.
- Nel Control Flow Obfuscation assumiamo di inserire i predicati opachi non prima di una riga che termini con ":", per evitare possibili errori di indentazione nell'inserimento del predicato opaco.

3.2 ALGORITMO E CODICE

Viene ora presentato il codice che implementa l'offuscatore.

Il codice sorgente è composto da un metodo `main()` che rappresenta l'entry point del nostro programma e da una serie di funzioni che vengono invocate all'interno di tale metodo per svolgere le funzioni di offuscamento.

```
def main():

    print("-----------------------------------")
    print("OBFUSCATION STARTED\n")

    name_src = sys.argv[1] # name of the source file, passed as argument
    name_dest = name_src+"_OBFUSCATED.py" # name of the destination file

    file_SRC = open(name_src, "r")
    file_DEST = open(name_dest, "w")

    # Call 'remove_comments_and_docstrings()' function on the source file object
    # which it is first converted to a string through the read() method.
    # 'out' contains a string with the entire code
    out = remove_comments_and_docstrings(file_SRC.read())

    # Call 'opaque_predicate()' function
    out = opaque_predicate(out)

    # Call 'encoding_literal_data()' function
    out = encoding_literal_data(out)

    # Call 'obfuscate_variables()' function
    out = obfuscate_variables(out)

    # Call 'obfuscate_functions()' function
```

```
    out = obfuscate_functions(out)

    # Write the result into file_DEST
    file_DEST.write(out)
    file_DEST.close()
    file_SRC.close()

    print("\nOBFUSCATION ENDED")
    print("-------------------------------------")


if __name__ == '__main__':
    main()
```

All'inizio dell'esecuzione del programma, viene letto il nome del file sorgente, contenente il software Python da offuscare e salvato nella variabile `name_src`: questa informazione deve essere passata come argomento di input all'offuscatore, con una sintassi del tipo:

```
python3 offuscatore.py <nome_file_da_offuscare>.py
```

dove `offuscatore.py` è il nome dell' offuscatore e `<nome_file_da_offuscare>` è il nome del file di input.

Successivamente viene aperto in sola lettura il file sorgente (variabile `file_SRC`) e viene creato un nuovo file, chiamato come il precedente ma con l'aggiunta della suffisso "OBFUSCATED" e salvato nella variabile `file_DEST`: esso rappresenta il file che conterrà l'output del nostro programma cioè il software offuscato.

In seguito vengono chiamate in sequenza le seguenti funzioni:

- **remove_comments_and_docstrings()**: esegue un *layout obfuscation* rimuovendo i commenti e la documentazione;
- **opaque_predicate()**: esegue un *control flow obfuscation*, inserendo predicati opachi;
- **encoding_literal_data()**: esegue un *data obfuscation*, codificando stringhe, array e interi;
- **obfuscate_variables()**: esegue un *layout obfuscation*, rinominando i nomi di variabili;
- **obfuscate_functions()**: esegue un *layout obfuscation*, rinominando i nomi di funzione e il relativo "corpo".

Ognuna di queste funzioni riceve in input il programma (variabile `out`) trasformato dalla funzione di offuscamento precedente, vi applica un certo insieme di trasformazioni offuscanti e successivamente restituisce un output trasformato, che viene poi dato in input alla funzione di offuscamento successiva.

Alla fine del metodo `main()`, viene scritto nel file di output il programma offuscato e vengono chiusi i file di input e output precedentemente aperti.

3.2.1 *Codice di Layout Obfuscation*

remove_comments_and_docstrings()

La prima funzione di layout obfuscation è *remove_comments_and_docstrings*. Questa funzione ha l'obiettivo di eliminare dal codice sorgente tutti i com-

menti e tutte le stringhe relative alla documentazione del codice presenti al suo interno.

```
def remove_comments_and_docstrings(source):
    """
    Returns 'source' minus comments and docstrings.
    """

    print("-> removing comments and docstrings")

    io_obj = StringIO(source)
    out = ""
    prev_toktype = tokenize.INDENT
    last_lineno = -1
    last_col = 0
    myline = ""
    is_junk = False

    for tok in tokenize.generate_tokens(io_obj.readline):
        token_type = tok[0]
        token_string = tok[1]
        start_line, start_col = tok[2]
        end_line, end_col = tok[3]
        ltext = tok[4]
        # The following two conditionals preserve indentation.
        # This is necessary because we're not using tokenize.untokenize()
        # (because it spits out code with copious amounts of oddly-placed
        # whitespace).
        if start_line > last_lineno:
            last_col = 0
        if start_col > last_col:
            myline += (" " * (start_col - last_col))
        # Remove comments:
        if token_type == tokenize.COMMENT:
            is_junk = True
            pass
        # This series of conditionals removes docstrings:
        elif token_type == tokenize.STRING:
            if prev_toktype != tokenize.INDENT:
        # This is likely a docstring; double-check we're not inside an operator:
                if prev_toktype != tokenize.NEWLINE:
                    # Note regarding NEWLINE vs NL: The tokenize module
                    # differentiates between newlines that start a new statement
                    # and newlines inside of operators such as parens, brackes,
                    # and curly braces. Newlines inside of operators are
                    # NEWLINE and newlines that start new code are NL.
                    # Catch whole-module docstrings:
                    if start_col > 0:
                        # Unlabelled indentation means we're inside an operator
                        myline += token_string
                    # Note regarding the INDENT token: The tokenize module
                    # does not label indentation inside of an operator (parens,
                    # brackets, and curly braces) as actual indentation.
                    # For example:
                    # def foo():
                    # "The spaces before this docstring are tokenize.INDENT"
                    # test = [
                    # "The spaces before this string do not get a token"
                    # ]
                    else:
                        is_junk = True
                else:
                    is_junk = True
            else:
                is_junk = True
        else:
            myline += token_string

        if token_type == tokenize.NEWLINE or token_type == tokenize.NL:
```

```
            if not (is_junk and re.match(r'^\s*$', myline)):
                out += myline
            myline = ""
            is_junk = False

        prev_toktype = token_type
        last_col = end_col
        last_lineno = end_line
    return out
```

Questa funzione sfrutta il modulo `tokenize` il quale fornisce uno scanner lessicale per codice sorgente Python. Essa preserva l'indentazione del codice, essenziale per il corretto funzionamento di un programma Python, inoltre, ogni qual volta una riga viene "svuotata" di tutti suoi token perchè non vi è presenta nessuna istruzione, quest'ultima viene eliminata dal codice per non lasciare inutili spazi vuoti.

obfuscate_variables()

La seconda funzione che esegue il layout obfuscation è *obfuscate_variables*. La funzione prende in input il codice del programma da offuscare (variabile `out`) e ritorna il medesimo codice con i nomi delle variabili offuscate.

