Certification of Python Programs on the Basis of Static Information Flow Analysis

A Thesis
submitted in partial fulfillment of the
requirements for the degree of
Master of Technology
by

Abhishek Pratap Singh
143050077

under the guidance of

Prof. R K Shyamsundar

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai - 400076 (India)

July, 2016

Abstract

In this thesis, we present our work on secure information flow analysis of python programs. We have built a platform that takes source code and labels of all objects used in python program as input for static analysis of information flow throughout the program. We started with Denning's lattice model [1] for verification of secure information flow. In this model, every object is associated with its security class. To prevent unauthorized leak of information, the flow of information should be in one way – from less secure to more secure class. A lattice represents such information model very well, the upward direction in lattice represents secure information flow. Verification of information flow only on the basis of security class is not sufficient to certify the security of system, there is a need to consider the process, user or subject that executes the code. Use of Reader Writer Flow model [1] with subjects makes it possible to do secure information flow analysis. We have developed four type of constraint generators C1, C2, C3 and C4 each implementing different approach. Constraint generator C1 considers fixed label and PC reset(PC reset denotes PC is not retaining information out of scope), C2 considers fixed label and monotonic PC (monotonic PC never lose information), C3 considers dynamic label and PC reset and finally the constraint generator C4 considers dynamic label and monotonic PC. Constraint resolver takes these constraints and RWFM labels [2] for each object defined by the user as input and provides answers to various queries related to information flow security.The report describes, the approach, implementation and case study done so far.

Contents

List of Figures

Listings

Chapter 1

Introduction

Language Based Security

Information security is major concern nowadays. Systems are vulnerable to various attacks and exposed to threats via network. There are many approaches to enforce security policies in the system one of them is language based security, this approach focuses on enforcement of security policies on a application using program analysis. There are three main branches in language based security (a) Reference Monitor (b) Type Safe (c) Certifying Compiler this report focuses on certifying compiler, certifying compiler approach is based on program analysis compiler checks whether program follows security policy. Compiler does certification from outside of the system so it matches with principle of security model (i) Principle of least privilege (ii) Minimal computing base [5]. Static analysis is simple for this type of certification because whole process completes in a one go. Dynamic analysis becomes necessary if information flow occurs only at run time. So hybrid approach can cover all type of ananlysis.

Noninterference

One of the earliest formal work in information security is concept of noninterference created by Goguen and Meseguer [6] in the context of MLS(Multi Level Secure). It gave concept to determine information leak in system of multiple users, suppose that there are two group of users A and B in a system S and if user of group A interacts with System S then view of users of B remains unchanged. This concept was developed for deterministic systems. Due to support for nondeterminism in most of programming languages, researcher [7] questioned relevance of noninterference for security of program. Volpano et al [8] says that noninterference can be used for current day programming languages by using purely value based interpretation of noninterference, and with the help of Denning's certification semantics. Volpano's work regarding noninterference has set standards in language based security.

Security of a program

Motivation

In the field of data security, there are a lot of approaches to prevent leak of information for example cryptography takes care of confidentiality and integrity while data is transmitting through less secure networks, access permission on files prevent unauthorized access to files in a system where users have different privileges. But at the time of execution of program, data used in the program is vulnerable to various attacks so to maintain security at the time of execution of program and processing of data, information flow policies are used. The subject is defined as an executing authority it can be a user or parent process, object can be a file, program variable, memory location etc. In a multilevel security system a subject has permission related to objects. Information flow verification of program only considering objects may seem to be secure but with a particular subject same program may be insecure, so we considered subjects in static analysis of python program.

Goals

To develop a platform that takes input a python program and labels of each variable used in the program, and provide answers to various queries regarding the security of information flow.

Chapter 2

Background

Bell Lapadula security model [2]

This model defines four senstivity labels "Top secret", "secret", "classified", and "Public" each object must be labled from one of these labels. This model uses mandetory access control , mandetory denotes that they can not be changed. System contains subjects (user) , labeled objects, state machine with a set of states it allowed to go. Model preserves security of information in transitions of one state to another. Introduces (i) *(star) prperty: No Write-Down (NWD), it prevents subjects from higher label to write in to objects of lower label, this stops leak of information. (ii) simple security property: No Read Up (NRU), it prevents subjects from lower label to read from objects from higher label. A access matrix of subject and object is used to define permissions for subjects to use objects. This model is based on confidentiality of information.

Biba Security Model [3]

This model ensures integrity of data. To maintain integrity of data it ensures three things (a) Prevention of modification from unathorized user. (b) Prevention of unathorized modification from authorized user. Bia model defines integrity classes and rules to preserve integrity of data. The simple integrity rule says that subjects can not read objects of lower label of integrity (No read down). The *(star) integrity rules says that subjects can not write into objects of higher label(No write up). These rules are in contrast with Bell Lapadula model because Biba model considers integrity of information instead of confidentiality. Here labels denotes degree of trust. Lowest label is not reliable so it is not allowed to write to others similarly highest label is not allowed to read from others otherwise it can be corrupted by unreliable information.

Denning's Lattice model [1]

This model is based on Lattice model. There is set of security classes each denotes disjoint set of information, unlike previous models it gives facility to perform operation on two security

class labels, operations are lub denoted by $\oplus$ and glb denoted by $\otimes$. Both operation helps to calculate security label of an expression involving many objects from different security classes. To preserve security in information flow there is a basic rule: if information flowing from x to y ($x \rightarrow y$) then for secure information flow constraint $\underline{x} \leq \underline{y}$ must satisfy, $\underline{x}$ denotes security class of x. So all information flow heading upward in lattice diagram are secure.

Reader Writer Flow Model [4]

—

Chapter 3

Secure Information Flow Analysis of Python Programs

There have been many studies on information flow control and all of them share some basic properties like information flow should be from less secure entity to more secure entity. Denning's book [1] has a chapter on information flow control, this chapter describes lattice model for information flow [9], this makes it easy to track information flow in a program using transitivity property. Analysis has been done on basic operations which involve information flow like assignment operation (explicit flow) based on data flow, conditional operations like if else, while etc. (implicit flows) based on control flow, information flow through covert channel based on traps and exception in programs. Here are some basic rules given in [1], (arrow $\rightarrow$ denotes information flow).

- x = y : y $\rightarrow$ x
- if e then x = y : e $\rightarrow$ x
- while w if e then x = y w =false : w $\oplus$ e $\rightarrow$ x
- infinite loop: while w ; x = y :- w $\rightarrow$ x etc.

Chen et al. [10](published in 2014) presented work on python byte-code and claimed that there was no work related to python at that time. They implemented information flow checker for python byte-code using static and dynamic analysis but their main focus is on information flow policies related to objects. Kumar et al. [4] introduce a new model to work with subjects and my work will be focused on this. Conti et al. [11] provide library support in python for information flow analysis in explicit flows only. Data Security can be achieved using information flow policies. Execution of any action like copying of file, read operation on file or memory, write operation on file or memory, execution of statement etc may cause flow of information. Usually, information revealed that was unknown before execution of statement termed as in-

formation flow otherwise if it was known already then there will be no information flow. This thesis focuses on information flow between variables used in a python program irrespective of the amount of information flowing. There are two kinds of information flow among variables:

1. **Explicit Information Flow**

```
1  x = y
2  x = math.log(y)
```

Listing 3.1: Python example

these assignment operations are example of explicit flow because information related to variable y is flowing into x using data dependency in both cases.

2. **Implicit Information Flow**

```
1  if y == 1:
2  x = 1
3
```

Listing 3.2: Python example

```
while y < z:
x = 1
y += 1
```

Listing 3.3: Python example

in these examples value of x after execution of statements depends on the control path taken by the program, variable y and z used in choosing between two control path in while program, this involvement of y and z in making decision reduce the uncertainty of variables y and z. Whether y < z or y > z can be observed with help of final value of x after execution of given code in listing 3.3 . So there is indirect implicit flow from y to x in the first example and implicit flow from y and z to x in the second example[1].

```
x = 0
while y == 1:
pass
x = 1
```

Listing 3.4: Python example

Here assignment operation on x in Listing 3.4 is outside of while body but still information flowing from y to x because execution of x = 1 statement depends on termination of while loop, so you can know value of y by checking the value of x after execution of program, for example if value of x is 1 and program terminated then y must be other than 1, if while loop goes in infinite loop then y must be 1.

The first chapter describes python program certification with the help of Denning's Lattice Model. The lowest security class in this lattice assumed is Low everyone can read information from this class, the highest class in the lattice is High, information from all security classes can flow into it but no information can flow from this class to others. For Example certification of python code in Listing 3.5 needs to follow Denning's lattice and constraints written in figure 3.1, security class of variable var is denoted by $\underline{var}$.

```
x = 0
z = 1
y = 0
if x :
while z :
y = 1
```

Listing 3.5: Python example

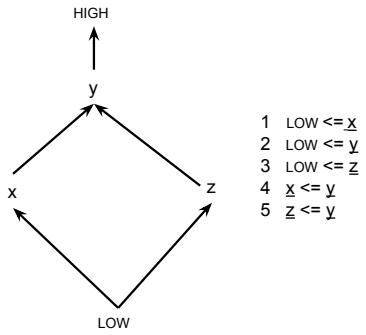

Figure 3.1: Lattice of Listing 3.5

Chapter **??** describes information flow by nonterminating while loop and how to certify program in such situation. Chapter **??** describes how information flows among thread in the multi-threaded program using semaphore. Chapter **??** presents a new approach to certifying a program using more sophisticated labels (RWFM label) and it also considers subjects for certification of the program.

Chapter 4

Approach to certify Python Programs

Category 1 constraint Generator : PC reset and fixed labels.

This chapter describes working of first algorithm for constraint generation. All four algorithms are different because of different combination of PC label management scheme and use of dynamic labels. In this algorithm we are using fixed label. Fixed label denotes that label will not change throughout the program. PC reset denotes that after completion of scope of a particular conditional/iteration body PC reset back to the PC just before the execution of conditional/iteration statement. PC monotonic denotes a scheme in which PC label never lose any information once it acquires, it just grows monotonically. This algorithm is able to capture basic information flows within a program, so this algorithm will certify a large number of programs as secure. Program certified secure by this algorithm does not mean that its fully secure, it certifies secure because of the limitation in detection of information flows in program. Some information flows which are not captured by this algorithm may violate information security.

Working

All four algorithm shares same basic structure for parsing input program. Dynamic analysis can not process all branches in a one go but static analysis process all branches in one run. Algorithms generates constraints for all possible control branches in program. PC keeps track of variables used in conditional statement and iteration statement. Assignment operation are responsible for information flows so at each occurrence of assignment operation constraints generated with the help of PC label.

Constraint Rules

1. < x := e > generate constraint $[\lambda(e) \oplus \lambda(PC) \leq \lambda(x)]$ and update PC label
   $\lambda(PC) = \lambda(e) \oplus \lambda(PC)$

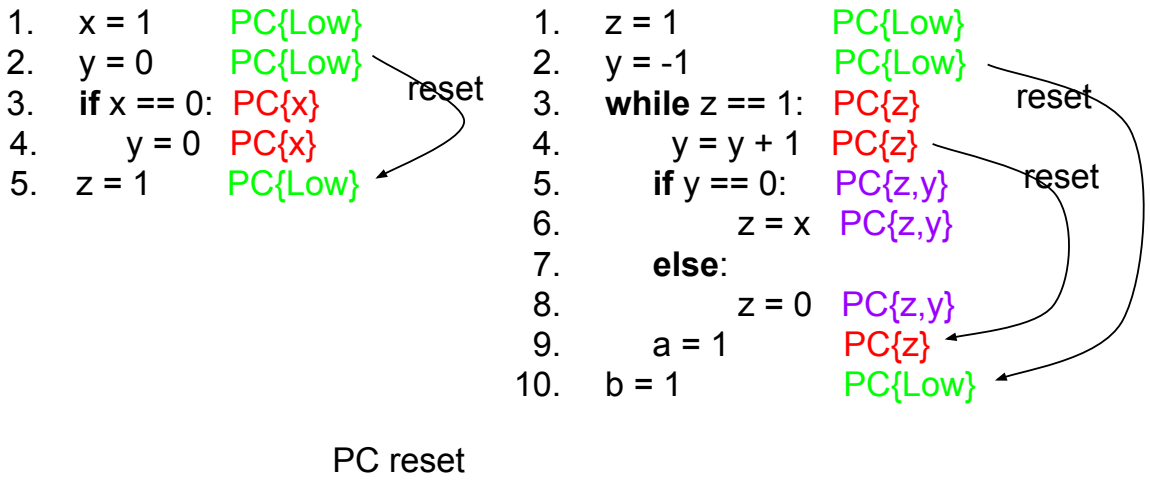

Figure 4.1: Example for PC reset

2. < x := e > generate constraint $[\lambda(e) \oplus \lambda(PC) \leq \lambda(x)]$ and update PC label $\lambda(PC) = \lambda(e) \oplus \lambda(PC)$
3. < if e then c1 else c2> $\forall x \in (modified\_global(\ c1\ \ and\ \ c2) \cup \{PC\})$ generate constraints $[\lambda(e) \oplus \lambda(PC) \leq \lambda(x)]$ and update PC label $\lambda(PC) = \lambda(e) \oplus \lambda(PC)$
4. < while e do c > $\forall x \in (modified\_global(c) \cup \{PC\})$ generate constraints $[\lambda(e) \oplus \lambda(PC) \leq \lambda(x)]$ and update PC label $\lambda(PC) = \lambda(e) \oplus \lambda(PC)$

Key Idea

Direct information flow happens because of copying or assigning values. Implicit information flow happens because of control dependency, this algorithm focuses on information flow from variables used in condition statement of if else(conditional) and while(iteration) to all variables modified in the body of iteration or conditional.

```
1 def(x,y):      #copy x to y
2 y = 0
3 z = 0
4 if x == 0:     # implicit flow x -> z PC{x}
5 z = 1
6 if z == 0:     # implicit flow z-> y PC{z}
7 y = 1
8 p = q          # direct flow q -> p PC{q}
```

Listing 4.1: Python example

Constraints generated by this algorithm for listing 4.1 are given below.

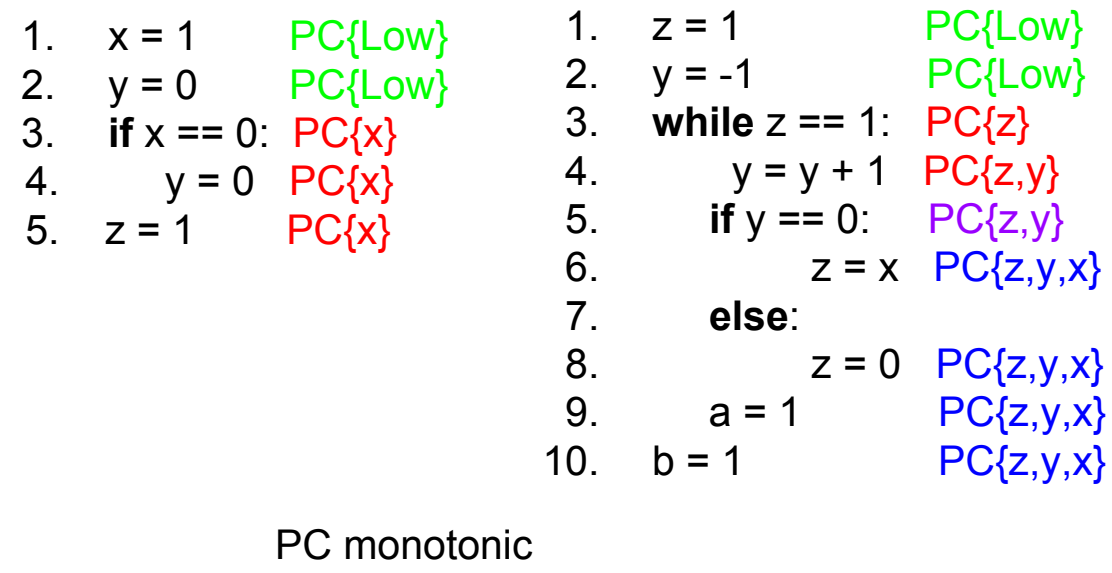

Figure 4.2: Example for PC monotonic

- x <= z
- z <= y

Limitations

For listing 4.1 this algorithm is able to capture all information flows, but in more complex programs it may declare falsely a program secure.

```
#Procedure copy5
y = 0
while x==0 :
pass
y = 1
```

Listing 4.2: Python version of copy5 example in [1]. goal: information flow from x to y

For listing 4.2 this algorithm generated only one constraint Low <= y, that shows this algorithm will certify listing 4.2 secure always irrespective of information flow x to y is secure or not. All these limitation in capturing information flow raised because of PC label management in this algorithm is only focus on local information flow. Next algorithm will try to remove these limitation using monotonic PC label management. Appendix A shows the implementation of this algorithm.

```
def fun(x, y, z):
a = x
```

```
3 y = a
4 a = z
5 fun(x, y, z)
```

Listing 4.3: Python version of dynamic label example in [1]. goal: information flow from x to y

Another limitation of this algorithm is related to use of fixed label. In listing 4.3 this algorithm detect false information flow z to y. Category 3 uses both dynamic and fixed label to remove this limitation.

Category 2 constraints : PC monotonic and fixed labels.

This chapter describes working of category 2 constraint generator. This algorithm is a improved version of previous category 1 algorithm. This algorithm using monotonic PC label instead of PC reset. By using monotonic PC this algorithm is able to detect additional information flows in program.

Working

rules

Key Idea

This analysis extension of previous algorithm, non terminating loops create a control dependency between variables used in condition of loop and the rest of the code where control can go subsequently on termination of loop, because of this behavior of non terminating loop PC storing all the dependencies.

