УДК 681.391

В. А. Пласковицкий, аспирант (БГТУ);
П. П. Урбанович, доктор технических наук, профессор, заведующий кафедрой (БГТУ)

ИСПОЛЬЗОВАНИЕ АБСТРАКТНЫХ СИНТАКСИЧЕСКИХ ДЕРЕВЬЕВ ДЛЯ ОБФУСКАЦИИ КОДА

Рассмотрено практическое использование абстрактных синтаксических деревьев для обфускации программного кода на примере языка Python. Проанализированы возможности изменения строк, чисел, функций, имен объектов, операторов. Проведен анализ механизма обработки кода с замером скорости выполнения преобразования в зависимости от типа обрабатываемых конструкций. Исследован этап обратной сборки программы на основе полученного дерева. Рассмотрена интеграция полученного обработчика с авторскими алгоритмами обфускации кода.

This paper considers the practical application of abstract syntax trees to obfuscate the example code in Python. The possibility of changing strings, numbers, functions, names of objects, operators are studied. The analysis of the mechanism for handling code execution speed measurements are made, depending on the type of processed structures. Studied reassembly step program based on the resulting tree. Considered integration with the author obtained handler code obfuscation algorithms.

Введение. Из-за хранения кода программных продуктов в сравнительно удобочитаемом виде несанкционированное применение их значительно упрощается. При этом зачастую обфускация является единственным механизмом защиты, поскольку сохраняет работоспособность программы без необходимости применения дополнительных узлов выполнения, но при этом делает чтение кода более трудоемким.

Однако помимо сложности создания алгоритмов обфускации и их оценки, возникает нетривиальная задача и при получении самих данных, которыми эти алгоритмы будут манипулировать. Синтаксис современных программных языков значительно расширился (лямбда выражения, списковые включения, декораторы, стрелочные функции и многое другое), а архитектура составляемых приложений стала сложней (mvc, mvvm, mvp и др.). Синтаксис языков программирования часто неоднозначен по своей природе. Для того чтобы избежать этой неоднозначности он часто задается в виде контекстно-свободной грамматики (CFG) [1]. Тем не менее есть аспекты языков программирования, которые не могут быть выражены через CFG, но являются частью языка и задокументированы в его спецификации. В этом случае необходим контекст, чтобы определить их обоснованность и поведение. Например, CFG не может предсказать ни имена новых типов, объявленных в программе, ни то, каким образом они должны быть использованы. Даже если в языке есть предопределенный набор типов, для обеспечения надлежащего использования, как правило, требуется некоторый контекст. Другим примером является «утиная» типизация, где тип элемента может меняться в зависимости от контекста. Перегрузка операторов – еще один случай, когда правильное использование и окончательная функция определяются в зависимости от контекста. Например, оператор «+» является и численным сложением, и конкатенацией строк.

Все это делает неприемлемым непосредственную работу с текстовым представлением кода. В данной статье рассмотрен способ обфускации кода, представленного в виде абстрактного синтаксического дерева, а также способы его преобразования к такому виду и обратно.

Основная часть. В абстрактных синтаксических деревьях (АСД) в качестве внутренних вершин выступают операторы языка программирования, а в качестве листьев – передаваемые в них операнды. При этом в отличие от дерева разбора в АСД не входят элементы, которые не влияют на семантику программы, например скобки, точки с запятой и т. п., поскольку информацию о разделении несет сама организация узлов в дереве.

Например, для описания выражения $x = 3 \times \times (y - z)$ можно построить АСД, представленное на рис. 1.

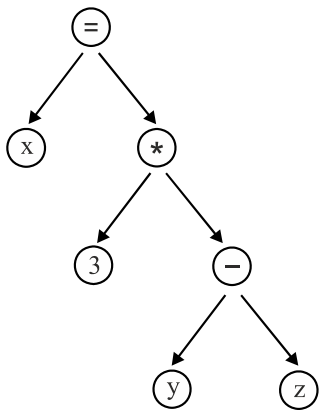

Рис. 1. Пример АСД

Это весьма упрощенное представление, поскольку описывает математическое выражение. В случае работы с программным кодом, необходимо учитывать еще множество других параметров: типы переменных, условия перехода и т. д. Например, даже для описания весьма простой php-конструкции:

```
<?php
   $x = 5;
   if($x == 5){
      echo "yes";
   }
   else{
      echo "no";
   }
?>
```

строится довольно громоздкое АСД (рис. 2) [2].