Riportiamo il codice:

```
def obfuscate_variables(out):
    """
        Returns 'source' code with variables offuscated.
    """

    print("-> obfuscating variables")

    # Tree object contains the AST of the code
    tree = ast.parse(out)

    # Extraction of all variables defined
    variables = get_variables(tree)

    # Choose new names
    for variable in variables:
        VARIABLES_DICT[variable] = random_sequence()

    # Renomination of all variable occurrences
    tree = rename_variables(tree, VARIABLES_DICT)

    # Convert the new AST to a valid code
    return astor.to_source(tree)
```

Attraverso il comando `ast.parse(out)`, il codice contenuto nella variabile `out` viene convertito in una struttura ad albero e salvata nella variabile `tree`. Questo albero rappresenta l'*abstract syntax tree* del codice sorgente del programma dato in input.

Successivamente vengono estratte tutte le variabili presenti nel programma, attraverso la funzione `get_variables()` e salvate nella lista `variables`. Ad ognuna di esse viene poi associato un nuovo nome casuale, utilizzando la funzione `random_sequence()`.
L'associazione tra il nome della variabile originale e il nuovo nome viene salvata nel dizionario `VARIABLES_DICT`, definito come variabile globale.

Infine, tramite la funzione `rename_variables()`, viene applicata la ridenominazione delle variabili sulla base delle associazioni precedentemente stabilite.

Viene quindi ritornato il codice sorgente del programma, invocando la funzione `astor.to_source(tree)`, che prende come input l'abstract syntax tree, su cui sono state apportate le precedenti trasformazioni offuscanti e ritorna come output il relativo codice sorgente. Questo metodo, che trasforma un abstract syntax tree nel corrispettivo codice sorgente, appartiene al modulo `astor`, esterno alla libreria Python.

Riportiamo ora il codice delle funzioni `get_variables()`, `random_sequence()` e `rename_variables()`, seguito da una breve descrizione.

La funzione `get_variables()` prende come input un abstract syntax tree e ritorna come output un dizionario, contenente ogni variabile presente nell albero (*key*) e il relativo tipo (*value*).

```
def get_variables(tree):
    """
    :param tree:
    :return: a dictionary contains each variable (key) and its type (value)
    contained in the code (passed as AST 'tree')
    """

    variables = {}

    for node in ast.walk(tree):
        # Saved variables declared
        if isinstance(node, ast.Assign) and isinstance(node.targets[0], ast.Name):
            if node.targets[0].id not in variables:
                    variables[node.targets[0].id] = type(node.value).__name__
        if isinstance(node, ast.Attribute):
            if isinstance(node.value, ast.Name):
                if node.value.id == "self":
                    variables[node.attr] = type(node.attr).__name__

    return variables
```

La funzione esegue una visita dell'albero, svolta tramite il metodo `ast.walk()`. In questo caso vengono presi in considerazione i nodi dell'albero che rappresentano un assegnamento (`ast.Assign`) oppure un attributo (`ast.Attribute`); quest'ultimo caso è necessario in presenza di una definizione di classe, in cui una variabile viene anteceduta dal prefisso `self`.

I nomi di variabili e il loro tipo vengono salvati nel dizionario `variables`, ritornato come output dalla funzione.

La funzione `random_sequence()` genera una sequenza casuale di 23 caratteri, scelti in un alfabeto di soli due elementi: il numero '0' oppure la lettera 'O'. La sequenza creata è sempre diversa dalla precedenti sequenze generate.

```
def random_sequence():

    seq = generate_sequence()

    if seq not in RANDOM_SEQUENCES_SET:
        RANDOM_SEQUENCES_SET.add(seq)
        return seq
    else:
        random_sequence()

def generate_sequence():
```

```
    cryptorand = SystemRandom()

    sequence = []

    for i in range(22):
        flip = random.randint(1, 2)
        if flip == 1:
            sequence.append("o")
        else:
            sequence.append("O")

    random.seed()
    cryptorand.shuffle(sequence)

    return "O"+"".join(sequence)
```

Questa funzione utilizza il metodo `generate_sequence()` per generare la sequenza. Quest'ultimo utilizza il modulo `randint` e `SystemRandom` della libreria `random`. Ogni sequenza generata, prima di essere ritornata alla funzione `random_sequence()`, viene mescolata tramite il metodo `cryptorand.shuffle()`: questo ci permette di avere meno probabilità di generare dupliacati.

Successivamente, se la sequenza generata è una sequenza valida (non è un duplicato), viene salvata nell'insieme `RANDOM_SEQUENCES_SET`, definito come globale e ritornata come output dalla funzione; altrimenti viene scartata e ne viene generata una nuova.

La funzione `rename_variables()` prende in input un abstract sintax tree (parametro `tree`) e un dizionario contenente le associazioni tra nomi originali delle variabili e i nuovi nomi generati (parametro `variables`). Ritorna in output l' abstract sintax tree con i nomi sostituiti.

```
def rename_variables(tree, variables):
    # Rename names of variables used (general usage)
    for node in ast.walk(tree):
        if isinstance(node, ast.Name) and node.id in variables:
            node.id = variables[node.id]
        if isinstance(node, ast.Attribute) and node.attr in variables:
            node.attr = variables[node.attr]

    return tree
```

obfuscate_functions()