Limitations

In static analysis if constraint resolver ignores the order of generated constraints then it may show some additional false information flow in program, this is responsible for overhead and imprecision in certification process. Use of dynamic label solves this problem easily on the cost of more complex analysis.

```
1 def fun(x, y, z):
2 a = x
3 y = a
4 a = z
5 fun(x, y, z)
```

Listing 4.4: Python version of dynamic label example in [1]. goal: information flow from x to y

constraints generated for listing 4.4 are given below:

- x <= a
- a + x <= y
- a + x + z <= a

these constraints shows that there is information flow z to y (using a <= y and z <= a) but in program there is no such flow exist. Because of such information flow constraint resolver may certify a secure program as not secure and it also create extra overhead on resolver. This example shows flaw in approach of using fixed label everywhere. Next algorithm will try to remove this limitation.

Category 3 constraints : PC reset and dynamic labels.

This chapter describes category 3 constraint generator. This algorithm introduces use of dynamic label. Global variable are using fixed label and all local variables are assigned dynamic label. Information can flow outside only because of modification of global variable, modification of local variable doe not cause information because information remains in program itself.

Constraint Rules

- < x := e > generate constraint $[\lambda(e) \oplus \lambda(PC) \leq \lambda(x)]$ and update PC label $\lambda(PC) = \lambda(e) \oplus \lambda(PC)$
- < if e then c1 else c2>
  1. $\forall x \in (modified\_global(\ c1\ \ and\ \ c2) \cup \{PC\})$ generate constraints $[\lambda(e) \oplus \lambda(PC) \leq \lambda(x)]$ and update PC label $\lambda(PC) = \lambda(e) \oplus \lambda(PC)$
  2. $\forall x \in (modified\_local(\ c1\ \ and\ \ c2) \cup \{PC\})$ update PC label $\lambda(PC) = \lambda(e) \oplus \lambda(PC)$
- < while e do c >
  1. $\forall x \in (modified\_global(c) \cup \{PC\})$ generate constraints $[\lambda(e) \oplus \lambda(PC) \leq \lambda(x)]$ and update PC label $\lambda(PC) = \lambda(e) \oplus \lambda(PC)$
  2. $\forall x \in (modified\_local(c) \cup \{PC\})$ update PC label $\lambda(PC) = \lambda(e) \oplus \lambda(PC)$

In static analysis if constraint resolver ignores the order of generated constraints then it may show some additional false information flow in program, this is responsible for overhead and imprecision in certification process. Use of dynamic label solves this problem easily on the cost of more complex analysis.

'a' is a local variable in function defined below.
def function(x,y,z):

    a = x

```
    y = a
    a = z
```

static analysis will generate constraints 1. x≤a, 2. a≤y, 3. z≤a.
Last two constraints shows false information flow from z to y (z→y).

Dynamic label analysis
1. $\lambda(a) :=$ x
2. $y \leq \lambda(a)\{x\}$
3. $\lambda(a) := z$

This analysis treats global and local variable differently so it avoids false constraints successfully without tracking order of constraints explicitly.

```
a = x
while 1:
    y = a
    a = z
```

Dynamic label Analysis:
First iteration of while:
1. $\lambda(a) :=$ x
2. $y \leq \lambda(a)\{x\}$
3. $\lambda(a) := z$

Second Iteration:
2. $y \leq \lambda(a)\{z\}$
3. $\lambda(a) := z$

Dynamic label analysis generating different constraints for first iteration and second iteration but static analysis is not able to distinguish between information flow in first iteration and second iteration of while loop.

Key Idea

Definition of information flow among objects : If any data can be guessed by using given objects which was unknown previously, by using this idea information can flow outside only because of modification of global objects, so any information flow from local objects to global objects must be checked for security breach. Local variable plays a role of temporary in flow of information from one global to another, so local variable must keep track of information they hold, dynamic label is a good technique to keep track of history of information stored in a local variable.

Limitations

This constraint generator again using PC reset label scheme. We created this category for thorough analysis and comparison among all categories. So it shares the first limitation of category 1. It fails to capture global information flows created by non terminating loops.

Category 4 constraints: PC monotonic and dynamic label.

This algorithm is best among all four algorithm in terms information security. Dynamic label processing increases time complexity of this algorithm but we used a property of PC label to make optimization. Constraint generation rules are same as category 3.

Key Idea

In this analysis PC never gets reset because we want to track all possible information flows including information flows from a nonterminating loop to rest of code.

'a' is a local var

a = x

while w:

    y = a

    a = z

z = y

Dynamic label Analysis:

1. $\lambda(a) = x$

2.PC{w}

3.PC{w, λ(a){x}} $w \oplus x \leq y$

4.$\lambda(a) = z$

5.PC{w, x, y} $w \oplus x \oplus y \leq z$

Input Program	C1	C2	C3	C4

`#'a' is a local var`
`a = x`
`while w:`
`y = a`
`a = z`
`z = y`
	`x <= a`
	`a + w <= y`
	`z + w <= a`
	`y <= z`
		`x <= a`
		`a + x + w <= y`
		`a + x + z + w <= a`
		`a + x + z + w <= y`
		`a + x + z + y + w <= z`
		`y + x + w <= z`
			`x + w <= y`
			`x + z + w <= y`
			`y <= z`
				`x`
				`x`
				`y`
				`y`

Table 4.1: Example for comparison

So this algorithm is capable to track information flow w→z in last statement z = y by using monotonic PC without generating additional false constraints.

Limitations

Use of dynamic label with monotonic creates challenge for processing of large number of labels.

Comparison among all categories of constraints.

Example given in table 4.1 is suitable to differentiate between all category. First algorithm fails to track information flow w→z in last statement z = y. Second algorithm is able to track information flow w→z in last statement z = y but it will show additional false information flow z →y too. Third algorithm avoids tracking of additional false information flow z →y but it fails to show information flow w →z because of PC reset. Fourth analysis avoids tracking of false information flow as well as tracks information flow caused by nonterminating loop(w →z). Table **??** shows constraints generated for copy program given in Denning,s book by all constraint generator. Figure 4.4 shows the relationship between set of programs declared

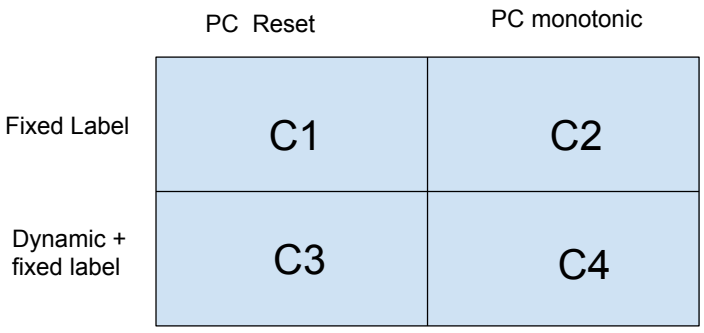

Figure 4.3: Category of constraint generator

Programs	C1	C2	C3	C4
Copy1	x <= z z <= y	x <= z x + z <= y	x <= y	x <= y
Copy2	Low <= z Low <= y y + z <= y y + x + z <= z y + z <= z	Low <= z Low <= y y + z <= y y + x + z <= z y + z <= z y + x + z <= y	Low <= y y <= y y + x <= y	Low <= y <= y y + x <=
Copy3	x + s0 <= s0 x + s1 <= s1 s0 <= s0 Low <= y s1 <= s1	x + s0 <= s0 x + s1 <= s1 s0 <= s0 s0 <= y s1 + s0 <= s1 s1 <= s1 s1 <= y s1 + s0 <= s0	x + s0 <= s0 x + s1 <= s1 s0 <= s0 Low <= y s1 <= s1	x + s0 x + s1 s0 <= s s0 <= y s1 + s0 s1 <= s s1 <= y s1 + s0
Copy4	x <= e0 x <= e1 Low <= y Low <= e1 Low <= e0	x <= e0 x <= e1 e0 <= y e0 <= e1 e1 <= y e1 <= e0	x <= e0 x <= e1 Low <= y Low <= e1 Low <= e0	x <= e0 x <= e1 e0 <= y e0 <= e e1 <= y e1 <= e
Copy5	Low <= y	Low <= y x <= y	Low <= y	Low <= x <= y
Copy6	Low <= z Low <= sum Low <= y x + sum + z <= sum y + z <= y	Low <= z Low <= sum Low <= y x + sum + z <= sum y + x + sum + z <= y y + x + sum + z <= sum	Low <= y y <= y	Low <= y + x <=
Dynamic label	x <= a a <= y z <= a	x <= a a + x <= y a + x + z <= a	x <= y	x <= y

Table 4.2: Constraints generated by all four algorithm for copy programs given Denning [1]

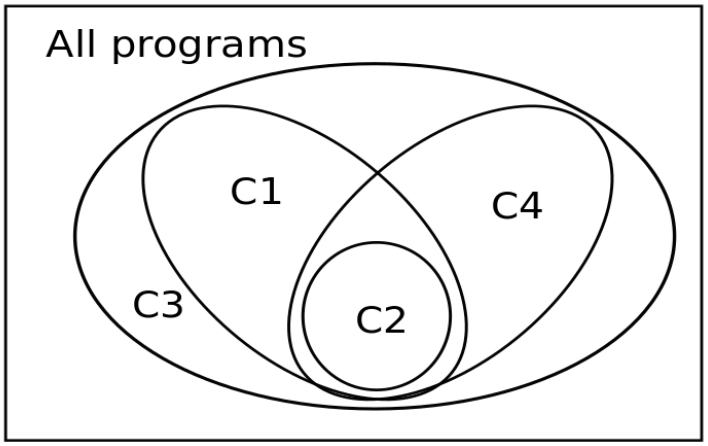

Figure 4.4: Set diagram

secure by all constraint generator. C1-C4 are abbreviation for category 1 - category 4. Number of constraints is inversely proportional to size of set of program declared secure, because more constraints means high probability of violation of security. In category 1 generator generates many false constraints because of absence of dynamic label, but category 3 uses dynamic labels with fixed so it reduces number of constraints. Constraints generated by category 1 are superset of constraints generated by category 3 this relationship shows that set of program declared secure by C1 must be subset of C3. Similarly C2 and C4 differ by use of dynamic labels so set of accepted program by C2 is a subset of set of programs accepted by C4. Use of monotonic PC label helps to capture global information flows so use of monotonic PC label increases the number of constraints. C3 and C4 differ by use of PC label scheme, C4 using monotonic PC and C3 using PC reset so constraints generated by C4 are superset of constraints generated by C3 so set of accepted programs of C4 must be subset of C3. Similarly C2 and C1 differ by PC label scheme so set of accepted programs is a subset of set of programs accepted by C1. Figure 4.5 shows the average time taken in processing one copy program by all four generator. Time taken in order C1 < C2 < C4 < C3. C4 taking little less time than C3 because of optimization in label generation, this optimization uses property of monotonic PC so it can not applied in C3.

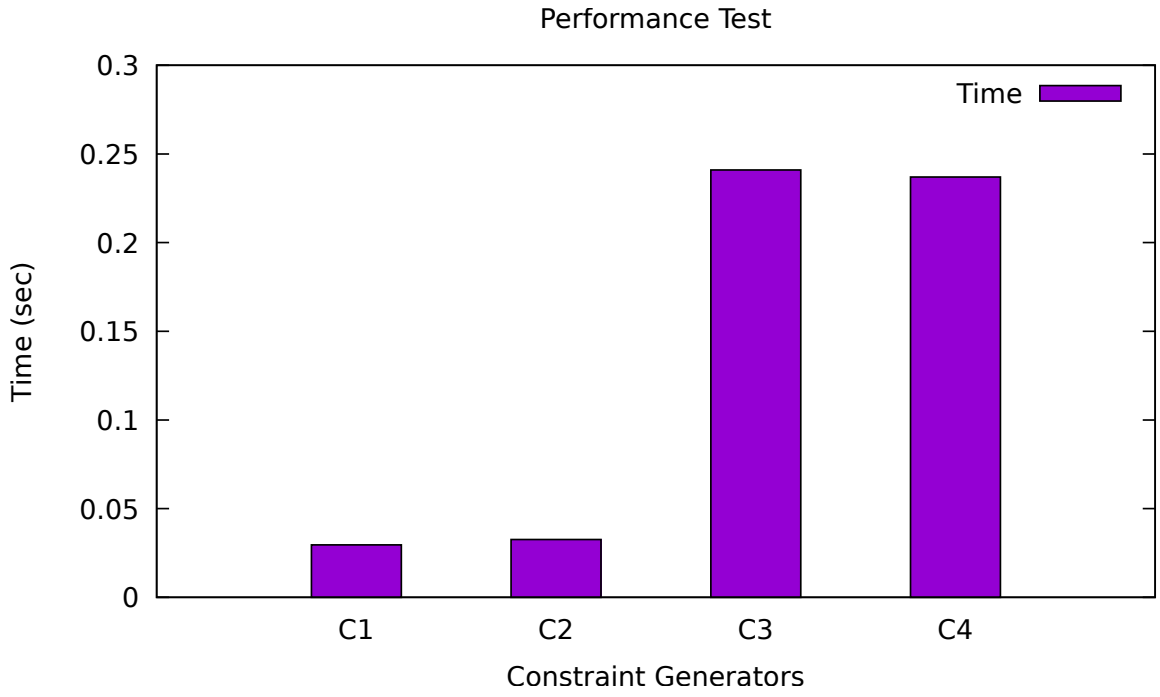

Figure 4.5: Performance test

Chapter 5

Implementation of Constraint Generator

We implemented fully automated certification platform for python source code using two python scripts, given in Appendix A and E. Block diagram in figure **??** shows modules of script1, first python source code is converted into abstract syntax tree (AST) with the help of ast library. The purpose of this step is to avoid tedious work of parsing of source code and comments. Figure 5.1 shows that parsing function reads AST word by word. If function finds any desired word it calls other handler functions to handle code related to particular word, for example: if "While" word is found, parsing function parses body of while and passes it as argument to while_handler(while_code) function. Handler function parses variables used in condition and passes the body part to parsing function again. Whenever parsing function finds assignment operation it generates constraint and goes to next word. The Block diagram for script 2 is given in figure of chapter . Effectiveness of this platform depends on the phase of constraint generation because if constraint generator fails to track information flow properly then verifier can not produce correct results. Chapter 4 describe about different constraint generators. Chapter shows the comparison among all constraint generator.

Contributions

Subject considered with objects for certification of python program. Reader Writer Flow Model [4] used to verify information flows in python program.

Implementation Details

Prerequisite Third Party libraries:

1. ast python library (for conversion of python source code into abstract syntax tree)
2. astpp python library (for readability of of abstract syntax tree of python source code)

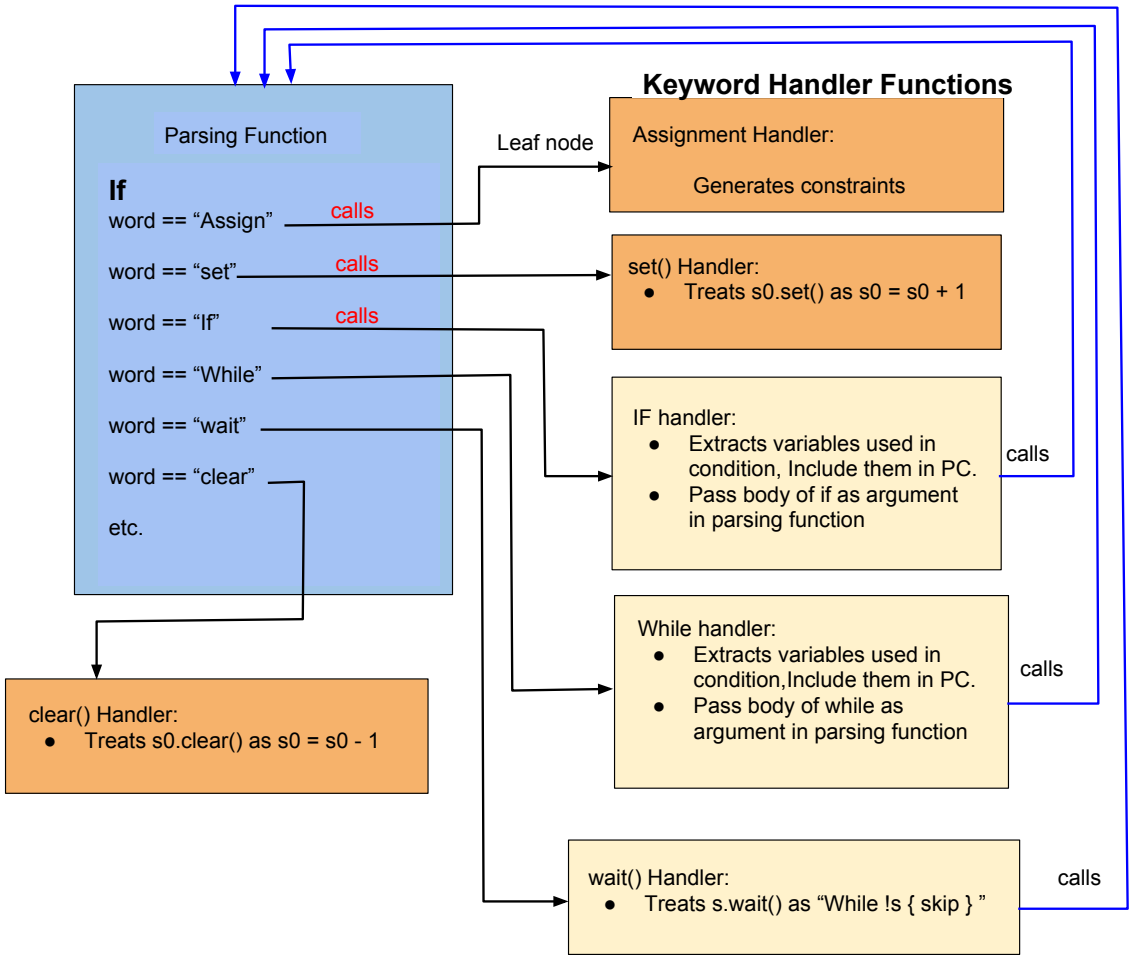

Figure 5.1: Block diagram of parsing function

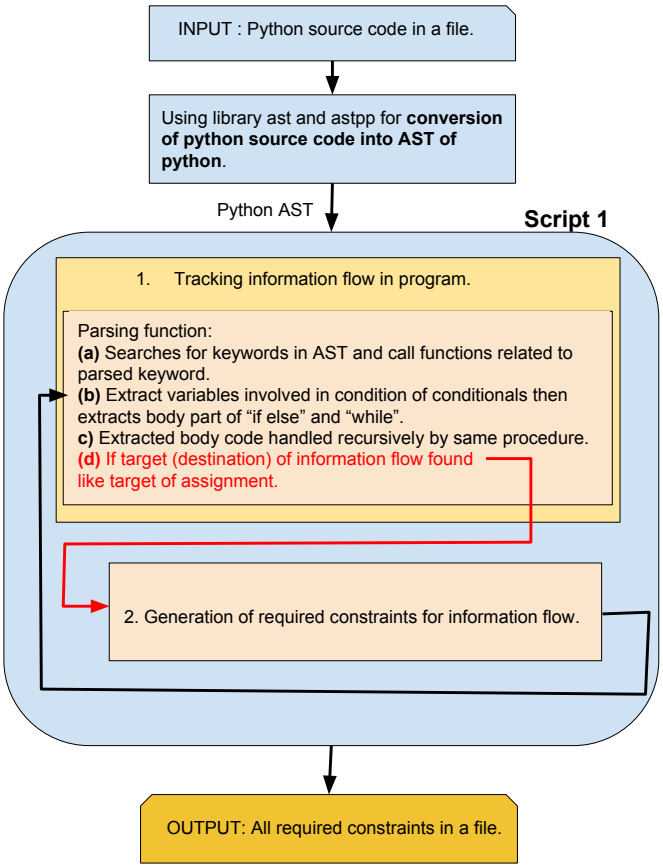

Figure 5.2: Block diagram of script1

Subset of features of python language considered for analysis.