Графическое представление АСД упрощает его восприятие для человека, но не подходит для использования при программной обработке кода, поэтому для этой цели эффективнее работать с ним, как с ассоциативным массивом.

Пример:

```
var x = "IsiT";
console.log(x);
function test(){
   x = "BSTU";
   var y = 2
}
```

Преобразуется к виду [3]:

```
[VarDecls(
   [VarDeclInit("x", String("ISiT"))]),
    Call(PropAccess(Var("console"),"log"),
    [Var("x")]),]
    Function("test",
             [],
             [Assign(Var("x"),
              String("BSTU")),
              VarDecls([VarDeclInit
                        ("y",Num("2"))])])
```

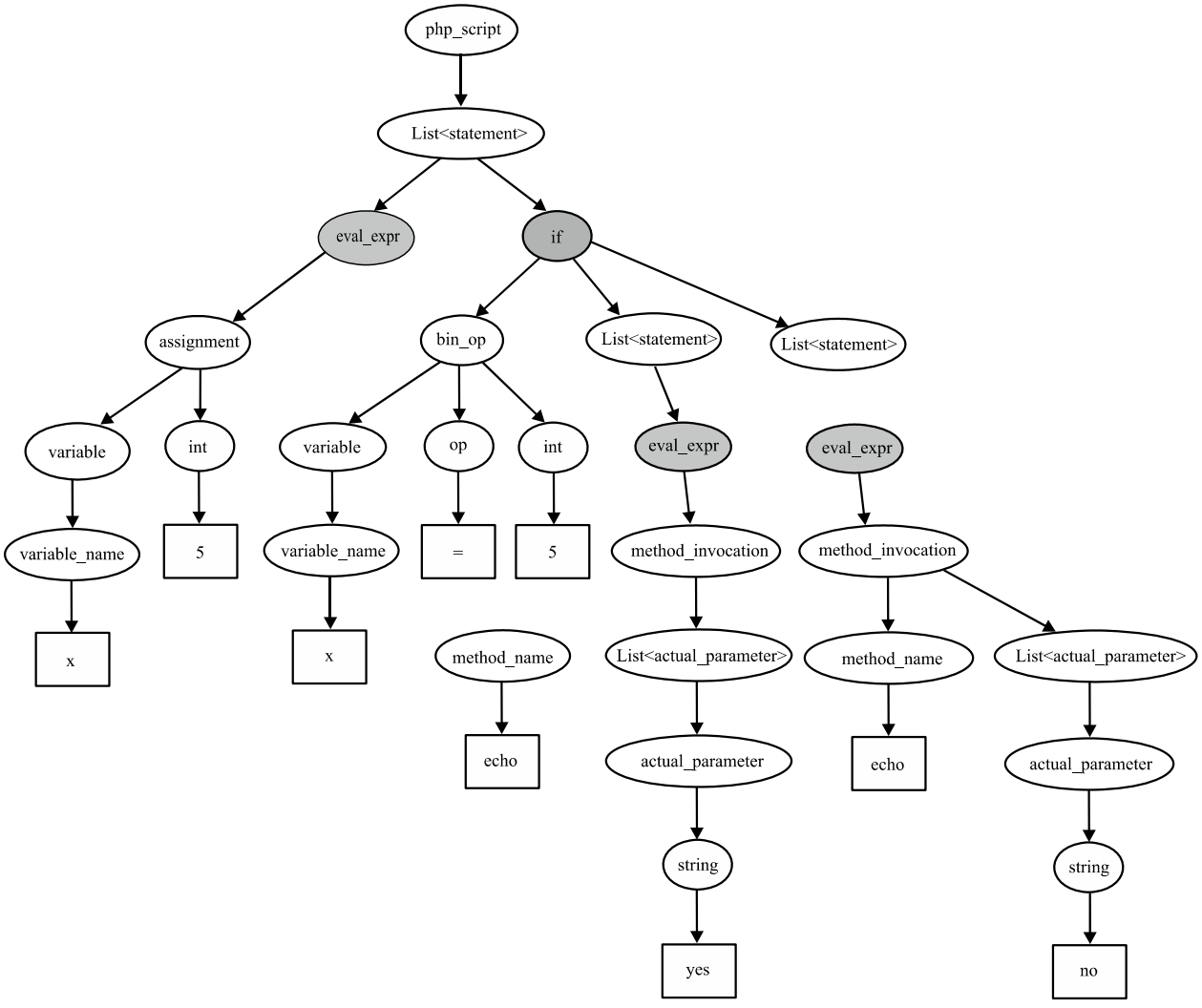

Рис. 2. АСД для php-конструкции

Однако при этом возникает необходимость разрабатывать алгоритм обхода такой структуры. Поэтому более простое решение заключается в использовании паттерна Visitor (рис. 3), позволяющего разработать отдельные методы обработки для нужных сущностей.

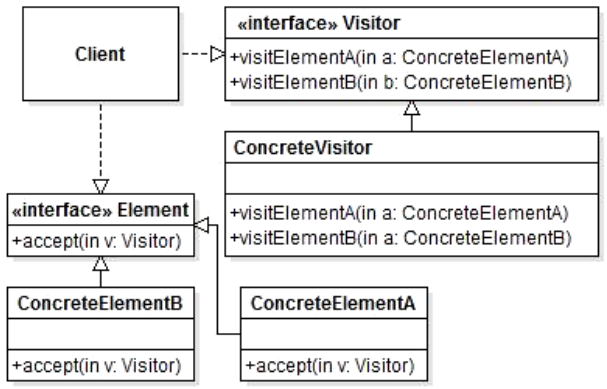

Рис. 3. Схема паттерна Visitor

Рассмотрим реализацию АСД в Python. Благодаря системному модулю ast [3] нет необходимости подключать стороннюю реализацию. Достаточно создать класс, наследуемый от класса NodeTransformer и реализовать в нем соответствующие методы.

Например, для обработки чисел в коде, можно реализовать обфускатор следующего вида:

```
from ast import NodeTransformer, BinOp, Num, Add

class Obfuscator(NodeTransformer):
  def __init__(self):
    ast.NodeTransformer.__init__(self)
    def visit_Num(self, node):
        value = node.n
        value_left = value_right = value / 2