La terza e ultima funzione che esegue il layout obuscation è `obfuscate_functions`. La funzione prende in input il codice del programma da offuscare (variabile `out`) e ritorna il medesimo codice con il nome della funzione, i nomi dei parametri, e le occorrenze di quest'ultimi nel corpo della funzione offuscati. La funzione sostituisce poi le occorrenze del nuovo nome presenti all'interno del codice.

```
def obfuscate_functions(out):
    """
        Returns 'source' with function def and call obfuscated.
    """

    print("-> obfuscating function definitions, body and call")
```

```
# Tree object contains the AST of the code
tree = ast.parse(out)

critical_func_names = ["__init__", "__str__" , "__repr__", "main"]

# --> obfuscate definition and body
functions = {}

for node in ast.walk(tree):
    # Saved functions declared
    if isinstance(node, ast.FunctionDef):
        # save function name
        if node.name not in critical_func_names:
            # assign random name
            functions[node.name] = random_sequence()
            # change the name of def function
            node.name = functions[node.name]

        # Obfuscate arguments into the function def

        # dict for function arguments name
        arguments = {}
        # get argument list structure from node
        arguments_list = node.args.args
        # get the numbers of arguments
        num_arguments = len(arguments_list)

        # assign a new rand sequence for arg name and
        # bind it with the old name
        for i in range(0, num_arguments):
            # if the name is a homonym of a variable previously found,
            # then I use that name
            if arguments_list[i].arg in VARIABLES_DICT:
                arguments[arguments_list[i].arg] =
                    VARIABLES_DICT[arguments_list[i].arg]
            # otherwise, choose a new random name
            else:
                arguments[arguments_list[i].arg] = random_sequence()

            # assign the new random name at each argument
            # between two parenthesis
            arguments_list[i].arg = arguments[arguments_list[i].arg]

        # rename arguments into body function
        rename_variables(node, arguments)

# --> obfuscate occurrences of various functions name

for node in ast.walk(tree):
    # Saved functions declared
    if isinstance(node, ast.Call):
        # if "func" is instance of ast.Name and it has been previously
        # defined, then change name
        if isinstance(node.func, ast.Name) and node.func.id in functions:
            node.func.id = functions[node.func.id]
        # if "func" is instance of ast.Attribute and it has been previously
        # defined, then change name
        if isinstance(node.func, ast.Attribute) and node.func.attr in functions:
            node.func.attr = functions[node.func.attr]

# Convert the new AST to a valid code
return astor.to_source(tree)
```

Dal codice ricevuto in input viene creato l'abstract syntax tree e salvato nella variabile `tree`. Successivamente, si esegue una visita dei nodi dell'albero (`ast.walk(tree)`) andando a selezionare solo quei nodi che rappresentano una definizione di funzione, cioè oggetti di tipo `ast.FunctionDef`.

Per ogni nodo selezionato, si controlla che non rappresenti un nome di

funzione "critico" cioè che non sia presente all'interno della lista `critical_func_names`: in questa lista sono presenti quei nomi di funzione, che se ridenominati causerebbero errori di esecuzione del programma Python, come ad esempio `main` oppure `_init_`.

Nel caso di un nodo valido, il nome della funzione viene salvato come chiave nel dizionario `functions` e gli viene associato un nome casuale come valore (`functions[node.name] = random_sequence()`); viene poi ridenominata la funzione con il nome scelto.

L'offuscamento prosegue andando a ridenominare gli argomenti che la funzione riceve in input e per ognuno di essi scelglie un nuovo nome casuale da associare, utilizzando la funzione `random_sequence()`, presentata nella sezione 3.2.1. Nel caso il nome di un argomento sia lo stesso nome di una variabile già ridenominata in precedenza, non viene generata nessuna nuova sequenza casuale, ma viene riutilizzata la sequenza casuale associata alla variabile omonima. Qui si può osservare l'importanza della dichiarazione a variabile globale del dizionario `VARIABLES_DICT`.
Successivamente le occorrenze dei parametri di funzione, ridenominati nella definizione, vengono a loro volta sostitutiti all'interno del corpo della funzione. Questa sostituzione utilizza il metodo `rename_variables()`, a cui viene passato il nodo (`node`), che rappresenta un oggetto `ast.FunctionDef` e la lista di parametri da ridenominare.

L'ultima parte di questo offuscamento, ha il compito di ridenominare le occorrenze della funzione offuscata quando viene invocata in altri punti del codice. Viene quindi effettuata una visita dell'albero sintattico astratto alla ricerca di nodi che rappresentino una chiamata a funzione (`ast.Call`). In questo caso, oltre a considerare una semplice invocazione della funzione offuscata (`node.func` istanza di tipo ast.Name), viene tenuto conto del fatto che una funzione, definita come metodo all'interno di una classe, potrebbe essere considerata un attributo di una certa istanza di un oggetto (`node.func` istanza di tipo ast.Attribute).

Infine, viene ritornato il codice sorgente del programma, invocando la funzione `astor.to_source(tree)`, che prende come input l'abstract syntax tree, su cui sono state apportate le precedenti trasformazioni offuscanti e ritorna come output il relativo codice sorgente.

3.2.2 *Codice di Control Flow Obfuscation*

opaque_predicate()