- Assignment operations : x = e (expression)
- Conditional statements : "if else", "elif".
- Iteration : "while".
- Semaphore operations : set(), wait(), clear(), initialization of semaphore.
- Global variables and local variable in a function.
- Function calls and definitions.
- Return Statement.

Chapter 6

Case Study

Analysis of Multi-threaded Programs

In a multi-threaded program, information flows among threads because of communication and synchronization among them. There are two types of semaphores counting and binary, for synchronization among threads. For now, our script handles binary semaphores only.

Handling Information flow due to WAIT and SIGNAL operations

Traditional operations related to binary semaphore are WAIT and SIGNAL. SIGNAL operation changes the value of semaphore 0 to 1 and WAIT operation wait for an infinite time if the current value of the semaphore is 0 otherwise it changes value 1 to 0 and allows control flow forward. There are three operations related to the binary semaphore in python language wait(), set() and clear(). Traditional WAIT operation can be simulated using python wait() followed by clear() operation, SIGNAL is equivalent to set().

```
1  s = threading.Event()
2  s.wait()
3  x = 1
4
5
```

Listing 6.1: Example of wait() operation on binary semaphore.Info Flow: s→x

```
1  y = 0
2  while x:
3  pass
4  y = 1
5
```

Listing 6.2: Infinite while loop, Information Flow: x→y

Listing 6.1 and Listing 6.2 show that control flow of wait() is similar to infinite while loop so we treat wait() in a similar way. All statements which use global variables as a target of assignment and are preceded by wait() may transmit information to other threads. So information flows from semaphore s_0 to targets of assignment operations which follows s_0.wait() statement.

All semaphore operations simplified into normal operations.

- s.set() treated as **s = s + 1**.
- s.clear() treated as **s = s - 1**.
- Listing 6.1 and 6.2 shows s.wait() equivalent to **while(s == 0) { skip }**.

Benchmarking of Certification Script using Denning's Example [1]

```
#Procedure copy3
import thread
import time
import threading
s0 = threading.Event()
s1 = threading.Event()

def thread1():
global x
if x==0:
s0.set()
else:
s1.set()

def thread2():
global y
s0.wait()
s0.clear()
y=1
s1.set()

def thread3():
global y
s1.wait()
s1.clear()
y=0
s0.set()

thread.start_new_thread(thread1,())
thread.start_new_thread(thread2,())
thread.start_new_thread(thread3,())
```

Listing 6.3: Python version of copy3 example in [1]. goal: information flow from x to y

To certify the multi-threaded program in Listing 6.3 correctly our script must track information flow from x to y ($x \rightarrow y$) and must generate constraints accordingly.

Constraints generated by our script for program in Listing 6.3 are:

1. $\underline{x} \oplus \underline{s0} \leqslant \underline{s0}$
2. $\underline{x} \oplus \underline{s1} \leqslant \underline{s1}$
3. $\underline{s0} \leqslant \underline{s0}$
4. $\underline{s0} \leqslant \underline{y}$
5. $\underline{s1} \oplus \underline{s0} \leqslant \underline{s1}$
6. $\underline{s1} \leqslant \underline{s1}$
7. $\underline{s1} \leqslant \underline{y}$
8. $\underline{s1} \oplus \underline{s0} \leqslant \underline{s0}$

constraint 1 ($\underline{x} \oplus \underline{s0} \leqslant \underline{s0}$) and constraint 4 ($\underline{s0} \leqslant \underline{y}$) $\equiv$ $\underline{x} \leqslant \underline{y}$.

constraint 2 ($\underline{x} \oplus \underline{s1} \leqslant \underline{s1}$) and constraint 7 ($\underline{s1} \leqslant \underline{y}$) $\equiv$ $\underline{x} \leqslant \underline{y}$.

Hence script is able to generate correct constraints in multi-threaded program too.

Chapter 7

Conclusion & Future work

We have implemented various algorithms for constraint generation for capturing information flow in program, category 4 algorithm represents intuitive notion of correct security. Second part is verification of constraints using RWFM [4] is also implemented. We considered the subject in information flow analysis, it helps to deal with real world problems related with information flow.

Future work

- Information flow analysis on python data structures list, dictionary etc.
- Information flow analysis related to dynamic types and objects in python
- Implementation for all features of python.

Appendices

Appendix A

Python Script category 1: Constraint Generator

```
#INPUT P - the set of principals that have a stake in the computation.
#       p - computing athority
#       S - set of all principals in system.
from  more_itertools import unique_everseen
import sys
import re, pdb
import copy
import itertools

iteration = 20
debug = 0
def debugPrint(x):
    if debug == 1:
        print x

class const:
    otime = "*"  # u"\u2295"
    oplus = "+"  # u"\u2297"
    lt = "<="  # u"\u2264"

def extract_Globals(fun_str):
    global_index = [m.start() for m in re.finditer("Global\(", fun_str)]
    globals = {}
    for it in global_index:
        global_str = parse_parenthesis(it + 6, fun_str)[0]
        sq_str = parse_square_br(global_str.find("["), global_str)[0]
```

```
27          ss = sq_str.strip("[").strip("]")
28          sslist = ss.split(",")
29          for it in sslist:
30              if it == '':
31                  continue
32              globals[(it.strip("'"))] = 1
33      return globals
34
35  def SemanticsOfProgram(P,p,c,PC,S):
36      if p not in P:
37          print "MISSUSE";
38      for x in AccessedGlobal(c):
39          if p not in R(lamda(x)):
40              print "MISSUSE";
41      #intialization
42      for x in Global(c):
43          M[x] = Md[x]
44          lamda[x] = lamdad[x]
45      for x in ((VA(c) - Global(c)) | set(PC)):
46          M[x] = 0
47          lamda[x] = (p,S,set([p]))
48
49  def VA(data):
50      return set(parse_variables(data))
51
52  def Global(data): #discard all whiles, ifs and functions -> then remaining
        code will have only globals.
53      str = ""
54      length = len(data)
55      i = 0
56      while i < length - 1:
57          # checking for keyword
58          if parse_keyword(i, data) == "FunctionDef":
59              i += 11
60              i = parse_parenthesis(i, data)[1]
61          elif parse_keyword(i, data) == "Expr(":
62              i += 4
63              i = parse_parenthesis(i, data)[1]
64          elif parse_keyword(i, data) == "AugAssign":
65              i += 9
66              i = parse_parenthesis(i, data)[1]
67          elif parse_keyword(i, data) == "If":
```

```
68              i += 2
69              i = parse_if(i, data)[1]
70
71          elif parse_keyword(i, data) == "While":
72              i += 5
73              i = parse_parenthesis(i, data)[1]
74          else:
75              str += data[i]
76          i += 1
77      return set(parse_variables(str))
78
79  def parseTestVariables(data):
80      test_index = [ m.start() for m in re.finditer('test=', data) ]
81      ll = []
82      for it in test_index:
83          ll += parse_variables(parse_next_parenthesis(it,str)[0])
84      return ll
85
86  def AccessedGlobal(data):
87      #(i)right-hand side of assignment
88      #(ii) condition of branching/iteration
89      #(iii) return
90      ll = []
91      length = len(data)
92      i = 0
93      while i < length - 1:
94          # checking for keyword
95          if parse_keyword(i, data) == "AugAssign":
96              i += 9
97              tmp = parse_parenthesis(i, data)
98              ll += parse_variables(tmp[0])
99              i = tmp[1]
100         elif parse_keyword(i, data) == "Assign":
101             i += 6
102             tmp = parse_parenthesis(i, data)
103             ryt = tmp[0].split("value=")[1]
104             ll += parse_variables(ryt)
105             i = tmp[1]
106         elif parse_keyword(i, data) == "If":
107             i += 2
108             tmp = parse_parenthesis(i, data)
109             ll += parseTestVariables(tmp[0])
```

```
110             i = tmp[1]
111         elif parse_keyword(i, data) == "While":
112             i += 5
113             tmp = parse_parenthesis(i, data)
114             ll += parseTestVariables(tmp[0])
115             i = tmp[1]
116         elif parse_keyword(i,data) == "Return":
117             i += 6
118             tmp = parse_parenthesis(i,data)
119             ll += parse_variables(tmp[0])
120             i = tmp[1]
121         i += 1
122     return set(ll)
123 
124 def ModifiedGlobal(data):
125     ss = target_of_assignment(data)
126     modifiedVarList = parse_variables(ss)
127     return Global(data) & set(modifiedVarList)
128 
129 
130 def make_lub_string(llist):  # assumption list containing string elemnts
131     if len(llist) == 0:
132         return "Low"
133     if len(llist) == 1:
134         return llist[0]
135     tmp = set(llist)
136     uniq_list = list(tmp)
137     if len(uniq_list) == 1:
138         return str(uniq_list[0])
139     ret = ""
140     ret += uniq_list[0]
141     i = 1
142     while i < len(uniq_list):
143         ret += " " + const.oplus + " "
144         ret += uniq_list[i]
145         i += 1
146     return ret
147 
148 
149 def make_glb_string(llist):  # assumption list containing string elemnts
150     # type: (list) -> string
151     if len(llist) == 0:
```

```
152         return "High"
153     if len(llist) == 1:
154         return llist[0]
155     uniq_list = list(set(llist))
156     if len(uniq_list) == 1:
157         return uniq_list[0]
158     ret = ""
159     ret += uniq_list[0]
160     i = 1
161     while i < len(uniq_list):
162         ret += " " + const.otime + " "
163         ret += uniq_list[i]
164         i += 1
165     return ret
166 
167 
168 def split_through_orelse(if_str):
169     # find first body word
170     i = if_str.find("body=[")
171     i = parse_square_br(i + 5, if_str)[1] + 1
172     return ["{" + if_str[1:i] + "}", if_str[i + 7:]]
173 
174 
175 def parse_keyword(i, data):
176     # checking for
177     funLen =  len("Expr(value=Call(func=Name(id='")
178     attrLen = len("Expr(value=Call(func=Attribute(value=Name(id='")
179 
180     if i + 6 < len(data) - 1 and data[i:i + 6] == 'Assign':
181         return "Assign"
182     if i + 9 < len(data) - 1 and data[i:i + 9] == 'AugAssign':
183         return "AugAssign"
184     if i + 2 < len(data) - 1 and data[i:i + 2] == 'If':
185         return "If"
186     if i + 5 < len(data) - 1 and data[i:i + 5] == 'While':
187         return "While"
188     if i + 11 < len(data) - 1 and data[i:i + 11] == 'FunctionDef':
189         return "FunctionDef"
190     if i + 6 < len(data) - 1 and data[i:i+6] == "Return":
191         return "Return"
192     if i +  funLen < len(data) - 1 and data[i:i + funLen] == "Expr(value=
    Call(func=Name(id='":
```

```
193   return  "fun_call"
194     if i + attrLen  < len(data) - 1 and data[i:i + attrLen ] == "Expr(value
     =Call(func=Attribute(value=Name(id='":
195         if extract_variavle_name(i+attrLen,data) == 'thread':
196             return "thread_fun_call"
197         else:
198       return "set_clear_wait"
199     return "none"
200
201 def parse_square_br(i, data):
202     if data[i] != '[':
203         print "Error: [ is missing"
204         return []
205     ret = "["
206     count = 1
207     i += 1
208     while count > 0 and i < len(data) - 1:
209         if data[i] == '[':
210             count += 1
211         if data[i] == ']':
212             count -= 1
213         ret += data[i]
214         i += 1
215     return [ret, i]
216
217
218 def parse_parenthesis(i, data):
219     # type: (int , string) -> string
220     if data[i] != '(':
221         print data[i-4:i+4],data[i]
222         print "Error: ( is missing"
223         return []
224     ret = "("
225     count = 1
226     i += 1
227     while count > 0 and i < len(data) - 1:
228         if data[i] == '(':
229             count += 1
230         if data[i] == ')':
231             count -= 1
232         ret += data[i]
233         i += 1
```

```
234     return [ret, i]
235
236 def parse_next_parenthesis(i,data):
237     while i < len(data)-1 and data[i] != '(':
238         i+=1
239     if i == len(data)-1:
240         print "No parenthesis in string"
241         return ["",i]
242     ret = "("
243     count = 1
244     i += 1
245     while count > 0 and i < len(data) - 1:
246         if data[i] == '(':
247             count += 1
248         if data[i] == ')':
249             count -= 1
250         ret += data[i]
251         i += 1
252     return [ret, i]
253
254 def extract_variavle_name(startpos, line):
255     # string->string
256     var = ""
257     while line[startpos] != "'":
258         var += line[startpos]
259         startpos += 1
260     return var
261
262 def target_of_assignment(str):  # find all targets
263     # string->list
264     targets_ptrn = r"targets=\[.*?\]"
265     ctargets_ptrn = re.compile(targets_ptrn)
266     temp_list = ctargets_ptrn.findall(str)
267     ret = ''.join(temp_list)  # converting to string
268     return ret
269
270
271 def parse_variables(line):
272     # type: (string) -> list
273     # type: (str) -> object
274     id_index = [m.start() for m in re.finditer('id=', line)]
275     var_list = []
```

```
276     for it in id_index:
277         vname = extract_variavle_name(it + 4, line)
278         if vname == "False" or vname == "True":
279             continue
280         var_list.append(vname)
281     return var_list
282 
283 
284 def multiple_assign(assign_str, target_id_index, PC):
285     global line
286     tmp = assign_str.split("value", 1)
287     rvalue = parse_variables(tmp[0])
288     lvalue = parse_variables(tmp[1])
289 
290     # printing denning's rule