        if value % 2:
            value_right += 1
    return     BinOp(left=BinOp(Num(value_left)),
op=Add(), right=Num(value_right))
```

Результат обработки:

```
1 = 0 + 1;
2 = 1 + 1;
5 = 2 + 3;
10 = 5 + 5.
```

Более сложным является обработка таких сущностей, как функции. В этом случае реализация может выглядеть так:

```
    def visit_FunctionDef(self, node):
        node.name =
self.obfuscate_function_name(node.name)
        node.body = [self.visit(x) for x in
node.body]
        return node
```

Здесь obfuscate_function_name – метод, инкапсулирующий какой-то набор действий по обработке строки, учитывающий ограничения, налагаемые на имя функции.

Обработка условного оператора может быть реализована так:

```
def visit_If(self, node):
  node.test = BoolOp(op=And(),
                     values=[node.test,
                         Compare(left=Num(1),
                                 ops=[Gt()],
                          comparators=[Num(0)])])
  node.test = self.visit(node.test)
  return node
```

Для примера обработки строк были использованы две операции – расщепление строки на две составляющие и реверс строки [4] с последующим считыванием в обратном порядке:

```
def visit_Str(self, node):
  def rev(x):
    return Subscript(value=Str(s=x[::-1]),
              slice=Slice(lower=None,
                          upper=None,
                          step=Num(n=-1)),
                          ctx=Load())
    return BinOp(left=rev(node.s[:len(node.s)/2]),
                 op=Add(),
                 right=rev(node.s[len(node.s)/2:]))
```