La funzione che abbiamo creato per Control Flow Obfuscation è `opaque_predicate()`. Questa funzione ha l'obiettivo di rendere più complicata l'analisi statica del Control Flow Graph del programma inserendo dei predicati opachi che aumentano la complessità del Control Flow Graph.

```
def opaque_predicate(out):
    """
        Returns 'source' with opaque predicate.
    """

    print("-> opaque_predicate")

    # Opaque predicate 1
    pred1 = "if(xxgxx[1] + xxgxx[1]^2) % 2 == 0:\n\
    xxgxx[5] = (xxgxx[1] * xxgxx[4]) % xxgxx[11] + xxgxx[6]% xxgxx[5]\n\
```

```
    xxgxx[14] = randint(0, 100)\n\
    xxgxx[4] = randint(0, 10) * xxgxx[11] + xxgxx[8]\n\
else: \n\
    xxgxx[2] = randint(0, 100)\n\
    xxgxx[5] = randint(0, 10) * xxgxx[11] + xxgxx[8]\n\
#print(xxgxx)\n\
"

    # Opaque predicate 2
    pred2 = "if(xxgxx[4]^3 - xxgxx[4]) % 3 == 0:\n\
    xxgxx[5] = (xxgxx[1] * xxgxx[4]) % xxgxx[11] + xxgxx[6]% xxgxx[5]\n\
    xxgxx[14] = randint(0, 100)\n\
    xxgxx[4] = randint(0, 10) * xxgxx[11] + xxgxx[8]\n\
else:\n\
    xxgxx[2] = randint(0, 100)\n\
    xxgxx[5] = randint(0, 10) * xxgxx[11] + xxgxx[8]\n\
#print(xxgxx)\n\
"

    # Opaque predicate 3
    pred3 = "if(7+xxgxx[4]^2 - 1 != xxgxx[5]^2):\n\
    xxgxx[5] = (xxgxx[1] * xxgxx[4]) % xxgxx[11] + xxgxx[6]% xxgxx[5]\n\
    xxgxx[14] = randint(0, 100)\n\
    xxgxx[4] = randint(0, 10) * xxgxx[11] + xxgxx[8]\n\
else:\n\
    xxgxx[2] = randint(0, 100)\n\
    xxgxx[5] = randint(0, 10) * xxgxx[11] + xxgxx[8]\n\
"

    # Opaque predicate 4
    pred4 = "if(xxgxx[4]^xxnxx - xxgxx[5]^xxnxx % xxgxx[4] - xxgxx[5]):\n\
    xxgxx[5] = (xxgxx[1] * xxgxx[4]) % xxgxx[11] + xxgxx[6]% xxgxx[5]\n\
    xxgxx[14] = randint(0, 100)\n\
    xxgxx[4] = randint(0, 10) * xxgxx[11] + xxgxx[8]\n\
else:\n\
    xxgxx[2] = randint(0, 100)\n\
    xxgxx[5] = randint(0, 10) * xxgxx[11] + g[8]\n\
"


    number = randint(1, 4)
    predicate = 'pred' + str(number)
    # Set a flag to False
    flag = False

    # check a position where to insert the opaque predicate
    # the position is choiced randomly and it can't be insert
    # after a row that finish with ':'
    while (flag == False):
        try:
            pos = randint(0, int(len(out) / 2))
            if (out[pos] == '\n' and out[pos - 1] != ':' and out[pos - 1] != '\n'):

                # Count tab indent position
                out_split = out[:pos].split("\n")
                for row in out_split:

                    tab = " "
                    indent = ""

                    while True:
                        if " " in row[0:4]:
                            row = row[4:len(row)]
                            indent += tab
                        else:
                            break

                # Add indent to predicate
                predicate_split = eval(predicate).split("\n")
                predicate = ""
```

```
                for line in predicate_split:
                    predicate += indent + line + "\n"

                # Insert the opaque predicate
                out = out[:pos] + '\n' + predicate + out[pos:]

                pos += len(pred1) + 3
                flag = True

        except IndexError as error:
            # Output expected IndexErrors.
            print("string index out of range")


    number = randint(1, 4)
    predicate = 'pred' + str(number)
    # set a flag to False
    flag = False

    # check a position where to insert the opaque predicate
    # the position is choiced randomly and it can't be insert
    # after a row that finish with ':'
    while (flag == False):
        try:
            pos = randint(0, int(len(out) / 2))
            if (out[pos] == '\n' and out[pos - 1] != ':' and out[pos - 1] != '\n'):

                # Count tab indent position
                out_split = out[:pos].split("\n")
                for row in out_split:

                    tab = " "
                    indent = ""

                    while True:
                        if " " in row[0:4]:
                            row = row[4:len(row)]
                            indent += tab
                        else:
                            break

                # Add indent to predicate
                predicate_split = eval(predicate).split("\n")
                predicate = ""
                for line in predicate_split:
                    predicate += indent + line + "\n"

                # Insert the opaque predicate
                out = out[:pos] + '\n' + predicate + out[pos:]

                pos += len(pred1) + 3
                flag = True

        except IndexError as error:
            # Output expected IndexErrors.
            print("string index out of range")

    # Return out with library random and array of number for opaque predicate
    out = '''from random import randint
from random import SystemRandom
xxnxx = 10
xxgxx = [36,58,1,46,23,5,16,65,2,41,2,7,1,37,0,11,16,2,21,16]\n''' + out

    return out
```

Questa funzione inizializza al suo interno 4 predicati opachi, presi dalla teoria dei numeri e combinati all'array analisi: i predicati opachi verranno valutati su valori contenuti all'interno di un array, questo array poi sia nel ramo *if* che nel ramo *else* del predicato opaco verrà rimescolato, mantenedo

invariate delle proprietà in base alla posizione del numero nell'array, per evitare ad un analizzatore statico di sapere il valore delle variabili. Ovviamente noi sapremo sempre quale ramo (*if* o *else*) verrà preso, per definizione di predicato opaco, però i due blocchi sono semanticamente equivalenti e quindi indistinguibili semanticamente.

In questa funzione i predicati opachi sono visibili all'attacante e questo non va bene perché potranno essere identificati, anche se prima viene eseguita una *Data Obfuscation*. Però, per lo scopo di questo offuscatore, non si poteva fare altrimenti, perché è un offuscatore che non è stato concepito per un unico programma ad hoc, ma riesce ad offuscare tutti quei programmi che gli vengono passati come parametro, che rispettano le caratteristiche delle nostre assunzioni.

La potenza di questa funzione è che si basa tutta sul fattore random: sia per i predicati opachi scelti sia per dove inserirli.

Viene scelto un numero random da 1 a 4 che corrisponderà al predicato opaco da inserire (sono numerati da 1 a 4). Dopodichè verrà scelta sempre a random la posizione in cui inserire questo predicato opaco nei primi `n/2` caratteri del programma e poi verrà scelto di inserire un secondo predicato opaco nei secondi `n/2` caratteri del programma. Prima dell'inserimenti vi è un controllo che posizionerà il preidcato opaco all'inizio di una nuova riga e non in mezzo ad una riga già presente, altrimenti il programma offuscato non funzionerà.

Essendo Python un linguaggio posizionale, verrà considerata l'indentazione di dove verrà inserire il predicato contando il numero di "tab" alla riga precedente.

Quando entrambi i predicati opachi sono stati inseriti correttamente nel programma verrà inserito all'inizio del programma i moduli *randint* e *SystemRandom*, una variabile il cui contenuto è 10 e l'array, altrimenti non sarebbe possibile nel programma offuscato eseguire i predicati opachi.

3.2.3 *Codice di Data Obfuscation*

encoding_literal_data()

La funzione che abbiamo creato per Data Obfuscation è `encoding_literal_data()`. Questa funzione ha l'obiettivo di codificare il valore delle variabili, se di tipo intere o liste tramite la tecnica dell'*Array Permutation*, se stringhe tramite la trasformazione verso una **Macchina di Mealy**.

La funzione definisce subito il codice per decodificare la macchina di Mealy all'interno del programma offuscato. La variabile stringa verrà trasformata in una macchina di mealy defindendo gli stati, tutte le transizioni possibili. La decodifica richiama la funzione passando come parametro gli stati di decodifica corretti. Poi viene definito il codice per fare l'encoding della permutazine degli array nel caso di variabili intere in modo tale da trasformare un array di "cifre" in un numero mantenendo così corretta la semantica del programma iniziale.