291     for it in rvalue:
292         # left = make_lub_string(dict[key])
293         if len(lvalue) == 0:
294             print "low " + const.lt + " " + it
295             line += 1
296         else:
297             print make_lub_string(lvalue), const.lt, it
298             line += 1
299     return PC[:]
300 
301 def pc_update(PC, list):
302     #print "pre update", PC
303     for pc in PC:
304         pc += list
305     #print "post update",PC
306 
307 def assign_denning(assign_str, PC):  # applying dennig's model on
    assignments
308     #pdb.set_trace()
309     global line
310     global output
311     ss = assign_str.split("value")
312     target_id_index = [m.start() for m in re.finditer('id=', ss[0])]
313 
314     if len(target_id_index) > 1:
315         return multiple_assign(assign_str, target_id_index, PC)
316 
```

```
317     if "id='" in ss[0]:
318         left = extract_variavle_name(0, ss[0].split("id='")[1])
319     else:
320         left = ['const']
321     id_index = [m.start() for m in
322             re.finditer('id=', ss[1])]  # list of starting index of
    variables in right part of string
323     rvalue = []
324     if len(id_index) == 0:
325         # rvalue.append("low")
326         pass
327     else:
328         for it in id_index:
329             startpos = it + 4
330             vname = extract_variavle_name(startpos, ss[1])
331             """Exclusion of False keyword"""
332             if vname == "False" or vname == "True":
333                 continue
334             rvalue.append(vname)
335     #pc_update(PC,rvalue)
336
337     ret = ""
338     l = rvalue
339     #l1 = l + lambda(pc)
340     #updating PC
341     for pc in PC:
342             output.append(make_lub_string(pc+rvalue)+ " " + const.lt + " "
    + make_glb_string([left]))
343     line += 1
344     return PC[:]
345
346
347 def augAssign_denning(called_by_fun, fun_global, augAssign_str, PC):
348     # i = augAssign_str.find("id=")
349     # Su = [extract_variavle_name(i + 4, augAssign_str)]
350     global output
351     Sr = parse_variables(augAssign_str)
352     for pc in PC:
353         output.append( make_lub_string(pc) + " " + const.lt + " " + Sr[0])
354     return PC[:]
355
356 PCA = []
```

```
357
358 def if_denning(called_by_fun, fun_global, if_str, rest, PC):
359     # type: (list, list, string, dict, string) -> print rules
360     #pdb.set_trace()
361     #print "begin IF",PC
362     if "orelse=" not in if_str:
363         # print "termination",if_str
364         if if_str[0:2] == "[]":  # absence of else part
365             return []
366         else:  # handling else part
367             else_str = if_str
368             continuous_parse([], ccalled_by_fun, fun_global, else_str, PC
    [:])
369             return []
370
371     tmp = split_through_orelse(if_str)
372     if_half = tmp[0]
373     ladder = tmp[1]
374
375     if if_str[1:5] != "test":
376         print "Error test not found in if"
377
378     """extract test=...() from if_half"""
379     i = if_half.find("(")
380     tmp = parse_parenthesis(i, if_half)
381     test_str = tmp[0]
382     #parent_list += parse_variables(test_str)
383     i = tmp[1]
384     pc_update(PC,parse_variables(test_str))
385
386     """then extract body part and process like normal AST text """
387     # body processing
388     body_onward_str = if_half[i:]  ### Asumption : Compare string always
    followed by body=[...] imediatly
389     # setting i to location of [ in body_str: ,body=[..
390     i = body_onward_str.find("[")
391     body_str = parse_square_br(i, body_onward_str)[0]
392     memo = {}
393     memo2 = {}
394     PC1 = copy._deepcopy_list(PC,memo)
395     PC2 = copy._deepcopy_list(PC, memo2)
396     PCA = list(continuous_parse([], called_by_fun, fun_global, body_str ,
```

```
    PC1[:]))
397     PCB = list(continuous_parse([], called_by_fun, fun_global, ladder , PC2
    [:]))
398     #debugPrint(PCA)
399     #debugPrint(PCB)
400     #print "end IF",PCB + PCA
401     return PCB + PCA
402 
403 def while_denning(iteration,  called_by_fun, fun_global, while_str, PC):
404     # debug print "printing while_str",while_str[6:10]
405     compare = "()"
406     if while_str[6:10] == "Name":
407         tmp = parse_parenthesis(10, while_str)
408         compare = tmp[0]
409         i = tmp[1]
410     elif while_str[6:10] == "Comp":
411         tmp = parse_parenthesis(13, while_str)
412         compare = tmp[0]
413         i = tmp[1]
414     pc_update(PC,parse_variables(compare))
415     # body processing
416     body_onward_str = while_str[i:]  ### Asumption : Compare string always
    followed by body=[...] imediatly
417     # setting i to location of [ in body_str: ,body=[..
418     i = body_onward_str.find("[")
419     body_str = parse_square_br(i, body_onward_str)[0]
420     memo = {}
421     memo2 = {}
422     PC1 = copy._deepcopy_list(PC, memo)
423     PC2 = copy._deepcopy_list(PC, memo2)
424 
425     PCB = list(continuous_parse( [], called_by_fun, fun_global, body_str,
    PC2[:]))
426     return PC1 + PCB
427 
428 def set_clear_denning( called_by_fun, fun_global, expr_str, PC):
429     #ASSUMPTION SEMAPHORE VAR IS ALWAYS GLOBAL
430     global output
431     i = expr_str.find("Call(")
432     i += 4
433     call_str = parse_parenthesis(i, expr_str)[0]
434     i = call_str.find("Attribute(")
```

```
435     i += 9
436     attribute_str = parse_parenthesis(i, call_str)[0]
437     i = attribute_str.find("Name(")
438     i += 4
439     # name_str = parse_parenthesis(i,attribute_str)[0]
440     i = attribute_str.find("id=")
441     var_name = extract_variavle_name(i + 4, attribute_str)
442     i = attribute_str.find("attr=")
443     attr = extract_variavle_name(i + 6, attribute_str)
444     if attr == "set" or attr == "clear":
445         # treat it like AugAssign s0 += 1
446         # treat it like AugAssign s0 -= 1
447         #print "set clear-> ",PC
448         for pc in PC:
449             output.append( make_lub_string(pc+[var_name])+ " " + const.lt +
    " " + make_glb_string([var_name])) #label[left]
450     #elif attr == "wait":
451         #global_while_list.append(var_name)
452         #print "wait -> ",PC
453
454
455
456 def fun_denning(fun_str,PC):
457     fun_name = extract_variavle_name(fun_str.find("name=") + 6, fun_str)
458     fun_globals = extract_Globals(fun_str)
459     funPC = []
460     PC = continuous_parse(funPC, fun_name, fun_globals, fun_str,PC)
461     return PC + funPC
462
463 # global var for counting
464 ww = ww1 = ww2 = 1
465
466 # global while list
467 global_while_list = []
468
469
470 def parse_if(i, data):
471     tmp = parse_parenthesis(i, data)
472     i = tmp[1]
473     if_str = tmp[0]
474     rest = data[i:]
475     return [if_str, i, rest]
```

```
476
477 def uniq(l):
478     ll = []
479     for it in l:
480         ll.append(list(set(it)))
481     return ll
482
483 fun_hash = {}
484
485 def duplicateRemoval(PC):
486     tmpPC = []
487     for pc in PC:
488         tmppc = list(set(pc))
489         tmppc.sort()
490         tmpPC.append(tmppc)
491     tmpPC.sort()
492     return list(tmpPC for tmpPC,_ in itertools.groupby(tmpPC))
493
494 recCount =0
495 def continuous_parse( funPC, called_by_fun, fun_global, data, PC):
496     #pdb.set_trace()
497     # type: (object, object) -> object
498     global recCount
499     length = len(data)
500     i = 0
501     recCount += 1
502     debugPrint(recCount)
503     debugPrint("Continuous_parse before loop:")
504     while i < length - 1:
505         #checking for keyword
506         if parse_keyword(i,data) == "FunctionDef":  #skipping all function
    definition
507             tmp = parse_parenthesis(i+11, data)
508             i = tmp[1]
509         if parse_keyword(i,data) == "thread_fun_call": #parsing function
    call used in threads
510             tmp = parse_parenthesis(i+4,data)
511             i = tmp[1]
512             if 'start_new_thread' in tmp[0]:
513                 #print "got thread call"
514                 fi = tmp[0].find("args=[")
515                 funName = extract_variavle_name(fi+15,tmp[0]) #len(args=[
```

```
    Name(id=') = 15
516                 if funName in fun_hash:
517                     findex = fun_hash[funName]
518                 else:
519                     print "Function not found but prgram called function"
520                 findex += 11
521                 tmp = parse_parenthesis(findex, data)
522                 fun_str = tmp[0]
523                 tmp = fun_denning(fun_str,[[]])
524                 memo = {}
525                 ls = duplicateRemoval(tmp)
526                 PC = copy._deepcopy_list(ls, memo)
527
528         if parse_keyword(i,data) == "fun_call":  #parsing functioncalls
529             tmp = parse_parenthesis(i+4,data)
530             #print "got fun call"
531             funName = extract_variavle_name(i + len("Expr(value=Call(func=
    Name(id='"),data)
532             i = tmp[1]
533             if funName in fun_hash:
534                 findex = fun_hash[funName]
535             else:
536                 print "Function not found but prgram called function"
537             findex += 11
538             tmp = parse_parenthesis(findex, data)
539             fun_str = tmp[0]
540             tmp = fun_denning(fun_str,PC)
541             memo = {}
542             ls = duplicateRemoval(tmp)
543             PC = copy._deepcopy_list(ls, memo)
544
545         if parse_keyword(i,data) == "Return":
546             debugPrint("Return stmt:")
547             funPC += PC
548         if parse_keyword(i, data) == "set_clear_wait":
549             i += 4
550             tmp = parse_parenthesis(i, data)
551             expr_str = tmp[0]
552             if expr_str.find("'set'") != -1 or expr_str.find("'clear'") !=
    -1 or expr_str.find("'wait'") != -1 :
553                 i = tmp[1]
554                 set_clear_denning( called_by_fun, fun_global, expr_str, PC)
```

```
555         if parse_keyword(i, data) == "AugAssign":
556             i += 9
557             tmp = parse_parenthesis(i, data)
558             augAssign_str = tmp[0]
559             i = tmp[1]
560             tmp = augAssign_denning(parent_list[:], global_while_list,
    called_by_fun, fun_global, augAssign_str, PC[:])
561             memo = {}
562             ls = duplicateRemoval(tmp)
563             PC = copy._deepcopy_list(ls, memo)
564 
565         if parse_keyword(i, data) == "Assign":
566             global ww
567             memo = {}
568             ls = duplicateRemoval(PC)
569             PC_reset = copy._deepcopy_list(ls,memo)
570             ww += 1
571             i += 6
572             tmp = parse_parenthesis(i, data)
573             assign_str = tmp[0]
574             i = tmp[1]
575             if "value=Name(id='threading'" in assign_str:
576                 continue
577             assign_denning(assign_str, PC[:])
578             memo = {}
579             PC = copy._deepcopy_list(PC_reset,memo)
580         elif parse_keyword(i, data) == "If":
581             debugPrint("If stmt:")
582             memo = {}
583             ls = duplicateRemoval(PC)
584             PC_reset = copy._deepcopy_list(ls,memo)
585             global ww1
586             ww1 += 1
587             i += 2
588             tmp = parse_if(i, data)
589             if_str = tmp[0]
590             i = tmp[1]
591             rest = tmp[2]
592             if_denning(called_by_fun, fun_global, if_str, rest, PC[:])
593             memo = {}
594             PC = copy._deepcopy_list(PC_reset,memo)
595         elif parse_keyword(i, data) == "While":
```

```
596             debugPrint("while stmt:")
597             global ww2
598             ww2 += 1
599             i += 5
600             tmp = parse_parenthesis(i, data)
601             while_str = tmp[0]
602             i = tmp[1]
603             it = 1
604             memo = {}
605             ls = duplicateRemoval(PC)
606             PC_reset = copy._deepcopy_list(ls,memo)
607             while( it <= iteration ):
608                 memo = {}
609                 lastPC = copy._deepcopy_list(PC, memo)
610                 tmp = duplicateRemoval(PC)
611                 memo = {}
612                 PC = copy._deepcopy_list(tmp, memo)
613                 #print "### While Iteration:",it
614                 tmp = while_denning(it,  called_by_fun, fun_global,
    while_str, PC[:])
615                 ls = duplicateRemoval(tmp)
616                 memo = {}
617                 PC = copy._deepcopy_list(ls, memo)
618                 #print lastPC,"|--|", PC
619                 #print "-> PC:",PC
620                 if lastPC == PC:
621                     #print "Saturation point of loop!"
622                     break
623                 it += 1
624             memo = {}
625             PC = copy._deepcopy_list(PC_reset,memo)
626         i += 1
627     debugPrint(recCount)
628     recCount -= 1;
629     debugPrint("END continuous parse:")
630     return PC[:]
631
632
633 ################################## main
    ############################################
634 with open(sys.argv[1], "r") as inputfile:
635     # data = inputfile.read().replace('\n', '').replace(' ','')
```

```
636     data = "".join(inputfile.read().split())
637 llist = []
638 dummy = []
639 su_sr_list = []
640 line = 0
641 PC = [[]]
642 gg = []
643 output = []
644 
645 fun_call_index = [m.start() for m in re.finditer("FunctionDef", data)]
646 for index in fun_call_index:
647     varName = extract_variavle_name(index+len("FunctionDef(name='"),data)
648     fun_hash[varName] = index
649 
650 #G = Global(data)
651 continuous_parse("", [], dummy, data, PC)
652 
653 for it in list(unique_everseen(output)):
654     print it
```