Чтобы просмотреть все уникальные типы узлов, присутствующие в коде, можно реализовать класс с обобщенным методом generic_visit:

```
    class LookNodes(ast.NodeVisitor):
    ...  def generic_visit(self, node):
    ...   print type(node).__name__
    ...   ast.NodeVisitor.generic_visit(self, node)
```

и применять его следующим образом:

```
    look = LookNodes()
    tree = ast.parse(str_code)
    look.visit(tree)
```

При этом, если в коде встречаются одинаковые по типу узлы, в консоль будет выведен каждый из них. Для того чтобы выводились только уникальные значения, можно заносить их в общий список типа set:

```
  def __init__(self):
    self.names = set()
  def generic_visit(self, node):
    self.names.add(type(node).__name__)
```

Вывести все дерево можно с помощью метода ast.dump.

Для того чтобы видоизменить операцию подключения модулей, можно получить данные о их названиях, перекрыв метод visit_Import, а

затем, добавить их в условном операторе, проверяющем наличие точки выхода.

Пример обработки строк с использованием двух операций – расщепление строки на составляющие и реверс строки с последующим считыванием в обратном порядке:

```
def visit_Str(self, node):
   def rev(x):
      return Subscript(value=Str(s=x[::-1]),
                  slice=Slice(lower=None,
                              upper=None,
                              step=Num(n=-1)),
                              ctx=Load())
      return BinOp(left=rev(node.s[:len(node.s)/2]),
                   op=Add(),
                   right=rev(node.s[len(node.s)/2:]))
```

Аналогичным образом можно обрабатывать и другие сущности: названия объектов (Name), классы (ClassDef), кортеджи (Tuple), циклы (For, While) и другие [3].

Общий вид преобразования кода с помощью АСД:

```
code = open(path, 'rb').read()
ast = ast.parse(code)
obf = Obfuscator()
ast2 = obf.visit(ast)
result = codegen.to_source(ast2)
open(path, 'w').write(result)
```

Для обратного преобразования АСД используется библиотека codegen, изменяющая узлы дерева в соответствующие фрагменты кода [5].

Чтобы оценить затраты на обработку кода с помощью АСД, был проведен ряд тестов на кодах, содержащих от 100 до 100 000 узлов определенных типов. Результаты отображены в табл. 1–4.

Таблица 1

Время обработки узлов Num

Количество узлов	ast	visit	codegen
100	0,006	0,003	0,005
1 000	0,016	0,022	0,041
10 000	0,037	0,227	0,413
100 000	0,409	3,838	4,947

Таблица 2

Время обработки узлов Name

Количество узлов	ast	visit	codegen
100	0,001	0,009	0,004
10 00	0,007	0,083	0,045
10 000	0,080	0,806	0,450
100 000	1,167	8,046	4,490

Таблица 3

Время обработки узлов Str

Количество узлов	ast	visit	codegen
100	0,000	0,004	0,018
1 000	0,002	0,035	0,126
10 000	0,031	0,432	1,237
100 000	0,263	4,927	12,327

Таблица 4

Время обработки узлов If

Количество узлов	ast	visit	codegen
100	0,002	0,002	0,018
1 000	0,020	0,022	0,187
10 000	0,273	0,312	2,305
100 000	3,086	4,019	18,463

Наибольшее время уходит на генерацию кода на основе АСД, это объясняется тем, что модуль codegen является сторонней разработкой на чистом Python, в то время как модуль ast – внутренний с оптимизацией на Си. Вместо формирования кода из АСД с помощью codegen, можно использовать непосредственную компиляцию через метод compile:

```
s = compile(tree, filename="<ast>",
                  mode="eval")
```

либо при задаче непосредственного выполнения:

```
exec(compile(tree, filename="<ast>"
                   mode="exec"))
```

Время выполнения обфускации сильно зависит от заложенных в нее алгоритмов, но сами затраты на выполнение обработки соответствующих узлов (visit) не значительны.