Dopo queste due operazioni viene ritornato il codice 'sorgente' con l'offuscamento dell'array permutation, la stringa con la macchina di Mealy e il numero con la permutazione. Viene scansionato riga per riga l'albero he contiene la parserizzazione del codice per trovare gli "oggetti" che ci interessano. E' implementato anche il calcolo dei "tab" per mantenere l'indentazione corretta del codice.

```
def encoding_literal_data(out):

    # Code for Mealy decoding
    class_mealy = "#!/usr/bin/python\n\
# −∗− coding: utf−8 −∗−\n\n\
from random import randint\n\
from random import SystemRandom\n\
class xxxxxMxxxxx(object):\n\
    def __init__(self, some_states, a_input_alphabet, a_output_alphabet,
     some_transitions, a_initial_state):\n\
        self.xxsxxtxxaxxtxxexxsxx = some_states\n\
        self.xxixxnxxpxxuxxtxx_xxaxxlxxpxxhxxaxxbxxexxtxx =
            a_input_alphabet\n\
        self.xxoxxuxxtxxpxxuxxtxx_xxaxxlxxpxxhxxaxxbxxexxtxx =
            a_output_alphabet\n\
        self.xxtxxrxxaxxnxxsxxixxtxxixxoxxnxxsxx = some_transitions\n\
        self.xxixxnxxixxtxxixxaxxlxx_xxsxxtxxaxxtxxexx = a_initial_state\n\
    def get_output_from_string(self, string):\n\
        temp_list = list(string)\n\
        current_state = self.xxixxnxxixxtxxixxaxxlxx_xxsxxtxxaxxtxxexx\n\
        output = ''\n\
        for x in temp_list:\n\
            output +=
                chr(self.xxtxxrxxaxxnxxsxxixxtxxixxoxxnxxsxx[current_state][x][1]
                )\n\
            current_state =
                self.xxtxxrxxaxxnxxsxxixxtxxixxoxxnxxsxx[current_state][x][0]\n\
        return output\n\n"

    # Code for permutation encoding
    def_encoding_integer = "class xxxxxExxxxx():\n\
    def integer(self,digits):\n\
        num = 0\n\
        moltiplicatore = 1\n\
        for xxxxxDxxxxx in digits:\n\
            num += xxxxxDxxxxx∗moltiplicatore\n\
            moltiplicatore ∗= 10\n\
        return num\n\n"

    """
        Returns 'source' with obfuscation of array with permutation,
        string with mealy machine and number with permutation.
    """

    print("−> obfuscating variables: str (mealy), int (permutation),
        array (permutation) ")

    # Tree object contains the AST of the code
    tree = ast.parse(out)
    tree_split = astor.to_source(tree).split("\n")
    tree = ""

    # Indent tab
    indent = ""
    tab = " "

    # For each row in code
    for subtree in tree_split:
        variables = {}
        values = {}
        new_values = {}

        # Count tab indent position
        indent = ""

        try:
            while True:
                if " " in subtree[0:4]:
                    subtree = subtree[4:len(subtree)]
```

```
                indent += tab
            else:
                break
        subtree = ast.parse(subtree)

    except:
        tree += indent + subtree + "\n"
        continue

    # Extraction of all variables defined
    variables, values = get_variables_values_Literal(subtree)

    # If string:
    if isAssignStr(subtree):

        # start mealy_machine and save new value
        for value in values:
            new_values[value], codice = mealy_machine(values[value])

        # Renomination of all variable occurrences
        subtree = rename_values_Literal(subtree, new_values)

        for variable in variables:
            variable = variable

        # Decode function for mealy machine
        tree += indent + astor.to_source(subtree) + indent + variable + " =
            eval(" + variable + ").get_output_from_string('%s')" % codice +
            "\n"

    # If integer:
    elif isAssignInt(subtree):

        # Encoding integer
        for value in values:
            num = values[value]

            # Extract the digits of number
            digits = []

            if num == 0:
                digits.append(num)

            while num != 0:
                digit = num % 10
                num = int(num / 10)
                digits.append(digit)

            new_values[value] = digits

        for variable in variables:
            variable = variable

        # Create permutation of digits
        a = new_values[variable]
        b, p = array_permutation(a)

        # Decode function for permutation digits
        tree += indent + variable + " = " + str(b) + "\n"
        tree += indent + variable + "_ooo = " + str(p) + "\n"
        tree += indent + variable + " =
            " + "xxxxxExxxxx().integer([ " + variable + "[" + variable +
            "_ooo" + "[xxxxxxxxxxxx]] for xxxxxxxxxxxx in range(0, len(" +
            variable + ")) ])" + "\n"

    # If array:
    elif isAssignList(subtree):

        # Encoding list
        for value in values:
```

```
                a = values[value]

            for variable in variables:
                variable = variable

            # Create permutation of position array
            b, p = array_permutation(a)

            # Decode funcyion for permutation position
            tree += indent + variable + " = " + str(b) + "\n"
            tree += indent + variable + "_ooo = " + str(p) + "\n"
            tree += indent + variable + " =
                " + "[ " + variable + "[" + variable + "_ooo" + "[xxxxxxxxxxx]]
                for xxxxxxxxxxx in range(0, len(" + variable + ")) ]" + "\n"

        # Else:
        else:

            tree += indent + astor.to_source(subtree)

    # Return definition class Mealy, function encoding for permutation and tree
    return class_mealy + def_encoding_integer + tree
```

La funzione `def get_variables_values_Literal()` prende in input un abstract sintax tree e ritorna in output due dizionari. Uno che contiene come chiave il nome della variabile e come valore il tipo della variabile, l'altro come chiave il nome della varibile e come valore il valore vero e proprio della variabile. I valori vengono estratti dall'abstract tree se di tipo stringa, intera e se sono variabili o liste. E' stata definita una classe *MyVisitor()* per visitare i nodi ed estratte i valori delle variabili attraverso le funzioni *visit()* implementate tipo per tipo.

```
def get_variables_values_Literal():
    """
    :param tree:
    :return: a dictionary contains each variable (key) and its type (value)
    contains in the code (passed as AST 'tree')
    """
    variables = {}
    values = {}

    for node in ast.walk(tree):
        # Saved variables declared
        if isinstance(node, ast.Assign) and isinstance(node.targets[0], ast.Name):

            if type(node.value).__name__ == "Str":
                variables[node.targets[0].id] = type(node.value).__name__
                values[node.targets[0].id] = ""

                MyVisitor(values, node.targets[0].id).visit(node)
                values = MyVisitor(values, node.targets[0].id).get_values(node)

            elif type(node.value).__name__ == "Num":
                variables[node.targets[0].id] = type(node.value).__name__
                values[node.targets[0].id] = 0

                MyVisitor(values, node.targets[0].id).visit(node)
                values = MyVisitor(values, node.targets[0].id).get_values(node)

            elif type(node.value).__name__ == "List":
                variables[node.targets[0].id] = type(node.value).__name__
                values[node.targets[0].id] = []

                MyVisitor(values, node.targets[0].id).visit(node)
                values = MyVisitor(values, node.targets[0].id).get_values(node)
```

```
            else:
                variables[node.targets[0].id] = type(node.value).__name__

    return variables, values
```