Appendix B

Python Script category 2: Constraint Generator

```
#INPUT P - the set of principals that have a stake in the computation.
#       p - computing athority
#       S - set of all principals in system.
from  more_itertools import unique_everseen
import sys
import re, pdb
import copy
import itertools

iteration = 20
debug = 0
def debugPrint(x):
    if debug == 1:
        print x

class const:
    otime = "*"  # u"\u2295"
    oplus = "+"  # u"\u2297"
    lt = "<="  # u"\u2264"


def SemanticsOfProgram(P,p,c,PC,S):
    if p not in P:
        print "MISSUSE";
    for x in AccessedGlobal(c):
        if p not in R(lamda(x)):
```

```
27              print "MISSUSE";
28      #intialization
29      for x in Global(c):
30          M[x] = Md[x]
31          lamda[x] = lamdad[x]
32      for x in ((VA(c) - Global(c)) | set(PC)):
33          M[x] = 0
34          lamda[x] = (p,S,set([p]))
35
36  def VA(data):
37      return set(parse_variables(data))
38
39  def Global(data): #discard all whiles, ifs and functions -> then remaining
        code will have only globals.
40      str = ""
41      length = len(data)
42      i = 0
43      while i < length - 1:
44          # checking for keyword
45          if parse_keyword(i, data) == "FunctionDef":
46              i += 11
47              i = parse_parenthesis(i, data)[1]
48          elif parse_keyword(i, data) == "Expr(":
49              i += 4
50              i = parse_parenthesis(i, data)[1]
51          elif parse_keyword(i, data) == "AugAssign":
52              i += 9
53              i = parse_parenthesis(i, data)[1]
54          elif parse_keyword(i, data) == "If":
55              i += 2
56              i = parse_if(i, data)[1]
57
58          elif parse_keyword(i, data) == "While":
59              i += 5
60              i = parse_parenthesis(i, data)[1]
61          else:
62              str += data[i]
63          i += 1
64      return set(parse_variables(str))
65
66  def parseTestVariables(data):
67      test_index = [ m.start() for m in re.finditer('test=', data) ]
```

```
    ll = []
    for it in test_index:
        ll += parse_variables(parse_next_parenthesis(it,str)[0])
    return ll

def AccessedGlobal(data):
    #(i)right-hand side of assignment
    #(ii) condition of branching/iteration
    #(iii) return
    ll = []
    length = len(data)
    i = 0
    while i < length - 1:
        # checking for keyword
        if parse_keyword(i, data) == "AugAssign":
            i += 9
            tmp = parse_parenthesis(i, data)
            ll += parse_variables(tmp[0])
            i = tmp[1]
        elif parse_keyword(i, data) == "Assign":
            i += 6
            tmp = parse_parenthesis(i, data)
            ryt = tmp[0].split("value=")[1]
            ll += parse_variables(ryt)
            i = tmp[1]
        elif parse_keyword(i, data) == "If":
            i += 2
            tmp = parse_parenthesis(i, data)
            ll += parseTestVariables(tmp[0])
            i = tmp[1]
        elif parse_keyword(i, data) == "While":
            i += 5
            tmp = parse_parenthesis(i, data)
            ll += parseTestVariables(tmp[0])
            i = tmp[1]
        elif parse_keyword(i,data) == "Return":
            i += 6
            tmp = parse_parenthesis(i,data)
            ll += parse_variables(tmp[0])
            i = tmp[1]
        i += 1
    return set(ll)
```

```
110
111 def ModifiedGlobal(data):
112     ss = target_of_assignment(data)
113     modifiedVarList = parse_variables(ss)
114     return Global(data) & set(modifiedVarList)
115
116
117 def make_lub_string(llist):  # assumption list containing string elemnts
118     if len(llist) == 0:
119         return "Low"
120     if len(llist) == 1:
121         return llist[0]
122     tmp = set(llist)
123     uniq_list = list(tmp)
124     if len(uniq_list) == 1:
125         return str(uniq_list[0])
126     ret = ""
127     ret += uniq_list[0]
128     i = 1
129     while i < len(uniq_list):
130         ret += " " + const.oplus + " "
131         ret += uniq_list[i]
132         i += 1
133     return ret
134
135
136 def make_glb_string(llist):  # assumption list containing string elemnts
137     # type: (list) -> string
138     if len(llist) == 0:
139         return "High"
140     if len(llist) == 1:
141         return llist[0]
142     uniq_list = list(set(llist))
143     if len(uniq_list) == 1:
144         return uniq_list[0]
145     ret = ""
146     ret += uniq_list[0]
147     i = 1
148     while i < len(uniq_list):
149         ret += " " + const.otime + " "
150         ret += uniq_list[i]
151         i += 1
```

```
152     return ret
153
154
155 def split_through_orelse(if_str):
156     # find first body word
157     i = if_str.find("body=[")
158     i = parse_square_br(i + 5, if_str)[1] + 1
159     return ["{" + if_str[1:i] + "}", if_str[i + 7:]]
160
161
162 def parse_keyword(i, data):
163     # checking for
164     funLen =  len("Expr(value=Call(func=Name(id='")
165     attrLen = len("Expr(value=Call(func=Attribute(value=Name(id='")
166
167     if i + 6 < len(data) - 1 and data[i:i + 6] == 'Assign':
168         return "Assign"
169     if i + 9 < len(data) - 1 and data[i:i + 9] == 'AugAssign':
170         return "AugAssign"
171     if i + 2 < len(data) - 1 and data[i:i + 2] == 'If':
172         return "If"
173     if i + 5 < len(data) - 1 and data[i:i + 5] == 'While':
174         return "While"
175     if i + 11 < len(data) - 1 and data[i:i + 11] == 'FunctionDef':
176         return "FunctionDef"
177     if i + 6 < len(data) - 1 and data[i:i+6] == "Return":
178         return "Return"
179     if i +  funLen < len(data) - 1 and data[i:i + funLen] == "Expr(value=
    Call(func=Name(id='":
180   return  "fun_call"
181     if i + attrLen  < len(data) - 1 and data[i:i + attrLen ] == "Expr(value
    =Call(func=Attribute(value=Name(id='":
182         if extract_variavle_name(i+attrLen,data) == 'thread':
183             return "thread_fun_call"
184         else:
185       return "set_clear_wait"
186     return "none"
187
188 def parse_square_br(i, data):
189     if data[i] != '[':
190         print "Error: [ is missing"
191         return []
```

```
192     ret = "["
193     count = 1
194     i += 1
195     while count > 0 and i < len(data) - 1:
196         if data[i] == '[':
197             count += 1
198         if data[i] == ']':
199             count -= 1
200         ret += data[i]
201         i += 1
202     return [ret, i]
203 
204 
205 def parse_parenthesis(i, data):
206     # type: (int , string) -> string
207     if data[i] != '(':
208         print data[i-4:i+4],data[i]
209         print "Error: ( is missing"
210         return []
211     ret = "("
212     count = 1
213     i += 1
214     while count > 0 and i < len(data) - 1:
215         if data[i] == '(':
216             count += 1
217         if data[i] == ')':
218             count -= 1
219         ret += data[i]
220         i += 1
221     return [ret, i]
222 
223 def parse_next_parenthesis(i,data):
224     while i < len(data)-1 and data[i] != '(':
225         i+=1
226     if i == len(data)-1:
227         print "No parenthesis in string"
228         return ["",i]
229     ret = "("
230     count = 1
231     i += 1
232     while count > 0 and i < len(data) - 1:
233         if data[i] == '(':
```

```
234             count += 1
235         if data[i] == ')':
236             count -= 1
237         ret += data[i]
238         i += 1
239     return [ret, i]
240 
241 def extract_variavle_name(startpos, line):
242     # string->string
243     var = ""
244     while line[startpos] != "'":
245         var += line[startpos]
246         startpos += 1
247     return var
248 
249 def target_of_assignment(str):  # find all targets
250     # string->list
251     targets_ptrn = r"targets=\[.*?\]"
252     ctargets_ptrn = re.compile(targets_ptrn)
253     temp_list = ctargets_ptrn.findall(str)
254     ret = ''.join(temp_list)  # converting to string
255     return ret
256 
257 
258 def parse_variables(line):
259     # type: (string) -> list
260     # type: (str) -> object
261     id_index = [m.start() for m in re.finditer('id=', line)]
262     var_list = []
263     for it in id_index:
264         vname = extract_variavle_name(it + 4, line)
265         if vname == "False" or vname == "True":
266             continue
267         var_list.append(vname)
268     return var_list
269 
270 
271 def multiple_assign(assign_str, target_id_index, PC):
272     global line
273     tmp = assign_str.split("value", 1)
274     rvalue = parse_variables(tmp[0])
275     lvalue = parse_variables(tmp[1])
```

```
276     pc_update(PC,lvalue)
277
278     # printing denning's rule
279     for it in rvalue:
280         # left = make_lub_string(dict[key])
281         if len(lvalue) == 0:
282             print "low " + const.lt + " " + it
283             line += 1
284         else:
285             print make_lub_string(lvalue), const.lt, it
286             line += 1
287     return PC[:]
288
289 def pc_update(PC, list):
290     #print "pre update", PC
291     for pc in PC:
292         pc += list
293     #print "post update",PC
294
295 def assign_denning(assign_str, PC):  # applying dennig's model on
    assignments
296     #pdb.set_trace()
297     global line
298     global output
299     ss = assign_str.split("value")
300     target_id_index = [m.start() for m in re.finditer('id=', ss[0])]
301
302     if len(target_id_index) > 1:
303         return multiple_assign(assign_str, target_id_index, PC)
304
305     if "id='" in ss[0]:
306         left = extract_variavle_name(0, ss[0].split("id='")[1])
307     else:
308         left = ['const']
309     id_index = [m.start() for m in
310             re.finditer('id=', ss[1])]  # list of starting index of
    variables in right part of string
311     rvalue = []
312     if len(id_index) == 0:
313         # rvalue.append("low")
314         pass
315     else:
```

```
316         for it in id_index:
317             startpos = it + 4
318             vname = extract_variavle_name(startpos, ss[1])
319             """Exclusion of False keyword"""
320             if vname == "False" or vname == "True":
321                 continue
322             rvalue.append(vname)
323     #pc_update(PC,rvalue)
324 
325     ret = ""
326     l = rvalue
327     #l1 = l + lambda(pc)
328     #updating PC
329     for pc in PC:
330         pc += l
331     for pc in PC:
332             output.append(make_lub_string(pc)+ " " + const.lt + " " +
    make_glb_string([left]))
333     line += 1
334     return PC[:]
335 
336 
337 def augAssign_denning(called_by_fun, fun_global, augAssign_str, PC):
338     # i = augAssign_str.find("id=")
339     # Su = [extract_variavle_name(i + 4, augAssign_str)]
340     global output
341     Sr = parse_variables(augAssign_str)
342     pc_update(PC,Sr)
343     for pc in PC:
344         output.append( make_lub_string(pc) + " " + const.lt + " " + Sr[0])
345     return PC[:]
346 
347 PCA = []
348 
349 def if_denning(called_by_fun, fun_global, if_str, rest, PC):
350     # type: (list, list, string, dict, string) -> print rules
351     #pdb.set_trace()
352     #print "begin IF",PC
353     if "orelse=" not in if_str:
354         # print "termination",if_str
355         if if_str[0:2] == "[]":  # absence of else part
356             return []
```

```
357         else:  # handling else part
358             else_str = if_str
359             continuous_parse([], ccalled_by_fun, fun_global, else_str, PC
    [:])
360             return []
361 
362     tmp = split_through_orelse(if_str)
363     if_half = tmp[0]
364     ladder = tmp[1]
365 
366     if if_str[1:5] != "test":
367         print "Error test not found in if"
368 
369     """extract test=...() from if_half"""
370     i = if_half.find("(")
371     tmp = parse_parenthesis(i, if_half)
372     test_str = tmp[0]
373     #parent_list += parse_variables(test_str)
374     i = tmp[1]
375     pc_update(PC,parse_variables(test_str))
376 
377     """then extract body part and process like normal AST text """
378     # body processing
379     body_onward_str = if_half[i:]  ### Asumption : Compare string always
    followed by body=[...] imediatly
380     # setting i to location of [ in body_str: ,body=[..
381     i = body_onward_str.find("[")
382     body_str = parse_square_br(i, body_onward_str)[0]
383     memo = {}
384     memo2 = {}
385     PC1 = copy._deepcopy_list(PC,memo)
386     PC2 = copy._deepcopy_list(PC, memo2)
387     PCA = list(continuous_parse([], called_by_fun, fun_global, body_str ,
    PC1[:]))
388     PCB = list(continuous_parse([], called_by_fun, fun_global, ladder , PC2
    [:]))
389     #debugPrint(PCA)
390     #debugPrint(PCB)
391     #print "end IF",PCB + PCA
392     return PCB + PCA
393 
394 def while_denning(iteration,  called_by_fun, fun_global, while_str, PC):
```

```
395     # debug print "printing while_str",while_str[6:10]
396     compare = "()"
397     if while_str[6:10] == "Name":
398         tmp = parse_parenthesis(10, while_str)
399         compare = tmp[0]
400         i = tmp[1]
401     elif while_str[6:10] == "Comp":
402         tmp = parse_parenthesis(13, while_str)
403         compare = tmp[0]
404         i = tmp[1]
405     pc_update(PC,parse_variables(compare))
406     # body processing
407     body_onward_str = while_str[i:]  ### Asumption : Compare string always
    followed by body=[...] imediatly
408     # setting i to location of [ in body_str: ,body=[..
409     i = body_onward_str.find("[")
410     body_str = parse_square_br(i, body_onward_str)[0]
411     memo = {}
412     memo2 = {}
413     PC1 = copy._deepcopy_list(PC, memo)
414     PC2 = copy._deepcopy_list(PC, memo2)
415
416     PCB = list(continuous_parse( [], called_by_fun, fun_global, body_str,
    PC2[:]))
417     return PC1 + PCB
418
419 def set_clear_denning( called_by_fun, fun_global, expr_str, PC):
420     #ASSUMPTION SEMAPHORE VAR IS ALWAYS GLOBAL
421     global output
422     i = expr_str.find("Call(")
423     i += 4
424     call_str = parse_parenthesis(i, expr_str)[0]
425     i = call_str.find("Attribute(")
426     i += 9
427     attribute_str = parse_parenthesis(i, call_str)[0]
428     i = attribute_str.find("Name(")
429     i += 4
430     # name_str = parse_parenthesis(i,attribute_str)[0]
431     i = attribute_str.find("id=")
432     var_name = extract_variavle_name(i + 4, attribute_str)
433     i = attribute_str.find("attr=")
434     attr = extract_variavle_name(i + 6, attribute_str)
```

```
435     if attr == "set" or attr == "clear":
436         # treat it like AugAssign s0 += 1
437         # treat it like AugAssign s0 -= 1
438         pc_update(PC,[var_name])
439         #print "set clear-> ",PC
440         for pc in PC:
441             output.append( make_lub_string(pc)+ " " + const.lt + " " +
    make_glb_string([var_name])) #label[left]
442     elif attr == "wait":
443         #global_while_list.append(var_name)
444         pc_update(PC,[var_name])
445         #print "wait -> ",PC
446
447 def extract_Globals(fun_str):
448     global_index = [m.start() for m in re.finditer("Global\(", fun_str)]
449     globals = {}
450     for it in global_index:
451         global_str = parse_parenthesis(it + 6, fun_str)[0]
452         sq_str = parse_square_br(global_str.find("["), global_str)[0]
453         ss = sq_str.strip("[").strip("]")
454         sslist = ss.split(",")
455         for it in sslist:
456             if it == '':
457                 continue
458             globals[(it.strip("'"))] = 1
459     return globals
460
461
462 def fun_denning(fun_str,PC):
463     fun_name = extract_variavle_name(fun_str.find("name=") + 6, fun_str)
464     fun_globals = extract_Globals(fun_str)
465     funPC = []
466     PC = continuous_parse(funPC, fun_name, fun_globals, fun_str,PC)
467     return PC + funPC
468
469 # global var for counting
470 ww = ww1 = ww2 = 1
471
472 # global while list
473 global_while_list = []
474
475
```

```
476 def parse_if(i, data):
477     tmp = parse_parenthesis(i, data)
478     i = tmp[1]
479     if_str = tmp[0]
480     rest = data[i:]
481     return [if_str, i, rest]
482 
483 def uniq(l):
484     ll = []
485     for it in l:
486         ll.append(list(set(it)))
487     return ll
488 
489 fun_hash = {}
490 
491 def duplicateRemoval(PC):
492     tmpPC = []
493     for pc in PC:
494         tmppc = list(set(pc))
495         tmppc.sort()
496         tmpPC.append(tmppc)
497     tmpPC.sort()
498     return list(tmpPC for tmpPC,_ in itertools.groupby(tmpPC))
499 
500 recCount =0
501 def continuous_parse( funPC, called_by_fun, fun_global, data, PC):
502     #pdb.set_trace()
503     # type: (object, object) -> object
504     global recCount
505     length = len(data)
506     i = 0
507     recCount += 1
508     debugPrint(recCount)
509     debugPrint("Continuous_parse before loop:")
510     while i < length - 1:
511         #checking for keyword
512         if parse_keyword(i,data) == "FunctionDef":  #skipping all function
    definition
513             tmp = parse_parenthesis(i+11, data)
514             i = tmp[1]
515         if parse_keyword(i,data) == "thread_fun_call": #parsing function
    call used in threads
```

```
516             tmp = parse_parenthesis(i+4,data)
517             i = tmp[1]
518             if 'start_new_thread' in tmp[0]:
519                 #print "got thread call"
520                 fi = tmp[0].find("args=[")
521                 funName = extract_variavle_name(fi+15,tmp[0]) #len(args=[
    Name(id=') = 15
522                 if funName in fun_hash:
523                     findex = fun_hash[funName]
524                 else:
525                     print "Function not found but prgram called function"
526                 findex += 11
527                 tmp = parse_parenthesis(findex, data)
528                 fun_str = tmp[0]
529                 tmp = fun_denning(fun_str,[[]])
530                 memo = {}
531                 ls = duplicateRemoval(tmp)
532                 PC = copy._deepcopy_list(ls, memo)
533 
534         if parse_keyword(i,data) == "fun_call":  #parsing functioncalls
535             tmp = parse_parenthesis(i+4,data)
536             #print "got fun call"
537             funName = extract_variavle_name(i + len("Expr(value=Call(func=
    Name(id='"),data)
538             i = tmp[1]
539             if funName in fun_hash:
540                 findex = fun_hash[funName]
541             else:
542                 print "Function not found but prgram called function"
543             findex += 11
544             tmp = parse_parenthesis(findex, data)
545             fun_str = tmp[0]
546             tmp = fun_denning(fun_str,PC)
547             memo = {}
548             ls = duplicateRemoval(tmp)
549             PC = copy._deepcopy_list(ls, memo)
550 
551         if parse_keyword(i,data) == "Return":
552             debugPrint("Return stmt:")
553             funPC += PC
554         if parse_keyword(i, data) == "set_clear_wait":
555             i += 4
```

```
556             tmp = parse_parenthesis(i, data)
557             expr_str = tmp[0]
558             if expr_str.find("'set'") != -1 or expr_str.find("'clear'") !=
    -1 or expr_str.find("'wait'") != -1 :
559                 i = tmp[1]
560                 set_clear_denning( called_by_fun, fun_global, expr_str, PC)
561         if parse_keyword(i, data) == "AugAssign":
562             i += 9
563             tmp = parse_parenthesis(i, data)
564             augAssign_str = tmp[0]
565             i = tmp[1]
566             tmp = augAssign_denning(parent_list[:], global_while_list,
    called_by_fun, fun_global, augAssign_str, PC[:])
567             memo = {}
568             ls = duplicateRemoval(tmp)
569             PC = copy._deepcopy_list(ls, memo)
570 
571         if parse_keyword(i, data) == "Assign":
572             global ww
573             ww += 1
574             i += 6
575             tmp = parse_parenthesis(i, data)
576             assign_str = tmp[0]
577             i = tmp[1]
578             if "value=Name(id='threading'" in assign_str:
579                 continue
580             tmp = assign_denning(assign_str, PC[:])
581             memo = {}
582             ls = duplicateRemoval(tmp)
583             PC = copy._deepcopy_list(ls,memo)
584         elif parse_keyword(i, data) == "If":
585             debugPrint("If stmt:")
586             global ww1
587             ww1 += 1
588             i += 2
589             tmp = parse_if(i, data)
590             if_str = tmp[0]
591             i = tmp[1]
592             rest = tmp[2]
593             tmp = if_denning(called_by_fun, fun_global, if_str, rest, PC
    [:])
594             ls = duplicateRemoval(tmp)
```

```
595             memo = {}
596             PC = copy._deepcopy_list(ls, memo)
597         elif parse_keyword(i, data) == "While":
598             debugPrint("while stmt:")
599             global ww2
600             ww2 += 1
601             i += 5
602             tmp = parse_parenthesis(i, data)
603             while_str = tmp[0]
604             i = tmp[1]
605             it = 1
606             while( it <= iteration ):
607                 memo = {}
608                 lastPC = copy._deepcopy_list(PC, memo)
609                 tmp = duplicateRemoval(PC)
610                 memo = {}
611                 PC = copy._deepcopy_list(tmp, memo)
612                 #print "### While Iteration:",it
613                 tmp = while_denning(it,  called_by_fun, fun_global,
    while_str, PC[:])
614                 ls = duplicateRemoval(tmp)
615                 memo = {}
616                 PC = copy._deepcopy_list(ls, memo)
617                 #print lastPC,"|--|", PC
618                 #print "-> PC:",PC
619                 if lastPC == PC:
620                     #print "Saturation point of loop!"
621                     break
622                 it += 1
623         i += 1
624     debugPrint(recCount)
625     recCount -= 1;
626     debugPrint("END continuous parse:")
627     return PC[:]
628
629
630 ################################## main
    ############################################
631 with open(sys.argv[1], "r") as inputfile:
632     # data = inputfile.read().replace('\n', '').replace(' ','')
633     data = "".join(inputfile.read().split())
634 llist = []
```

```
635 dummy = []
636 su_sr_list = []
637 line = 0
638 PC = [[]]
639 gg = []
640 output = []
641
642 fun_call_index = [m.start() for m in re.finditer("FunctionDef", data)]
643 for index in fun_call_index:
644     varName = extract_variavle_name(index+len("FunctionDef(name='"),data)
645     fun_hash[varName] = index
646
647 #G = Global(data)
648 continuous_parse("", [], dummy, data, PC)
649
650 for it in list(unique_everseen(output)):
651     print it
```