Используемые тесты:

```
def generate_num(count):
   return [10 for _ in xrange(0, count)]
def generate_if(count):
   code = ["x = 1"]
   for _ in xrange(count):
      code.append("\nif x > 0:\n   x += 1")
   return "\n".join(code)
def generate_str(count):
   return str(["abcd" for _ in xrange(count)])
def generate_name(count):
   code = []
   for i in xrange(count):
      code.append("x{0}={0}".format(i))
   return "\n".join(code)
```

Для простых алгоритмов преобразования не сложно разработать алгоритм деобфускации, использующий то же самое АСД. Например,

можно получить результат сложения строк или чисел, перекрыв оператор BinOp:

```
def visit_BinOp(self, node):
    node = self.generic_visit(node)
    if isinstance(node.left, _ast.Num) and \
       isinstance(node.right, _ast.Num):
       return self.do_eval(node, ast.Num)
    if isinstance(node.left, _ast.Str) and \
       isinstance(node.right, _ast.Str):
       return self.do_eval(node, ast.Str)
return node
```

Подобное действие вернет в исходное состояние код:

```
x = 1 + 2 + 3
x = "1" + "2" + "3"
```

Чтобы нейтрализовать переворачивание строки, можно анализировать вызываемые функции и в случае соответствия операции reversed производить обратное действие

```
def visit_Call(self, node):
    node = self.generic_visit(node)
    if isinstance(node.func, _ast.Name) and
      node.func.id == 'reversed':
      if len(node.args) == 1 and
        isinstance(node.args[0], _ast.Str) and
        not node.keywords and
        not node.starargs and not node.kwargs:
      return ast.Str(node.args[0].s[::-1],
                     lineno=0, col_offset=0)
```

Для того чтобы препятствовать такой деобфускации, можно использовать проход значения через дополнительную функцию:

```
def f(x):
   return x

x = f(1)+f(2)+f(3)
x = f("1")+f("2")+f("3")
```

Для этого достаточно преобразовать аргументы, передаваемые в BinOp к следующему виду:

```
value1 = Call(func=Name(id='f', ctx=Load()),
args=[Str(x)])
value2 = Call(func=Name(id='f', ctx=Load()),
args=[Str(y)])
```

АСД также можно использовать для подсчета метрик программного обеспечения [5]. Например, вести подсчет всех встречающихся блоков с помощью глобального словаря, задав веса в соответствии со сложностью объекта (строковые узлы и числа – 1, операторы – 2, функции – 5). Оценку глубины вложенности функций можно производить за счет рекурсивного вызова visit для содержимого функции.

Заключение. Обфускация кода, представленного в виде АСД, значительно упрощается при реализации обхода узлов через паттерн Visitor. Недостатком при этом является строго заданная схема обхода, что позволяет реализовать только определенные алгоритмы защиты. Возможен обратный процесс восстановления кода с помощью АСД за счет динамического выполнения обходных узлов. Затруднить деобфускацию можно используя более сложные преобразования с вынесением участков кода в отдельные функции, не позволяя тем самым заменять их статическим результатом.

Помимо внедрения алгоритмов обфускации, АСД является удобным инструментом для вычисления метрик программного обеспечения [6] особенно количественного типа.

Литература

1. Russell J., Cohn R. Context-free grammar. M.: VSD, 2013. 120 p.

2. PHC Documentation [Электронный ресурс]. Режим доступа: http://www.phpcompiler.org/doc/phc-0.2,0/devintro.html. Дата доступа: 20.02.2014.

3. Python Abstract Syntax Trees [Электронный ресурс]. Режим доступа: http://docs.python.org/2/library/ast. Дата доступа: 20.02.2014.

4. Python Source Obfuscation using ASTs [Электронный ресурс]. Режим доступа: http://jbremer.org/python-source-obfuscation-using-asts. Дата доступа: 20.02.2014.

5. Extension to ast that allow ast python code generation [Электронный ресурс]. Режим доступа: https://github.com/andreif/codegen. Дата доступа: 20.02.2014.

6. Пласковицкий В. А., Урбанович П. П. Защита программного обеспечения от несанкционированного использования и модификации методами обфускации // Труды БГТУ. 2011. Сер. VI, Физ.-мат. науки и информатика. Вып. XIX. С. 173–176.

Поступила 20.03.2014