La funzione `isAssignStr()` prende in input un abstract sintax tree e ritorna un valore booleano (*True/False*) se la variabile è di tipo stringa.

```
def isAssignStr(tree):
    values = {}

    for node in ast.walk(tree):
        # Saved variables declared
        if isinstance(node, ast.Assign) and isinstance(node.targets[0], ast.Name):
            if type(node.value).__name__ == "Str":
                return True

    return False
```

La funzione `isAssignInt()` rende in input un abstract sintax tree e ritorna un valore booleano (*True/False*) se la variabile è di tipo intero.

```
def isAssignInt(tree):
    values = {}

    for node in ast.walk(tree):
        # Saved variables declared
        if isinstance(node, ast.Assign) and isinstance(node.targets[0], ast.Name):
            if type(node.value).__name__ == "Num":
                return True

    return False
```

La funzione `rename_values_Literal` prende in input un abstract sintax tree e un array di valori. L'arrai di valory contiene le nuove definifioni delle variaibli, per esempio la nuova struttura delle stringhe in macchine di mealy. Ritorna in output l'abstract sintax tree con i valori delle variabili stringhe trasformati in macchine di mealy.

```
def rename_values_Literal(tree, values):
    # Rename values of variables used (general usage)
    for value in values:
        for node in ast.walk(tree):
            if isinstance(node, ast.Assign) and
            isinstance(node.targets[0], ast.Name):
                MyModifyValue(values[value]).visit(node)

    return tree
```

La funzione `isAssignList()` prende in input un abstract sintax tree e ritorna un valore booleano (*True/False*) se la variabile è di tipo array.

```
def isAssignList(tree):
    values = {}

    for node in ast.walk(tree):
        # Saved variables declared
        if isinstance(node, ast.Assign) and isinstance(node.targets[0], ast.Name):
```

```
            if type(node.value).__name__ == "List":
                return True

    return False
```

La funzione `mealy_machine()` prende in input una astringa e ritorna la macchina di mealy corrispondente. L'algoritmo prevede di trattare la stringa carattere per carattere e di inserirli come stati nella macchina. Per creare più confusione e rallentare il reverse engineering ogni carattere viene codificato nel suo corrispondente codice ascii.

```
def mealy_machine(string_input):
    len_string_input = len(string_input)

    # Create states
    iIdx = 0
    states = []
    for c in string_input:
        states.append('q'+str(iIdx))
        iIdx+=1

    # Create transitions
    new_string_input = ''
    iIdx = 0
    transitions = {}
    for q in states:
        stato1 = random.randint(0,1)
        if stato1 == 0:
            stato2 = 1
        else:
            stato2 = 0
        new_string_input += str(stato1)
        char_random = random.randint(97,122)
        if (iIdx+1) == len_string_input:
            transitions.update({q:{str(stato1) :
            ('q0', ord(string_input[iIdx])), str(stato2) :
            ('q'+str(iIdx-1), char_random)}})
        elif (iIdx == 0):
            transitions.update({q:{str(stato1) :
            ('q'+str(iIdx+1), ord(string_input[iIdx])), str(stato2) :
            ('q'+str(len_string_input-1), char_random)}})
        else:
            transitions.update({q:{str(stato1) :
            ('q'+str(iIdx+1), ord(string_input[iIdx])), str(stato2) :
            ('q'+str(iIdx-1), char_random)}})
        iIdx+=1

    return "xxxxxMxxxxx({},['0', '1'],['0', '1'],{},'q0')".format(states,transitions),
        new_string_input
```

La funzione `array_permutation()` prende in input un array e ritorna due array che sono la permutazione di quello passatogli come input. In realtà un array conterrà la permutazione, l'altro le posizioni per ricostruire quello originale.

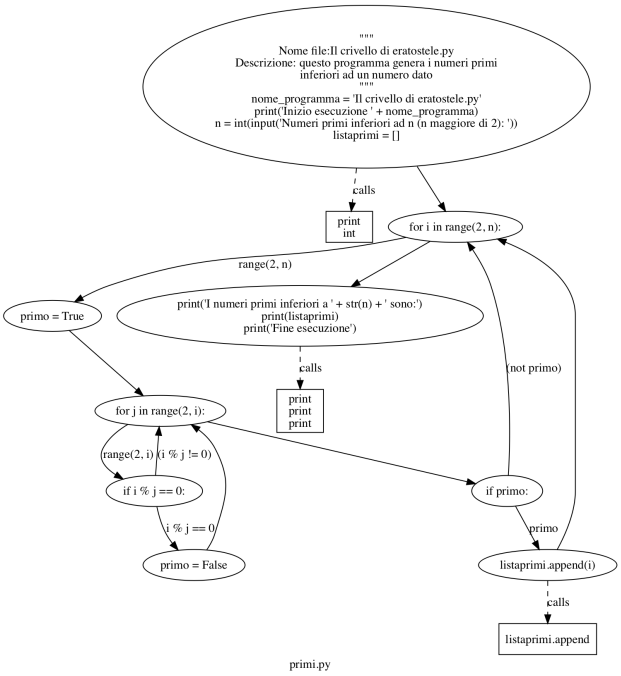
```
def array_permutation(a):
    i = 1
    divisori = []
    while i <= len(a):
        if len(a) % i == 0:
            divisori.append(i)
        i += 1
```

```
n = random.randint(1, 3000)
i = 1
divisori_n = []
while i <= n:
    if n % i == 0:
        divisori_n.append(i)
    i += 1

x = 0
while x < len(divisori):
    div = divisori[x]

    if div in divisori_n and div != 1:
        n = random.randint(1, 3000)
        i = 1
        divisori_n = []
        while i <= n:
            if n % i == 0:
                divisori_n.append(i)
            i += 1
        x = 0
    else:
        x += 1

b = [ x*0 for x in range(0, len(a)) ]
p = [ ((i-1)*n)%len(a) for i in range(0, len(a)) ]

for i in range(0, len(a)):
    b[p[i]] = a[i]

return b, p
```