Appendix C

Python Script category 3: Constraint Generator

```
#INPUT P - the set of principals that have a stake in the computation.
#       p - computing athority
#       S - set of all principals in system.
#       label file
from  more_itertools import unique_everseen
import sys
import re, pdb
import copy
import itertools

iteration = 20
debug = 0
def debugPrint(x):
    if debug == 1:
        print x

class const:
    otime = "*"  # u"\u2295"
    oplus = "+"  # u"\u2297"
    lt = "<="  # u"\u2264"


def SemanticsOfProgram(P,p,c,PC,S):
    if p not in P:
        print "MISSUSE";
    for x in AccessedGlobal(c):
```

```
27          if p not in R(lamda(x)):
28              print "MISSUSE";
29      #intialization
30      for x in Global(c):
31          M[x] = Md[x]
32          lamda[x] = lamdad[x]
33      for x in ((VA(c) - Global(c)) | set(PC)):
34          M[x] = 0
35          lamda[x] = (p,S,set([p]))
36
37  def VA(data):
38      return set(parse_variables(data))
39
40  def Global(data): #discard all whiles, ifs and functions -> then remaining
        code will have only globals.
41      str = ""
42      length = len(data)
43      i = 0
44      while i < length - 1:
45          # checking for keyword
46          if parse_keyword(i, data) == "FunctionDef":
47              i += 11
48              i = parse_parenthesis(i, data)[1]
49          elif parse_keyword(i, data) == "Expr(":
50              i += 4
51              i = parse_parenthesis(i, data)[1]
52          elif parse_keyword(i, data) == "AugAssign":
53              i += 9
54              i = parse_parenthesis(i, data)[1]
55          elif parse_keyword(i, data) == "If":
56              i += 2
57              i = parse_if(i, data)[1]
58
59          elif parse_keyword(i, data) == "While":
60              i += 5
61              i = parse_parenthesis(i, data)[1]
62          else:
63              str += data[i]
64          i += 1
65      return set(parse_variables(str))
66
67  def parseTestVariables(data):
```

```
68      test_index = [ m.start() for m in re.finditer('test=', data) ]
69      ll = []
70      for it in test_index:
71          ll += parse_variables(parse_next_parenthesis(it,str)[0])
72      return ll
73
74  def AccessedGlobal(data):
75      #(i)right-hand side of assignment
76      #(ii) condition of branching/iteration
77      #(iii) return
78      ll = []
79      length = len(data)
80      i = 0
81      while i < length - 1:
82          # checking for keyword
83          if parse_keyword(i, data) == "AugAssign":
84              i += 9
85              tmp = parse_parenthesis(i, data)
86              ll += parse_variables(tmp[0])
87              i = tmp[1]
88          elif parse_keyword(i, data) == "Assign":
89              i += 6
90              tmp = parse_parenthesis(i, data)
91              ryt = tmp[0].split("value=")[1]
92              ll += parse_variables(ryt)
93              i = tmp[1]
94          elif parse_keyword(i, data) == "If":
95              i += 2
96              tmp = parse_parenthesis(i, data)
97              ll += parseTestVariables(tmp[0])
98              i = tmp[1]
99          elif parse_keyword(i, data) == "While":
100             i += 5
101             tmp = parse_parenthesis(i, data)
102             ll += parseTestVariables(tmp[0])
103             i = tmp[1]
104         elif parse_keyword(i,data) == "Return":
105             i += 6
106             tmp = parse_parenthesis(i,data)
107             ll += parse_variables(tmp[0])
108             i = tmp[1]
109         i += 1
```

```
110     return set(ll)
111
112 def ModifiedGlobal(data):
113     ss = target_of_assignment(data)
114     modifiedVarList = parse_variables(ss)
115     return Global(data) & set(modifiedVarList)
116
117
118 def make_lub_string(llist):  # assumption list containing string elemnts
119     if len(llist) == 0:
120         return "Low"
121     if len(llist) == 1:
122         return llist[0]
123     tmp = set(llist)
124     uniq_list = list(tmp)
125     if len(uniq_list) == 1:
126         return str(uniq_list[0])
127     ret = ""
128     ret += uniq_list[0]
129     i = 1
130     while i < len(uniq_list):
131         ret += " " + const.oplus + " "
132         ret += uniq_list[i]
133         i += 1
134     return ret
135
136
137 def make_glb_string(llist):  # assumption list containing string elemnts
138     # type: (list) -> string
139     if len(llist) == 0:
140         return "High"
141     if len(llist) == 1:
142         return llist[0]
143     uniq_list = list(set(llist))
144     if len(uniq_list) == 1:
145         return uniq_list[0]
146     ret = ""
147     ret += uniq_list[0]
148     i = 1
149     while i < len(uniq_list):
150         ret += " " + const.otime + " "
151         ret += uniq_list[i]
```

```
152         i += 1
153     return ret
154 
155 
156 def split_through_orelse(if_str):
157     # find first body word
158     i = if_str.find("body=[")
159     i = parse_square_br(i + 5, if_str)[1] + 1
160     return ["{" + if_str[1:i] + "}", if_str[i + 7:]]
161 
162 
163 def parse_keyword(i, data):
164     # checking for
165     funLen =  len("Expr(value=Call(func=Name(id='")
166     attrLen = len("Expr(value=Call(func=Attribute(value=Name(id='")
167 
168     if i + 6 < len(data) - 1 and data[i:i + 6] == 'Assign':
169         return "Assign"
170     if i + 9 < len(data) - 1 and data[i:i + 9] == 'AugAssign':
171         return "AugAssign"
172     if i + 2 < len(data) - 1 and data[i:i + 2] == 'If':
173         return "If"
174     if i + 5 < len(data) - 1 and data[i:i + 5] == 'While':
175         return "While"
176     if i + 11 < len(data) - 1 and data[i:i + 11] == 'FunctionDef':
177         return "FunctionDef"
178     if i + 6 < len(data) - 1 and data[i:i+6] == "Return":
179         return "Return"
180     if i +  funLen < len(data) - 1 and data[i:i + funLen] == "Expr(value=
    Call(func=Name(id='":
181   return  "fun_call"
182     if i + attrLen  < len(data) - 1 and data[i:i + attrLen ] == "Expr(value
    =Call(func=Attribute(value=Name(id='":
183         if extract_variavle_name(i+attrLen,data) == 'thread':
184             return "thread_fun_call"
185         else:
186       return "set_clear_wait"
187     return "none"
188 
189 def parse_square_br(i, data):
190     if data[i] != '[':
191         print "Error: [ is missing"
```

```
192         return []
193     ret = "["
194     count = 1
195     i += 1
196     while count > 0 and i < len(data) - 1:
197         if data[i] == '[':
198             count += 1
199         if data[i] == ']':
200             count -= 1
201         ret += data[i]
202         i += 1
203     return [ret, i]
204 
205 
206 def parse_parenthesis(i, data):
207     # type: (int , string) -> string
208     if data[i] != '(':
209         print data[i-4:i+4],data[i]
210         print "Error: ( is missing"
211         return []
212     ret = "("
213     count = 1
214     i += 1
215     while count > 0 and i < len(data) - 1:
216         if data[i] == '(':
217             count += 1
218         if data[i] == ')':
219             count -= 1
220         ret += data[i]
221         i += 1
222     return [ret, i]
223 
224 def parse_next_parenthesis(i,data):
225     while i < len(data)-1 and data[i] != '(':
226         i+=1
227     if i == len(data)-1:
228         print "No parenthesis in string"
229         return ["",i]
230     ret = "("
231     count = 1
232     i += 1
233     while count > 0 and i < len(data) - 1:
```

```
234         if data[i] == '(':
235             count += 1
236         if data[i] == ')':
237             count -= 1
238         ret += data[i]
239         i += 1
240     return [ret, i]
241 
242 def extract_variavle_name(startpos, line):
243     # string->string
244     var = ""
245     while line[startpos] != "'":
246         var += line[startpos]
247         startpos += 1
248     return var
249 
250 def target_of_assignment(str):  # find all targets
251     # string->list
252     targets_ptrn = r"targets=\[.*?\]"
253     ctargets_ptrn = re.compile(targets_ptrn)
254     temp_list = ctargets_ptrn.findall(str)
255     ret = ''.join(temp_list)  # converting to string
256     return ret
257 
258 
259 def parse_variables(line):
260     # type: (string) -> list
261     # type: (str) -> object
262     id_index = [m.start() for m in re.finditer('id=', line)]
263     var_list = []
264     for it in id_index:
265         vname = extract_variavle_name(it + 4, line)
266         if vname == "False" or vname == "True":
267             continue
268         var_list.append(vname)
269     return var_list
270 
271 
272 def multiple_assign(assign_str, target_id_index, PC):
273     global line
274     tmp = assign_str.split("value", 1)
275     rvalue = parse_variables(tmp[0])
```

```
276     lvalue = parse_variables(tmp[1])
277     pc_update(PC,lvalue)
278 
279     # printing denning's rule
280     for it in rvalue:
281         # left = make_lub_string(dict[key])
282         if len(lvalue) == 0:
283             print "low " + const.lt + " " + it
284             line += 1
285         else:
286             print make_lub_string(lvalue), const.lt, it
287             line += 1
288     return PC[:]
289 
290 def pc_update(PC, list):
291     #print "pre update", PC
292     for pc in PC:
293         pc += list
294     #print "post update",PC
295 def label_to_lub(llist):
296     if len(llist) == 1:
297         return llist[0]
298     ret = ""
299     ret += llist[0]
300     i = 1
301     while i<len(llist):
302         ret += " "+const.oplus+" "+ llist[i]
303         i +=1
304     return ret
305 
306 
307 def make_lub_string_label(llist):
308     if len(llist) == 0:
309         return "Low"
310     if len(llist) == 1:
311         return label_to_lub(label[llist[0]])
312     tmp = set(llist)
313     uniq_list = list(tmp)
314     if len(uniq_list) == 1:
315         return label_to_lub(label[uniq_list[0]])
316     ret = ""
317     ret += label_to_lub(label[uniq_list[0]])
```

```
318     i = 1
319     while i < len(uniq_list):
320         ret += " " + const.oplus + " "
321         ret += label_to_lub(label[uniq_list[i]])
322         i += 1
323     return ret
324 
325 def convertToLabelList(ll):
326     llist = list(set(ll))
327     global label
328     if len(llist) == 0:
329         return llist
330     ret = []
331     for it in llist:
332         if it not in label:
333             ret += [it]
334             continue
335         ret += label[it]
336     return list(set(ret))
337 
338 def make_lubPC(PC):
339     llist = []
340     for lt in PC:
341         llist += lt
342     return list(set(llist))
343 
344 def assign_denning(G,assign_str, PC):  # applying dennig's model on
    assignments
345     #pdb.set_trace()
346     global line
347     global label
348     global output
349     ss = assign_str.split("value")
350     target_id_index = [m.start() for m in re.finditer('id=', ss[0])]
351 
352     if len(target_id_index) > 1:
353         return multiple_assign(assign_str, target_id_index, PC)
354 
355     if "id='" in ss[0]:
356         left = extract_variavle_name(0, ss[0].split("id='")[1])
357     else:
358         left = ['const']
```

```
359     id_index = [m.start() for m in
360             re.finditer('id=', ss[1])]  # list of starting index of
    variables in right part of string
361     rvalue = []
362     if len(id_index) == 0:
363         # rvalue.append("low")
364         pass
365     else:
366         for it in id_index:
367             startpos = it + 4
368             vname = extract_variavle_name(startpos, ss[1])
369             """Exclusion of False keyword"""
370             if vname == "False" or vname == "True":
371                 continue
372             rvalue.append(vname)
373     #pc_update(PC,rvalue)
374 
375     ret = ""
376     l = convertToLabelList(rvalue)
377     #ll = l + lambda(pc)
378     #updating PC
379     if left in G:
380         for pc in PC:
381             output.append(make_lub_string(convertToLabelList(pc)+l)+ " " +
    const.lt + " " + make_lub_string(label[left]))
382     else:
383         # update dynamic label, for all x (MNG(BR) union {pc}) [lambda(x)=l
    +lambda(x)+lambda(pc)]
384         if left not in label:
385             label[left] = []
386 
387         label[left] =  list(set(label[left] + make_lubPC(PC[:]+[l])))
388         if left in label[left]:
389             label[left].remove(left)
390 
391         #print label[left],"changed", left
392     line += 1
393     return PC[:]
394 
395 
396 def augAssign_denning(called_by_fun, fun_global, augAssign_str, PC):
397     # i = augAssign_str.find("id=")
```

```
398     # Su = [extract_variavle_name(i + 4, augAssign_str)]
399     global output
400     Sr = parse_variables(augAssign_str)
401     for pc in PC:
402         output.append( make_lub_string(pc) + " " + const.lt + " " + Sr[0])
403     return PC[:]
404 
405 PCA = []
406 
407 def if_denning(G,called_by_fun, fun_global, if_str, rest, PC):
408     # type: (list, list, string, dict, string) -> print rules
409     #pdb.set_trace()
410     #print "begin IF",PC
411     if "orelse=" not in if_str:
412         # print "termination",if_str
413         if if_str[0:2] == "[]":  # absence of else part
414             return []
415         else:  # handling else part
416             else_str = if_str
417             continuous_parse(G,[], ccalled_by_fun, fun_global, else_str, PC
    [:])
418             return []
419 
420     tmp = split_through_orelse(if_str)
421     if_half = tmp[0]
422     ladder = tmp[1]
423 
424     if if_str[1:5] != "test":
425         print "Error test not found in if"
426 
427     """extract test=...() from if_half"""
428     i = if_half.find("(")
429     tmp = parse_parenthesis(i, if_half)
430     test_str = tmp[0]
431     #parent_list += parse_variables(test_str)
432     i = tmp[1]
433     pc_update(PC,parse_variables(test_str))
434 
435     """then extract body part and process like normal AST text """
436     # body processing
437     body_onward_str = if_half[i:]  ### Asumption : Compare string always
    followed by body=[...] imediatly
```

```
438     # setting i to location of [ in body_str: ,body=[..
439     i = body_onward_str.find("[")
440     body_str = parse_square_br(i, body_onward_str)[0]
441     memo = {}
442     memo2 = {}
443     PC1 = copy._deepcopy_list(PC,memo)
444     PC2 = copy._deepcopy_list(PC, memo2)
445     PCA = list(continuous_parse(G,[], called_by_fun, fun_global, body_str,
        PC1[:]))
446     PCB = list(continuous_parse(G,[], called_by_fun, fun_global, ladder,
       PC2[:]))
447     #debugPrint(PCA)
448     #debugPrint(PCB)
449     #print "end IF",PCB + PCA
450     return PCB + PCA
451
452 def while_denning(iteration, G, called_by_fun, fun_global, while_str, PC):
453     # debug print "printing while_str",while_str[6:10]
454     compare = "()"
455     if while_str[6:10] == "Name":
456         tmp = parse_parenthesis(10, while_str)
457         compare = tmp[0]
458         i = tmp[1]
459     elif while_str[6:10] == "Comp":
460         tmp = parse_parenthesis(13, while_str)
461         compare = tmp[0]
462         i = tmp[1]
463     pc_update(PC, parse_variables(compare))
464     # body processing
465     body_onward_str = while_str[i:]  ### Asumption : Compare string always
       followed by body=[...] imediatly
466     # setting i to location of [ in body_str: ,body=[..
467     i = body_onward_str.find("[")
468     body_str = parse_square_br(i, body_onward_str)[0]
469     memo = {}
470     memo2 = {}
471     PC1 = copy._deepcopy_list(PC, memo)
472     PC2 = copy._deepcopy_list(PC, memo2)
473
474     PCB = list(continuous_parse(G, [], called_by_fun, fun_global, body_str,
        PC2[:]))
475     return PC1 + PCB
```

```
476
477 def set_clear_denning(G, called_by_fun, fun_global, expr_str, PC):
478     #ASSUMPTION SEMAPHORE VAR IS ALWAYS GLOBAL
479     global output
480     i = expr_str.find("Call(")
481     i += 4
482     call_str = parse_parenthesis(i, expr_str)[0]
483     i = call_str.find("Attribute(")
484     i += 9
485     attribute_str = parse_parenthesis(i, call_str)[0]
486     i = attribute_str.find("Name(")
487     i += 4
488     # name_str = parse_parenthesis(i,attribute_str)[0]
489     i = attribute_str.find("id=")
490     var_name = extract_variavle_name(i + 4, attribute_str)
491     i = attribute_str.find("attr=")
492     attr = extract_variavle_name(i + 6, attribute_str)
493     if attr == "set" or attr == "clear":
494         # treat it like AugAssign s0 += 1
495         # treat it like AugAssign s0 -= 1
496         #pc_update(PC,convertToLabelList([var_name]))
497         #print "set clear-> ",PC
498         for pc in PC:
499             output.append( make_lub_string(pc+[var_name])+ " " + const.lt +
    " " + make_lub_string(label[var_name])) #label[left]
500     elif attr == "wait":
501         #global_while_list.append(var_name)
502         #pc_update(PC,convertToLabelList([var_name]))
503         pass
504         #print "wait -> ",PC
505
506 def extract_Globals(fun_str):
507     global_index = [m.start() for m in re.finditer("Global\(", fun_str)]
508     globals = {}
509     for it in global_index:
510         global_str = parse_parenthesis(it + 6, fun_str)[0]
511         sq_str = parse_square_br(global_str.find("["), global_str)[0]
512         ss = sq_str.strip("[").strip("]")
513         sslist = ss.split(",")
514         for it in sslist:
515             if it == '':
516                 continue
```

```
517             globals[(it.strip("'"))] = 1
518     return globals
519
520
521 def fun_denning(fun_str,PC):
522     fun_name = extract_variavle_name(fun_str.find("name=") + 6, fun_str)
523     fun_globals = extract_Globals(fun_str)
524     funPC = []
525     PC = continuous_parse(G,funPC, fun_name, fun_globals, fun_str,PC)
526     return PC + funPC
527
528 # global var for counting
529 ww = ww1 = ww2 = 1
530
531 # global while list
532 global_while_list = []
533
534
535 def parse_if(i, data):
536     tmp = parse_parenthesis(i, data)
537     i = tmp[1]
538     if_str = tmp[0]
539     rest = data[i:]
540     return [if_str, i, rest]
541
542 def uniq(l):
543     ll = []
544     for it in l:
545         ll.append(list(set(it)))
546     return ll
547
548 fun_hash = {}
549
550 def duplicateRemoval(PC):
551     tmpPC = []
552     for pc in PC:
553         tmppc = list(set(pc))
554         tmppc.sort()
555         tmpPC.append(tmppc)
556     tmpPC.sort()
557     return list(tmpPC for tmpPC,_ in itertools.groupby(tmpPC))
558
```

```
559 recCount =0
560 def continuous_parse(G, funPC, called_by_fun, fun_global, data, PC):
561     #pdb.set_trace()
562     # type: (object, object) -> object
563     global recCount
564     length = len(data)
565     i = 0
566     recCount += 1
567     debugPrint(recCount)
568     debugPrint("Continuous_parse before loop:")
569     while i < length - 1:
570         #checking for keyword
571         if parse_keyword(i,data) == "FunctionDef":  #skipping all function
    definition
572             tmp = parse_parenthesis(i+11, data)
573             i = tmp[1]
574         if parse_keyword(i,data) == "thread_fun_call": #parsing function
    call used in threads
575             tmp = parse_parenthesis(i+4,data)
576             i = tmp[1]
577             if 'start_new_thread' in tmp[0]:
578                 #print "got thread call"
579                 fi = tmp[0].find("args=[")
580                 funName = extract_variavle_name(fi+15,tmp[0]) #len(args=[
    Name(id=') = 15
581                 if funName in fun_hash:
582                     findex = fun_hash[funName]
583                 else:
584                     print "Function not found but prgram called function"
585                 findex += 11
586                 tmp = parse_parenthesis(findex, data)
587                 fun_str = tmp[0]
588                 tmp = fun_denning(fun_str,[[]])
589                 memo = {}
590                 ls = duplicateRemoval(tmp)
591                 PC = copy._deepcopy_list(ls, memo)
592
593         if parse_keyword(i,data) == "fun_call":  #parsing functioncalls
594             tmp = parse_parenthesis(i+4,data)
595             #print "got fun call"
596             funName = extract_variavle_name(i + len("Expr(value=Call(func=
    Name(id='"),data)
```

```
597             i = tmp[1]
598             if funName in fun_hash:
599                 findex = fun_hash[funName]
600             else:
601                 print "Function not found but prgram called function"
602             findex += 11
603             tmp = parse_parenthesis(findex, data)
604             fun_str = tmp[0]
605             tmp = fun_denning(fun_str,PC)
606             memo = {}
607             ls = duplicateRemoval(tmp)
608             PC = copy._deepcopy_list(ls, memo)
609 
610         if parse_keyword(i,data) == "Return":
611             debugPrint("Return stmt:")
612             funPC += PC
613         if parse_keyword(i, data) == "set_clear_wait":
614             i += 4
615             tmp = parse_parenthesis(i, data)
616             expr_str = tmp[0]
617             if expr_str.find("'set'") != -1 or expr_str.find("'clear'") !=
    -1 or expr_str.find("'wait'") != -1 :
618                 i = tmp[1]
619                 set_clear_denning(G, called_by_fun, fun_global, expr_str,
    PC)
620         if parse_keyword(i, data) == "AugAssign":
621             i += 9
622             tmp = parse_parenthesis(i, data)
623             augAssign_str = tmp[0]
624             i = tmp[1]
625             tmp = augAssign_denning(parent_list[:], global_while_list,
    called_by_fun, fun_global, augAssign_str, PC[:])
626 
627         if parse_keyword(i, data) == "Assign":
628             global ww
629             ww += 1
630             i += 6
631             tmp = parse_parenthesis(i, data)
632             assign_str = tmp[0]
633             i = tmp[1]
634             if "value=Name(id='threading'" in assign_str:
635                 continue
```

```
636             assign_denning(G, assign_str , PC[:])
637         elif parse_keyword(i, data) == "If":
638             debugPrint("If stmt:")
639             ls = duplicateRemoval(PC)
640             memo = {}
641             PC_reset = copy._deepcopy_list(ls , memo)
642             global ww1
643             ww1 += 1
644             i += 2
645             tmp = parse_if(i, data)
646             if_str = tmp[0]
647             i = tmp[1]
648             rest = tmp[2]
649             if_denning(G, called_by_fun , fun_global , if_str , rest , PC[:])
650             memo = {}
651             PC = copy._deepcopy_list(PC_reset , memo)
652         elif parse_keyword(i, data) == "While":
653             debugPrint("while stmt:")
654             global ww2
655             ww2 += 1
656             i += 5
657             tmp = parse_parenthesis(i, data)
658             while_str = tmp[0]
659             i = tmp[1]
660             it = 1
661             ls = duplicateRemoval(PC)
662             memo = {}
663             PC_reset = copy._deepcopy_list(ls , memo)
664             while( it <= iteration ):
665                 memo = {}
666                 lastPC = copy._deepcopy_list(PC, memo)
667                 tmp = duplicateRemoval(PC)
668                 memo = {}
669                 PC = copy._deepcopy_list(tmp, memo)
670                 #print "### While Iteration:",it
671                 tmp = while_denning(it , G, called_by_fun , fun_global ,
    while_str , PC[:])
672                 ls = duplicateRemoval(tmp)
673                 memo = {}
674                 PC = copy._deepcopy_list(ls , memo)
675                 #print lastPC ,"|--|", PC
676                 #print "-> PC:",PC
```

```
677                 if lastPC == PC:
678                     #print "Saturation point of loop!",PC,lastPC
679                     break
680                 it += 1
681             memo = {}
682             PC = copy._deepcopy_list(PC_reset, memo)
683         i += 1
684     debugPrint(recCount)
685     recCount -= 1;
686     debugPrint("END continuous parse:")
687     return PC[:]
688 
689 def process_label_fille(filename):
690     # a = x,y,z
691     label = {}
692     with open(filename, "r") as inputfile:
693         for line in inputfile:
694             ll = "".join(line.split())
695             tmp = ll.split("=")
696             var = tmp[0]
697             lineList = tmp[1].split(",")
698             label[var] = lineList
699     return label
700 
701 ################################### main
    ##############################################
702 with open(sys.argv[1], "r") as inputfile:
703     # data = inputfile.read().replace('\n', '').replace(' ','')
704     data = "".join(inputfile.read().split())
705 llist = []
706 dummy = []
707 su_sr_list = []
708 line = 0
709 PC = [[]]
710 label = {}
711 label = process_label_fille(sys.argv[2])
712 gg = []
713 output = []
714 for key in label:
715     gg.append(key)
716 G = set(gg)
717 fun_call_index = [m.start() for m in re.finditer("FunctionDef", data)]
```

```
718 for index in fun_call_index:
719     varName = extract_variavle_name(index+len("FunctionDef(name='"),data)
720     fun_hash[varName] = index
721
722 #G = Global(data)
723 continuous_parse(G,"", [], dummy, data, PC)
724
725 for it in list(unique_everseen(output)):
726     print it
```