4 ESEMPIO DI FUNZIONAMENTO

Prendiamo come esempio il crivello di eratostele che calcolo i numeri primi.

```
#!/usr/bin/python
# −∗− coding: utf−8 −∗−

'''
Nome file:Il crivello di eratostele.py
Descrizione: questo programma genera i numeri primi
inferiori ad un numero dato
'''

nome_programma="Il crivello di eratostele.py"
print("Inizio esecuzione " + nome_programma)

n=int(input("Numeri primi inferiori ad n (n maggiore di 2): "))
listaprimi=[]
for i in range(2,n): # analizzo tutti i numeri i tra 2 ed n
    primo=True # controllo se i e' primo
    for j in range(2,i):
        if i%j==0:
            primo=False # i non e' primo
    if primo: # i e' primo, quindi lo aggiungo alla lista dei primi
            listaprimi.append(i)

print("I numeri primi inferiori a " + str(n) + " sono:")
print(listaprimi)

print("Fine esecuzione")
```



primi.py

Lanciamo il nostro offuscatore:

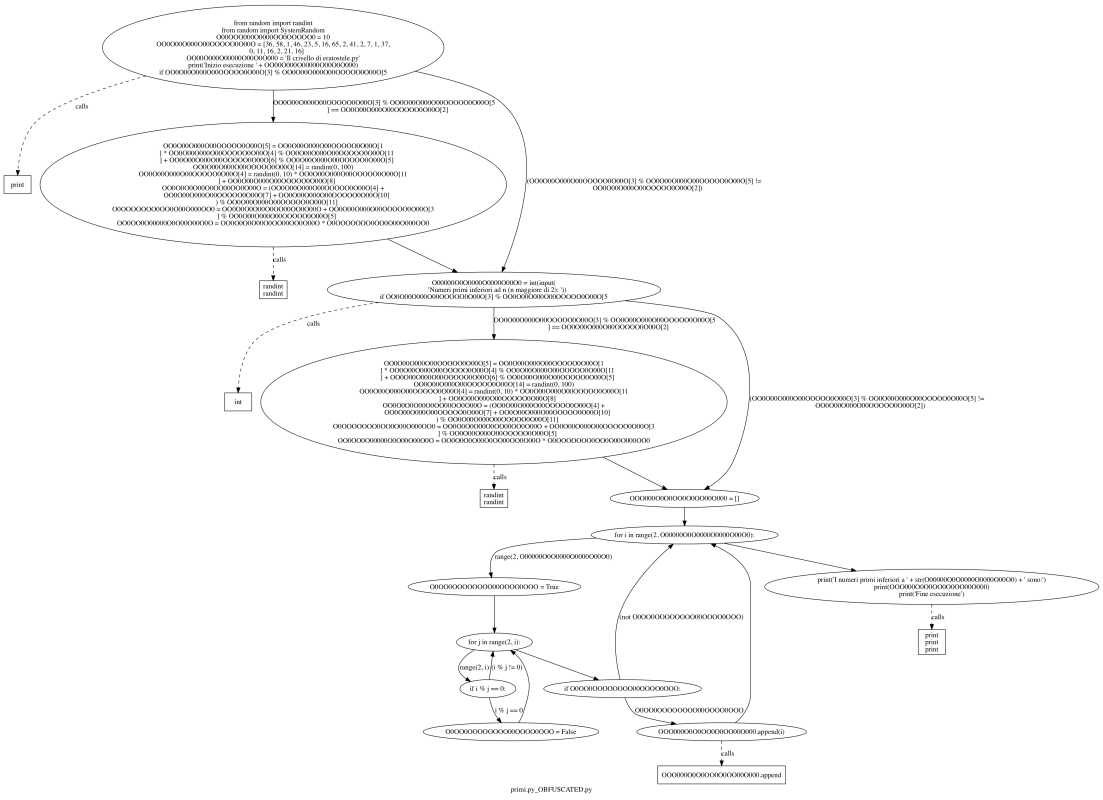
```
python3  offuscatore.py  primi.py
```

E otteniamo questo risultato:

```
from random import randint
from random import SystemRandom
OooOOOoooOooooOOoOOOOOo = 10
OOoOooOoooOooOOOOOOoOooO = [36, 58, 1, 46, 23, 5, 16, 65, 2, 41,
                            2, 7, 1, 37, 0, 11, 16, 2, 21, 16]
OOooOoooOooooooOooOoOooo = 'Il crivello di eratostele.py'
print('Inizio esecuzione ' + OOooOoooOooooooOooOoOooo)

if OOoOooOoooOooOOOOOOoOooO[3] %
   OOoOooOoooOooOOOOOOoOooO[5] ==
   OOoOooOoooOooOOOOOOoOooO[2]:

      OOoOooOoooOooOOOOOOoOooO[5] =
      OOoOooOoooOooOOOOOOoOooO[1] *
      OOoOooOoooOooOOOOOOoOooO[4] %
      OOoOooOoooOooOOOOOOoOooO[11] +
      OOoOooOoooOooOOOOOOoOooO[6] %
      OOoOooOoooOooOOOOOOoOooO[5]

      OOoOooOoooOooOOOOOOoOooO[14] = randint(0, 100)

      OOoOooOoooOooOOOOOOoOooO[4] = randint(0, 10) *
      OOoOooOoooOooOOOOOOoOooO[11] +
      OOoOooOoooOooOOOOOOoOooO[8]
      OOoOoOoOooOoOOooOOoOooO =
      (OOoOooOoooOooOOOOOOoOooO[4] +
      OOoOooOoooOooOOOOOOoOooO[7] +
      OOoOooOoooOooOOOOOOoOooO[10]) %
      OOoOooOoooOooOOOOOOoOooO[11]

      OoOOOOOOOoOOoOooOoooOOo =
      (OOoOoOoOooOoOOooOOoOooO +
      OOoOooOoooOooOOOOOOoOooO[3] %
      OOoOooOoooOooOOOOOOoOooO[5])

      OOoOOoOooooooOoOooOooOoO =
      OOoOoOoOooOoOOooOOoOooO *
      OoOOOOOOOoOOoOooOoooOOo

OooooooOoOoooooOooooOooOo = int(input(
      'Numeri primi inferiori ad n (n maggiore di 2): '))

if OOoOooOoooOooOOOOOOoOooO[3] %
   OOoOooOoooOooOOOOOOoOooO[5] ==
   OOoOooOoooOooOOOOOOoOooO[2]:

      OOoOooOoooOooOOOOOOoOooO[5] =
      OOoOooOoooOooOOOOOOoOooO[1] *
      OOoOooOoooOooOOOOOOoOooO[4] %
      OOoOooOoooOooOOOOOOoOooO[11] +
      OOoOooOoooOooOOOOOOoOooO[6] %
      OOoOooOoooOooOOOOOOoOooO[5]
```

```
        OOoOooOoooOooOOOOOoOooO[14] = randint(0, 100)

        OOoOooOoooOooOOOOOoOooO[4] = randint(0, 10) *
        OOoOooOoooOooOOOOOoOooO[11] +
        OOoOooOoooOooOOOOOoOooO[8]

        OOoOoOoOooOoOOooOOoOooO =
        (OOoOooOoooOooOOOOOoOooO[4] +
        OOoOooOoooOooOOOOOoOooO[7] +
        OOoOooOoooOooOOOOOoOooO[10]) %
        OOoOooOoooOooOOOOOoOooO[11]

        OoOOOOOOOoOOoOooOoooOOo =
        (OOoOoOoOooOoOOooOOoOooO +
        OOoOooOoooOooOOOOOoOooO[3] %
        OOoOooOoooOooOOOOOoOooO[5])

        OOoOOoOooooooOoOooOooOoO =
        OOoOoOoOooOoOOooOOoOooO *
        OoOOOOOOOoOOoOooOoooOOo

    OOOoooOoOoOOoOoOOooOooo = []
    for i in range(2, OooooooOoOooooOooooOooOo):
        OoOOoOOOOOOOOOooOOOOoOOO = True
        for j in range(2, i):
            if i % j == 0:
                OoOOoOOOOOOOOOooOOOOoOOO = False
        if OoOOoOOOOOOOOOooOOOOoOOO:
            OOOoooOoOoOOoOoOOooOooo.append(i)

    print('I numeri primi inferiori a '
      + str(OooooooOoOooooOooooOooOo)
      + ' sono:')

    print(OOOoooOoOoOOoOoOOooOooo)
    print('Fine esecuzione')
```