Appendix D

Python Script category 4: Constraint Generator

```
#INPUT P - the set of principals that have a stake in the computation.
#       p - computing athority
#       S - set of all principals in system.
#       label file
from  more_itertools import unique_everseen
import sys
import re, pdb
import copy
import itertools

iteration = 20
debug = 0
def debugPrint(x):
    if debug == 1:
        print x

class const:
    otime = "*"  # u"\u2295"
    oplus = "+"  # u"\u2297"
    lt = "<="  # u"\u2264"


def SemanticsOfProgram(P,p,c,PC,S):
    if p not in P:
        print "MISSUSE";
    for x in AccessedGlobal(c):
```

```
27          if p not in R(lamda(x)):
28              print "MISSUSE";
29      #intialization
30      for x in Global(c):
31          M[x] = Md[x]
32          lamda[x] = lamdad[x]
33      for x in ((VA(c) - Global(c)) | set(PC)):
34          M[x] = 0
35          lamda[x] = (p,S,set([p]))
36
37  def VA(data):
38      return set(parse_variables(data))
39
40  def Global(data): #discard all whiles, ifs and functions -> then remaining
        code will have only globals.
41      str = ""
42      length = len(data)
43      i = 0
44      while i < length - 1:
45          # checking for keyword
46          if parse_keyword(i, data) == "FunctionDef":
47              i += 11
48              i = parse_parenthesis(i, data)[1]
49          elif parse_keyword(i, data) == "Expr(":
50              i += 4
51              i = parse_parenthesis(i, data)[1]
52          elif parse_keyword(i, data) == "AugAssign":
53              i += 9
54              i = parse_parenthesis(i, data)[1]
55          elif parse_keyword(i, data) == "If":
56              i += 2
57              i = parse_if(i, data)[1]
58
59          elif parse_keyword(i, data) == "While":
60              i += 5
61              i = parse_parenthesis(i, data)[1]
62          else:
63              str += data[i]
64          i += 1
65      return set(parse_variables(str))
66
67  def parseTestVariables(data):
```

```
68      test_index = [ m.start() for m in re.finditer('test=', data) ]
69      ll = []
70      for it in test_index:
71          ll += parse_variables(parse_next_parenthesis(it,str)[0])
72      return ll
73
74  def AccessedGlobal(data):
75      #(i)right-hand side of assignment
76      #(ii) condition of branching/iteration
77      #(iii) return
78      ll = []
79      length = len(data)
80      i = 0
81      while i < length - 1:
82          # checking for keyword
83          if parse_keyword(i, data) == "AugAssign":
84              i += 9
85              tmp = parse_parenthesis(i, data)
86              ll += parse_variables(tmp[0])
87              i = tmp[1]
88          elif parse_keyword(i, data) == "Assign":
89              i += 6
90              tmp = parse_parenthesis(i, data)
91              ryt = tmp[0].split("value=")[1]
92              ll += parse_variables(ryt)
93              i = tmp[1]
94          elif parse_keyword(i, data) == "If":
95              i += 2
96              tmp = parse_parenthesis(i, data)
97              ll += parseTestVariables(tmp[0])
98              i = tmp[1]
99          elif parse_keyword(i, data) == "While":
100             i += 5
101             tmp = parse_parenthesis(i, data)
102             ll += parseTestVariables(tmp[0])
103             i = tmp[1]
104         elif parse_keyword(i,data) == "Return":
105             i += 6
106             tmp = parse_parenthesis(i,data)
107             ll += parse_variables(tmp[0])
108             i = tmp[1]
109         i += 1
```

```
110     return set(ll)
111
112 def ModifiedGlobal(data):
113     ss = target_of_assignment(data)
114     modifiedVarList = parse_variables(ss)
115     return Global(data) & set(modifiedVarList)
116
117
118 def make_lub_string(llist):  # assumption list containing string elemnts
119     if len(llist) == 0:
120         return "Low"
121     if len(llist) == 1:
122         return llist[0]
123     tmp = set(llist)
124     uniq_list = list(tmp)
125     if len(uniq_list) == 1:
126         return str(uniq_list[0])
127     ret = ""
128     ret += uniq_list[0]
129     i = 1
130     while i < len(uniq_list):
131         ret += " " + const.oplus + " "
132         ret += uniq_list[i]
133         i += 1
134     return ret
135
136
137 def make_glb_string(llist):  # assumption list containing string elemnts
138     # type: (list) -> string
139     if len(llist) == 0:
140         return "High"
141     if len(llist) == 1:
142         return llist[0]
143     uniq_list = list(set(llist))
144     if len(uniq_list) == 1:
145         return uniq_list[0]
146     ret = ""
147     ret += uniq_list[0]
148     i = 1
149     while i < len(uniq_list):
150         ret += " " + const.otime + " "
151         ret += uniq_list[i]
```

```
152         i += 1
153     return ret
154 
155 
156 def split_through_orelse(if_str):
157     # find first body word
158     i = if_str.find("body=[")
159     i = parse_square_br(i + 5, if_str)[1] + 1
160     return ["{" + if_str[1:i] + "}", if_str[i + 7:]]
161 
162 
163 def parse_keyword(i, data):
164     # checking for
165     funLen =  len("Expr(value=Call(func=Name(id='")
166     attrLen = len("Expr(value=Call(func=Attribute(value=Name(id='")
167 
168     if i + 6 < len(data) - 1 and data[i:i + 6] == 'Assign':
169         return "Assign"
170     if i + 9 < len(data) - 1 and data[i:i + 9] == 'AugAssign':
171         return "AugAssign"
172     if i + 2 < len(data) - 1 and data[i:i + 2] == 'If':
173         return "If"
174     if i + 5 < len(data) - 1 and data[i:i + 5] == 'While':
175         return "While"
176     if i + 11 < len(data) - 1 and data[i:i + 11] == 'FunctionDef':
177         return "FunctionDef"
178     if i + 6 < len(data) - 1 and data[i:i+6] == "Return":
179         return "Return"
180     if i +  funLen < len(data) - 1 and data[i:i + funLen] == "Expr(value=
    Call(func=Name(id='":
181   return  "fun_call"
182     if i + attrLen  < len(data) - 1 and data[i:i + attrLen ] == "Expr(value
    =Call(func=Attribute(value=Name(id='":
183         if extract_variavle_name(i+attrLen,data) == 'thread':
184             return "thread_fun_call"
185         else:
186       return "set_clear_wait"
187     return "none"
188 
189 def parse_square_br(i, data):
190     if data[i] != '[':
191         print "Error: [ is missing"
```

```
192         return []
193     ret = "["
194     count = 1
195     i += 1
196     while count > 0 and i < len(data) - 1:
197         if data[i] == '[':
198             count += 1
199         if data[i] == ']':
200             count -= 1
201         ret += data[i]
202         i += 1
203     return [ret, i]
204 
205 
206 def parse_parenthesis(i, data):
207     # type: (int , string) -> string
208     if data[i] != '(':
209         print data[i-4:i+4],data[i]
210         print "Error: ( is missing"
211         return []
212     ret = "("
213     count = 1
214     i += 1
215     while count > 0 and i < len(data) - 1:
216         if data[i] == '(':
217             count += 1
218         if data[i] == ')':
219             count -= 1
220         ret += data[i]
221         i += 1
222     return [ret, i]
223 
224 def parse_next_parenthesis(i,data):
225     while i < len(data)-1 and data[i] != '(':
226         i+=1
227     if i == len(data)-1:
228         print "No parenthesis in string"
229         return ["",i]
230     ret = "("
231     count = 1
232     i += 1
233     while count > 0 and i < len(data) - 1:
```

```
234         if data[i] == '(':
235             count += 1
236         if data[i] == ')':
237             count -= 1
238         ret += data[i]
239         i += 1
240     return [ret, i]
241 
242 def extract_variavle_name(startpos, line):
243     # string->string
244     var = ""
245     while line[startpos] != "'":
246         var += line[startpos]
247         startpos += 1
248     return var
249 
250 def target_of_assignment(str):  # find all targets
251     # string->list
252     targets_ptrn = r"targets=\[.*?\]"
253     ctargets_ptrn = re.compile(targets_ptrn)
254     temp_list = ctargets_ptrn.findall(str)
255     ret = ''.join(temp_list)  # converting to string
256     return ret
257 
258 
259 def parse_variables(line):
260     # type: (string) -> list
261     # type: (str) -> object
262     id_index = [m.start() for m in re.finditer('id=', line)]
263     var_list = []
264     for it in id_index:
265         vname = extract_variavle_name(it + 4, line)
266         if vname == "False" or vname == "True":
267             continue
268         var_list.append(vname)
269     return var_list
270 
271 
272 def multiple_assign(assign_str, target_id_index, PC):
273     global line
274     tmp = assign_str.split("value", 1)
275     rvalue = parse_variables(tmp[0])
```

```
276     lvalue = parse_variables(tmp[1])
277     pc_update(PC,lvalue)
278 
279     # printing denning's rule
280     for it in rvalue:
281         # left = make_lub_string(dict[key])
282         if len(lvalue) == 0:
283             print "low " + const.lt + " " + it
284             line += 1
285         else:
286             print make_lub_string(lvalue), const.lt, it
287             line += 1
288     return PC[:]
289 
290 def pc_update(PC, list):
291     #print "pre update", PC
292     for pc in PC:
293         pc += list
294     #print "post update",PC
295 def label_to_lub(llist):
296     if len(llist) == 1:
297         return llist[0]
298     ret = ""
299     ret += llist[0]
300     i = 1
301     while i<len(llist):
302         ret += " "+const.oplus+" "+ llist[i]
303         i +=1
304     return ret
305 
306 
307 def make_lub_string_label(llist):
308     if len(llist) == 0:
309         return "Low"
310     if len(llist) == 1:
311         return label_to_lub(label[llist[0]])
312     tmp = set(llist)
313     uniq_list = list(tmp)
314     if len(uniq_list) == 1:
315         return label_to_lub(label[uniq_list[0]])
316     ret = ""
317     ret += label_to_lub(label[uniq_list[0]])
```

```
318     i = 1
319     while i < len(uniq_list):
320         ret += " " + const.oplus + " "
321         ret += label_to_lub(label[uniq_list[i]])
322         i += 1
323     return ret
324 
325 def convertToLabelList(llist):
326     global label
327     if len(llist) == 0:
328         return llist
329     ret = []
330     for it in llist:
331         if it not in label:
332             ret += [it]
333             continue
334         ret += label[it]
335     return list(set(ret))
336 
337 def assign_denning(G,assign_str, PC):  # applying dennig's model on
    assignments
338     #pdb.set_trace()
339     global line
340     global label
341     global output
342     ss = assign_str.split("value")
343     target_id_index = [m.start() for m in re.finditer('id=', ss[0])]
344 
345     if len(target_id_index) > 1:
346         return multiple_assign(assign_str, target_id_index, PC)
347 
348     if "id='" in ss[0]:
349         left = extract_variavle_name(0, ss[0].split("id='")[1])
350     else:
351         left = ['const']
352     id_index = [m.start() for m in
353             re.finditer('id=', ss[1])]  # list of starting index of
    variables in right part of string
354     rvalue = []
355     if len(id_index) == 0:
356         # rvalue.append("low")
357         pass
```

```
358     else:
359         for it in id_index:
360             startpos = it + 4
361             vname = extract_variavle_name(startpos, ss[1])
362             """Exclusion of False keyword"""
363             if vname == "False" or vname == "True":
364                 continue
365             rvalue.append(vname)
366     #pc_update(PC,rvalue)
367 
368     ret = ""
369     l = convertToLabelList(rvalue)
370     #ll = l + lambda(pc)
371     #updating PC
372     for pc in PC:
373         pc += l
374     if left in G:
375         for pc in PC:
376             output.append(make_lub_string(pc)+ " " + const.lt + " " +
    make_glb_string(label[left]))
377     else:
378         # update dynamic label, for all x (MNG(BR) union {pc}) [lambda(x)=l
    +lambda(x)+lambda(pc)]
379         if left not in label:
380             label[left] = []
381         label[left] += l
382         #print label[left],"changed", left
383     line += 1
384     return PC[:]
385 
386 
387 def augAssign_denning(called_by_fun, fun_global, augAssign_str, PC):
388     # i = augAssign_str.find("id=")
389     # Su = [extract_variavle_name(i + 4, augAssign_str)]
390     global output
391     Sr = parse_variables(augAssign_str)
392     pc_update(PC,Sr)
393     for pc in PC:
394         output.append( make_lub_string(pc) + " " + const.lt + " " + Sr[0])
395     return PC[:]
396 
397 PCA = []
```

```
398
399 def if_denning(G,called_by_fun, fun_global, if_str, rest, PC):
400     # type: (list, list, string, dict, string) -> print rules
401     #pdb.set_trace()
402     #print "begin IF",PC
403     if "orelse=" not in if_str:
404         # print "termination",if_str
405         if if_str[0:2] == "[]":  # absence of else part
406             return []
407         else:  # handling else part
408             else_str = if_str
409             continuous_parse(G,[], ccalled_by_fun, fun_global, else_str, PC
    [:])
410             return []
411
412     tmp = split_through_orelse(if_str)
413     if_half = tmp[0]
414     ladder = tmp[1]
415
416     if if_str[1:5] != "test":
417         print "Error test not found in if"
418
419     """extract test=...() from if_half"""
420     i = if_half.find("(")
421     tmp = parse_parenthesis(i, if_half)
422     test_str = tmp[0]
423     #parent_list += parse_variables(test_str)
424     i = tmp[1]
425     pc_update(PC,convertToLabelList(parse_variables(test_str)))
426
427     """then extract body part and process like normal AST text """
428     # body processing
429     body_onward_str = if_half[i:]  ### Asumption : Compare string always
    followed by body=[...] imediatly
430     # setting i to location of [ in body_str: ,body=[..
431     i = body_onward_str.find("[")
432     body_str = parse_square_br(i, body_onward_str)[0]
433     memo = {}
434     memo2 = {}
435     PC1 = copy._deepcopy_list(PC,memo)
436     PC2 = copy._deepcopy_list(PC, memo2)
437     PCA = list(continuous_parse(G,[], called_by_fun, fun_global, body_str ,
```

```
    PC1[:]))
438     PCB = list(continuous_parse(G,[], called_by_fun, fun_global, ladder,
        PC2[:]))
439     #debugPrint(PCA)
440     #debugPrint(PCB)
441     #print "end IF",PCB + PCA
442     return PCB + PCA
443
444 def while_denning(iteration, G, called_by_fun, fun_global, while_str, PC):
445     # debug print "printing while_str",while_str[6:10]
446     compare = "()"
447     if while_str[6:10] == "Name":
448         tmp = parse_parenthesis(10, while_str)
449         compare = tmp[0]
450         i = tmp[1]
451     elif while_str[6:10] == "Comp":
452         tmp = parse_parenthesis(13, while_str)
453         compare = tmp[0]
454         i = tmp[1]
455     pc_update(PC,convertToLabelList(parse_variables(compare)))
456     # body processing
457     body_onward_str = while_str[i:]  ### Asumption : Compare string always
        followed by body=[...] imediatly
458     # setting i to location of [ in body_str: ,body=[..
459     i = body_onward_str.find("[")
460     body_str = parse_square_br(i, body_onward_str)[0]
461     memo = {}
462     memo2 = {}
463     PC1 = copy._deepcopy_list(PC, memo)
464     PC2 = copy._deepcopy_list(PC, memo2)
465
466     PCB = list(continuous_parse(G, [], called_by_fun, fun_global, body_str,
        PC2[:]))
467     return PC1 + PCB
468
469 def set_clear_denning(G, called_by_fun, fun_global, expr_str, PC):
470     #ASSUMPTION SEMAPHORE VAR IS ALWAYS GLOBAL
471     global output
472     i = expr_str.find("Call(")
473     i += 4
474     call_str = parse_parenthesis(i, expr_str)[0]
475     i = call_str.find("Attribute(")
```

```
476     i += 9
477     attribute_str = parse_parenthesis(i, call_str)[0]
478     i = attribute_str.find("Name(")
479     i += 4
480     # name_str = parse_parenthesis(i,attribute_str)[0]
481     i = attribute_str.find("id=")
482     var_name = extract_variavle_name(i + 4, attribute_str)
483     i = attribute_str.find("attr=")
484     attr = extract_variavle_name(i + 6, attribute_str)
485     if attr == "set" or attr == "clear":
486         # treat it like AugAssign s0 += 1
487         # treat it like AugAssign s0 -= 1
488         pc_update(PC,convertToLabelList([var_name]))
489         #print "set clear-> ",PC
490         for pc in PC:
491             output.append( make_lub_string(pc)+ " " + const.lt + " " +
    make_glb_string(label[var_name])) #label[left]
492     elif attr == "wait":
493         #global_while_list.append(var_name)
494         pc_update(PC,convertToLabelList([var_name]))
495         #print "wait -> ",PC
496
497 def extract_Globals(fun_str):
498     global_index = [m.start() for m in re.finditer("Global\(", fun_str)]
499     globals = {}
500     for it in global_index:
501         global_str = parse_parenthesis(it + 6, fun_str)[0]
502         sq_str = parse_square_br(global_str.find("["), global_str)[0]
503         ss = sq_str.strip("[").strip("]")
504         sslist = ss.split(",")
505         for it in sslist:
506             if it == '':
507                 continue
508             globals[(it.strip("'"))] = 1
509     return globals
510
511
512 def fun_denning(fun_str,PC):
513     fun_name = extract_variavle_name(fun_str.find("name=") + 6, fun_str)
514     fun_globals = extract_Globals(fun_str)
515     funPC = []
516     PC = continuous_parse(G,funPC, fun_name, fun_globals, fun_str,PC)
```

```
517     return PC + funPC
518 
519 # global var for counting
520 ww = ww1 = ww2 = 1
521 
522 # global while list
523 global_while_list = []
524 
525 
526 def parse_if(i, data):
527     tmp = parse_parenthesis(i, data)
528     i = tmp[1]
529     if_str = tmp[0]
530     rest = data[i:]
531     return [if_str, i, rest]
532 
533 def uniq(l):
534     ll = []
535     for it in l:
536         ll.append(list(set(it)))
537     return ll
538 
539 fun_hash = {}
540 
541 def duplicateRemoval(PC):
542     tmpPC = []
543     for pc in PC:
544         tmppc = list(set(pc))
545         tmppc.sort()
546         tmpPC.append(tmppc)
547     tmpPC.sort()
548     return list(tmpPC for tmpPC,_ in itertools.groupby(tmpPC))
549 
550 recCount =0
551 def continuous_parse(G, funPC, called_by_fun, fun_global, data, PC):
552     #pdb.set_trace()
553     # type: (object, object) -> object
554     global recCount
555     length = len(data)
556     i = 0
557     recCount += 1
558     debugPrint(recCount)
```

```
559     debugPrint("Continuous_parse before loop:")
560     while i < length - 1:
561         #checking for keyword
562         if parse_keyword(i,data) == "FunctionDef":  #skipping all function
    definition
563             tmp = parse_parenthesis(i+11, data)
564             i = tmp[1]
565         if parse_keyword(i,data) == "thread_fun_call": #parsing function
    call used in threads
566             tmp = parse_parenthesis(i+4,data)
567             i = tmp[1]
568             if 'start_new_thread' in tmp[0]:
569                 #print "got thread call"
570                 fi = tmp[0].find("args=[")
571                 funName = extract_variavle_name(fi+15,tmp[0]) #len(args=[
    Name(id=') = 15
572                 if funName in fun_hash:
573                     findex = fun_hash[funName]
574                 else:
575                     print "Function not found but prgram called function"
576                 findex += 11
577                 tmp = parse_parenthesis(findex, data)
578                 fun_str = tmp[0]
579                 tmp = fun_denning(fun_str,[[]])
580                 memo = {}
581                 ls = duplicateRemoval(tmp)
582                 PC = copy._deepcopy_list(ls, memo)
583
584         if parse_keyword(i,data) == "fun_call":  #parsing functioncalls
585             tmp = parse_parenthesis(i+4,data)
586             #print "got fun call"
587             funName = extract_variavle_name(i + len("Expr(value=Call(func=
    Name(id='"),data)
588             i = tmp[1]
589             if funName in fun_hash:
590                 findex = fun_hash[funName]
591             else:
592                 print "Function not found but prgram called function"
593             findex += 11
594             tmp = parse_parenthesis(findex, data)
595             fun_str = tmp[0]
596             tmp = fun_denning(fun_str,PC)
```

```
597             memo = {}
598             ls = duplicateRemoval(tmp)
599             PC = copy._deepcopy_list(ls, memo)
600 
601         if parse_keyword(i,data) == "Return":
602             debugPrint("Return stmt:")
603             funPC += PC
604         if parse_keyword(i, data) == "set_clear_wait":
605             i += 4
606             tmp = parse_parenthesis(i, data)
607             expr_str = tmp[0]
608             if expr_str.find("'set'") != -1 or expr_str.find("'clear'") !=
    -1 or expr_str.find("'wait'") != -1 :
609                 i = tmp[1]
610                 set_clear_denning(G, called_by_fun, fun_global, expr_str,
    PC)
611         if parse_keyword(i, data) == "AugAssign":
612             i += 9
613             tmp = parse_parenthesis(i, data)
614             augAssign_str = tmp[0]
615             i = tmp[1]
616             tmp = augAssign_denning(parent_list[:], global_while_list,
    called_by_fun, fun_global, augAssign_str, PC[:])
617             memo = {}
618             ls = duplicateRemoval(tmp)
619             PC = copy._deepcopy_list(ls, memo)
620 
621         if parse_keyword(i, data) == "Assign":
622             global ww
623             ww += 1
624             i += 6
625             tmp = parse_parenthesis(i, data)
626             assign_str = tmp[0]
627             i = tmp[1]
628             if "value=Name(id='threading'" in assign_str:
629                 continue
630             tmp = assign_denning(G,assign_str, PC[:])
631             memo = {}
632             ls = duplicateRemoval(tmp)
633             PC = copy._deepcopy_list(ls,memo)
634         elif parse_keyword(i, data) == "If":
635             debugPrint("If stmt:")
```

```
636             global ww1
637             ww1 += 1
638             i += 2
639             tmp = parse_if(i, data)
640             if_str = tmp[0]
641             i = tmp[1]
642             rest = tmp[2]
643             tmp = if_denning(G,called_by_fun, fun_global, if_str, rest, PC
    [:])
644             ls = duplicateRemoval(tmp)
645             memo = {}
646             PC = copy._deepcopy_list(ls, memo)
647         elif parse_keyword(i, data) == "While":
648             debugPrint("while stmt:")
649             global ww2
650             ww2 += 1
651             i += 5
652             tmp = parse_parenthesis(i, data)
653             while_str = tmp[0]
654             i = tmp[1]
655             it = 1
656             while( it <= iteration ):
657                 memo = {}
658                 lastPC = copy._deepcopy_list(PC, memo)
659                 tmp = duplicateRemoval(PC)
660                 memo = {}
661                 PC = copy._deepcopy_list(tmp, memo)
662                 #print "### While Iteration:",it
663                 tmp = while_denning(it, G, called_by_fun, fun_global,
    while_str, PC[:])
664                 ls = duplicateRemoval(tmp)
665                 memo = {}
666                 PC = copy._deepcopy_list(ls, memo)
667                 #print lastPC,"|--|", PC
668                 #print "-> PC:",PC
669                 if lastPC == PC:
670                     #print "Saturation point of loop!"
671                     break
672                 it += 1
673         i += 1
674     debugPrint(recCount)
675     recCount -= 1;
```

```
676     debugPrint("END continuous parse:")
677     return PC[:]
678 
679 def process_label_fille(filename):
680     # a = x,y,z
681     label = {}
682     with open(filename, "r") as inputfile:
683         for line in inputfile:
684             ll = "".join(line.split())
685             tmp = ll.split("=")
686             var = tmp[0]
687             lineList = tmp[1].split(",")
688             label[var] = lineList
689     return label
690 
691 ################################## main
    ##############################################
692 with open(sys.argv[1], "r") as inputfile:
693     # data = inputfile.read().replace('\n', '').replace(' ','')
694     data = "".join(inputfile.read().split())
695 llist = []
696 dummy = []
697 su_sr_list = []
698 line = 0
699 PC = [[]]
700 label = {}
701 label = process_label_fille(sys.argv[2])
702 gg = []
703 output = []
704 for key in label:
705     gg.append(key)
706 G = set(gg)
707 fun_call_index = [m.start() for m in re.finditer("FunctionDef", data)]
708 for index in fun_call_index:
709     varName = extract_variavle_name(index+len("FunctionDef(name='"),data)
710     fun_hash[varName] = index
711 
712 #G = Global(data)
713 continuous_parse(G,"", [], dummy, data, PC)
714 
715 for it in list(unique_everseen(output)):
716     print it
```

Appendix E

Python Script 2: Constraint Checker

```
"""It takes two files as input First: constraint file Second: label file
    """
"""Format of constraint file :  no of constraint
                                x + y <= z
                                a <= b
                                ...
                                """
"""Format of label file :  u = [['s1'],[x,y,...],[a,b,...]]   single qute '
    are optional
                           v = [['s2'],[x,y,...],[a,b,...]]
    """

import sys

if len(sys.argv) != 3:
    print "Error: wrong parameter (constraint file labelfile)"
    exit(0)

def process_constraint_file():
    cons = []
    with open(sys.argv[1], "r") as inputfile:
        i = 1
        constraints = -1
        for line in inputfile:
            line = "".join(line.split())
            if line == "":
                break
            if i == 1:
```

```
27                  constraints = int(line)
28                  i+=1
29                  continue
30              tmp = line.split("<=")
31              left = tmp[0]
32              right = tmp[1]
33              left_list = left.split("+")
34              cons.append([left_list,right])
35      return cons
36
37  def process_label_file():
38      labels = {}
39      with open(sys.argv[2], "r") as inputfile:
40          for line in inputfile:
41              line = "".join(line.split())
42              line = "".join(line.split("'"))
43              if line == "":
44                  break
45              tmp = line.split("=")
46              left = tmp[0]
47              right = tmp[1]
48              tmp_s = right.strip("[").strip("]").split("],[")
49              owner = tmp_s[0]
50              first_list = tmp_s[1].split(",")
51              second_list = tmp_s[2].split(",")
52              labels[left] = [set(first_list),set(second_list),owner]
53      return labels
54
55
56  def join(label1,label2):
57      R = label1[0].intersection(label2[0])
58      W = label1[1] | label2[0]
59      return [R,W]
60
61  def can_flow(label1,label2):
62      return label1[0].issuperset(label2[0]) and label1[1].issubset(label2
        [1])
63
64  def sat(cons,labels):
65      for constraint in cons:
66          tmp = set(constraint[0])
67          left_list = list(tmp)
```

```
68          right = constraint[1]
69          for it in left_list:
70              if not can_flow(labels[it],labels[right]):
71                  return False
72      return True
73
74  def can_perform(subject,constraint,labels):
75      if subject not in labels[constraint[1]][1]:
76          return False
77      tmp = set(constraint[0])
78      left_list = list(tmp)
79      for it in left_list:
80          if subject not in labels[it][0]:
81              return False
82      return True
83
84  def can_perform_set(cons,labels):
85      # type: (list, dict{set,set,"str"}) -> set
86      s = labels[cons[0][1]][1]
87      for constraint in cons:
88          tmp = set(constraint[0])
89          left_list = list(tmp)
90          right = constraint[1]
91          for it in left_list:
92              s = s.intersection(labels[it][0])
93              if len(s) == 0:
94                  return s
95          s = s.intersection(labels[right][1])
96      return s
97
98  def can_perform_all(subject,cons,labels):
99      for constraint in cons:
100         if can_perform(subject,constraint,labels) == False:
101             return False
102     return True
103
104 ########################## main ######################
105 print """Enter Choice
106          1.Satisfied or not
107          2.Can given subject perform all constraints
108          3.Checking existance of subjects who follows constraints"""
109 choice = raw_input()
```

```
110
111 cons = process_constraint_file()
112 labels = process_label_file()
113
114 if choice == '1':
115     print sat(cons,labels)
116 elif choice == '2':
117     print "Enter subject"
118     subject = raw_input()
119     print can_perform_all(subject, cons, labels)
120 elif choice == '3':
121     print can_perform_set(cons,labels)
```

Appendix F

Copy Programs

```
1  ################################################
2  #Procedure copy1
3  #z=0
4  #y=0
5  if x == 0:
6    z=1
7  if z==0:
8    y=1
9
10 ################################################
11 # Procedure copy2
12 z = 1
13 y = -1
14 while z == 1:
15   y = y + 1
16   if y == 0:
17     z = x
18   else:
19     z = 0
20
21 ################################################
22 #copy3 synchronization flow
23 import thread
24 import time
25 import threading
26
27 #x=7
28 #y=6
```

```
#def copy3(x,y):      # copy x to y
s0 = threading.Event()
s1 = threading.Event()

def thread1():
    global x
    if x==0:
        s0.set()
    else:
        s1.set()

def thread2():
    global y
    s0.wait()
    s0.clear()
    y=1
    s1.set()

def thread3():
    global y
    s1.wait()
    s1.clear()
    y=0
    s0.set()

try:
    thread.start_new_thread(thread1,())
    thread.start_new_thread(thread2,())
    thread.start_new_thread(thread3,())
except:
    print "Error: unable to start thread"

##############################################
#copy4 Global flow in concurrent programs
import thread
import time
import threading


def thread1():
    global x
    global e0
```

```
71      global e1
72      if x==0:
73          e0 = False
74      else:
75          e1 = False
76
77  def thread2():
78      global e0
79      global e1
80      global y
81      while e0:
82          pass
83
84      y = 1
85      e1 = False
86
87  def thread3():
88      global e1
89      global e0
90      global y
91      while e1:
92          pass
93      y = 0
94      e0 = False
95
96  try:
97      thread.start_new_thread(thread1,())
98      thread.start_new_thread(thread2,())
99      thread.start_new_thread(thread3,())
100 except:
101     print "Error: unable to start thread"
102
103
104 #copy5
105 y = 0
106 while x==0 :
107   pass
108 y = 1
109
110 ################################################
111 #copy 6
112 z = 0
```

```
113 sum = 0
114 y = 0
115 while z == 0 :
116   sum = sum + x
117         y = y + 1
118
119 ################################################
120 #dynamic label
121 def fun(x, y, z):
122     a = x
123     y = a
124     a = z
125 fun(x, y, z)
```

References

[1] Dorothy Elizabeth Robling Denning. *Cryptography And Data Security*. Addison-Wesley, Reading, Massachusetts, California, London, Amsterdam, Don Mills, Ontario, Sydney, 1982.

[2] D Elliott Bell and Leonard J LaPadula. Secure computer systems: Mathematical foundations. Technical report, DTIC Document, 1973.

[3] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[4] NV Narendra Kumar and RK Shyamasundar. Realizing purpose-based privacy policies succinctly via information-flow labels. In *Big Data and Cloud Computing (BdCloud), 2014 IEEE Fourth International Conference on*, pages 753–760. IEEE, 2014.

[5] Fred B Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In *Informatics*, pages 86–101. Springer, 2001.

[6] Security policies and security models. Technical report.

[7] Daryl McCullough. Noninterference and the composability of security properties. In *Security and Privacy, 1988. Proceedings., 1988 IEEE Symposium on*, pages 177–186. IEEE, 1988.

[8] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2-3):167–187, 1996.

[9] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[10] Zhifei Chen, Lin Chen, and Baowen Xu. Hybrid information flow analysis for python bytecode. In *Web Information System and Application Conference (WISA), 2014 11th*, pages 95–100. IEEE, 2014.

[11] Juan José Conti and Alejandro Russo. A taint mode for python via a library. In *Nordic Conference on Secure IT Systems*, pages 210–222. Springer, 2010.