Se eseguiamo i due codici la semantica viene mantenuta. La struttura del control flow graph esplode.


from random import randint
from random import SystemRandom
OOOOOOOOOOOOOOOOOOOOOOOO = 10
OOOOOOOOOOOOOOOOOOOOOOOO = [36, 58, 1, 46, 23, 5, 16, 65, 2, 41, 2, 7, 1, 37,
0, 11, 16, 2, 21, 16]
OOOOOOOOOOOOOOOOOOOOOOOO = 'Il crivello di eratostele py'
print('Inizio esecuzione ' + OOOOOOOOOOOOOOOOOOOO)
if OOOOOOOOOOOOOOOOOOOOOOOO[3] % OOOOOOOOOOOOOOOOOOOOOOOO[5]
calls
print
OOOOOOOOOOOOOOOOOOOOOOOO[3] % OOOOOOOOOOOOOOOOOOOOOOOO[5]
!= OOOOOOOOOOOOOOOOOOOOOOOO[2]
OOOOOOOOOOOOOOOOOOOOOOOO[5] = OOOOOOOOOOOOOOOOOOOOOOOO[1
] * OOOOOOOOOOOOOOOOOOOOOOOO[4] % OOOOOOOOOOOOOOOOOOOOOOOO[11
] + OOOOOOOOOOOOOOOOOOOOOOOO[6] % OOOOOOOOOOOOOOOOOOOOOOOO[5]
OOOOOOOOOOOOOOOOOOOOOOOO[14] = randint(0, 100)
OOOOOOOOOOOOOOOOOOOOOOOO[4] = randint(0, 10) * OOOOOOOOOOOOOOOOOOOOOOOO[11
] + OOOOOOOOOOOOOOOOOOOOOOOO[8]
OOOOOOOOOOOOOOOOOOOOOOOO = (OOOOOOOOOOOOOOOOOOOOOOOO[4] +
OOOOOOOOOOOOOOOOOOOOOOOO[7] + OOOOOOOOOOOOOOOOOOOOOOOO[10]
) % OOOOOOOOOOOOOOOOOOOOOOOO[11]
OOOOOOOOOOOOOOOOOOOOOOOO = OOOOOOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOOOOOO[3
] % OOOOOOOOOOOOOOOOOOOOOOOO[5]
OOOOOOOOOOOOOOOOOOOOOOOO = OOOOOOOOOOOOOOOOOOOOOOOO * OOOOOOOOOOOOOOOOOOOOOOOO
calls
randint
randint
OOOOOOOOOOOOOOOOOOOO = int(input(
'Numeri primi inferiori ad n (n maggiore di 2): '))
if OOOOOOOOOOOOOOOOOOOO[3] % OOOOOOOOOOOOOOOOOOOO[5
calls
int
OOOOOOOOOOOOOOOOOOOO[3] % OOOOOOOOOOOOOOOOOOOO[5
!= OOOOOOOOOOOOOOOOOOOO[2]
OOOOOOOOOOOOOOOOOOOO[5] = OOOOOOOOOOOOOOOOOOOO[1
] * OOOOOOOOOOOOOOOOOOOO[4] % OOOOOOOOOOOOOOOOOOOO[11
] + OOOOOOOOOOOOOOOOOOOO[6] % OOOOOOOOOOOOOOOOOOOO[5]
OOOOOOOOOOOOOOOOOOOO[14] = randint(0, 100)
OOOOOOOOOOOOOOOOOOOO[4] = randint(0, 10) * OOOOOOOOOOOOOOOOOOOO[11
] + OOOOOOOOOOOOOOOOOOOO[8]
OOOOOOOOOOOOOOOOOOOO = (OOOOOOOOOOOOOOOOOOOO[4] +
OOOOOOOOOOOOOOOOOOOO[7] + OOOOOOOOOOOOOOOOOOOO[10]
) % OOOOOOOOOOOOOOOOOOOO[11]
OOOOOOOOOOOOOOOOOOOO = OOOOOOOOOOOOOOOOOOOO + OOOOOOOOOOOOOOOOOOOO[3
] % OOOOOOOOOOOOOOOOOOOO[5]
OOOOOOOOOOOOOOOOOOOO = OOOOOOOOOOOOOOOOOOOO * OOOOOOOOOOOOOOOOOOOO
calls
randint
randint
OOOOOOOOOOOOOOOOOOOO = []
for i in range(2, OOOOOOOOOOOOOOOOOOOO):
range(2, OOOOOOOOOOOOOOOOOOOO)
OOOOOOOOOOOOOOOOOOOO = True
print('I numeri primi inferiori a ' + str(OOOOOOOOOOOOOOOOOOOO) + ' sono:')
print(OOOOOOOOOOOOOOOOOOOO)
print('Fine esecuzione')
calls
print
print
print
for j in range(2, i):
range(2, i)
if i % j == 0:
i % j != 0
i % j == 0
OOOOOOOOOOOOOOOOOOOO = False
if OOOOOOOOOOOOOOOOOOOO:
not OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO.append(i)
calls
OOOOOOOOOOOOOOOOOOOO.append
primi.py_OBFUSCATED.py