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 information flow analysis of python source code. Our platform takes source code and labels of all objects used in source code 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 make a secure system, who is allowed to execute particular code should be considered too, this could be any process or user, we termed it subject. Use of Reader Writer Flow model [1] with subjects makes this information flow analysis more secure and easy to process. We developed a script to track all possible information flow from a given source code of python and it generates constraints for secure flow of information. The second script 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.

Contents

List of Figures

Listings

Chapter 1

Introduction

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 information 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**

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

   Listing 1.1: 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**

   ```
   if y == 1:
     x = 1

   ```

   Listing 1.2: Python example

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

Listing 1.3: 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 1.3 . 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].

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

Listing 1.4: Python example

Here assignment operation on x in Listing 1.4 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 1.5 needs to follow Denning's lattice and constraints written in figure 1.1, security class of variable var is denoted by $\underline{var}$.

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

Listing 1.5: Python example

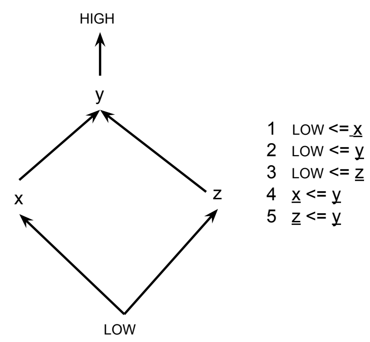

Figure 1.1: Lattice of Listing 1.5

Chapter 3 describes information flow by nonterminating while loop and how to certify program in such situation. Chapter 4 describes how information flows among thread in the multi-threaded program using semaphore. Chapter 5 presents a new approach to certifying a program using more sophisticated labels (RWFM label) and it also considers subjects for certification of the 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.

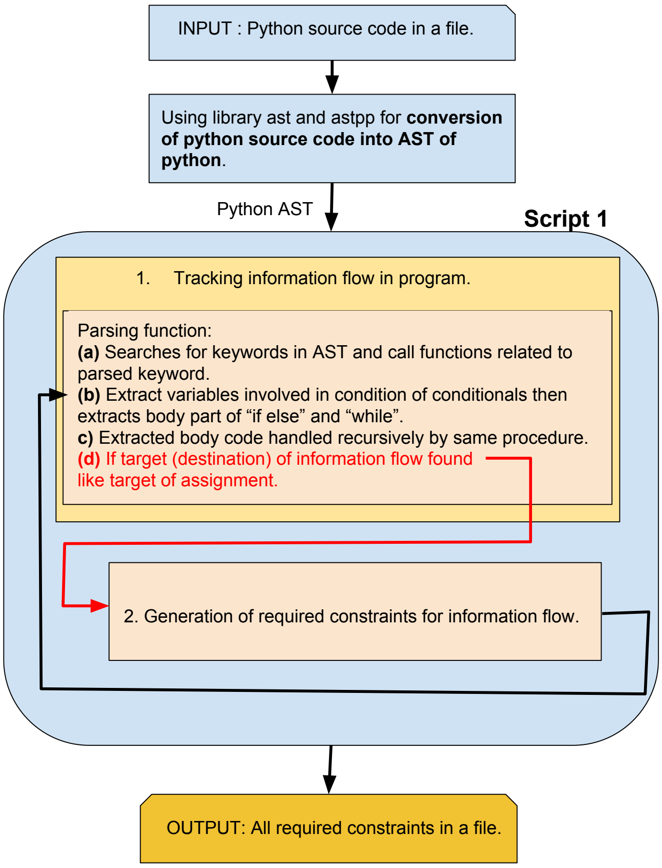

Figure 1.2: Block diagram of script1

Contributions

Subject considered with objects for certification of python program. Reader Writer Flow Model [2] 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)

Subset of features of python language considered for analysis.

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

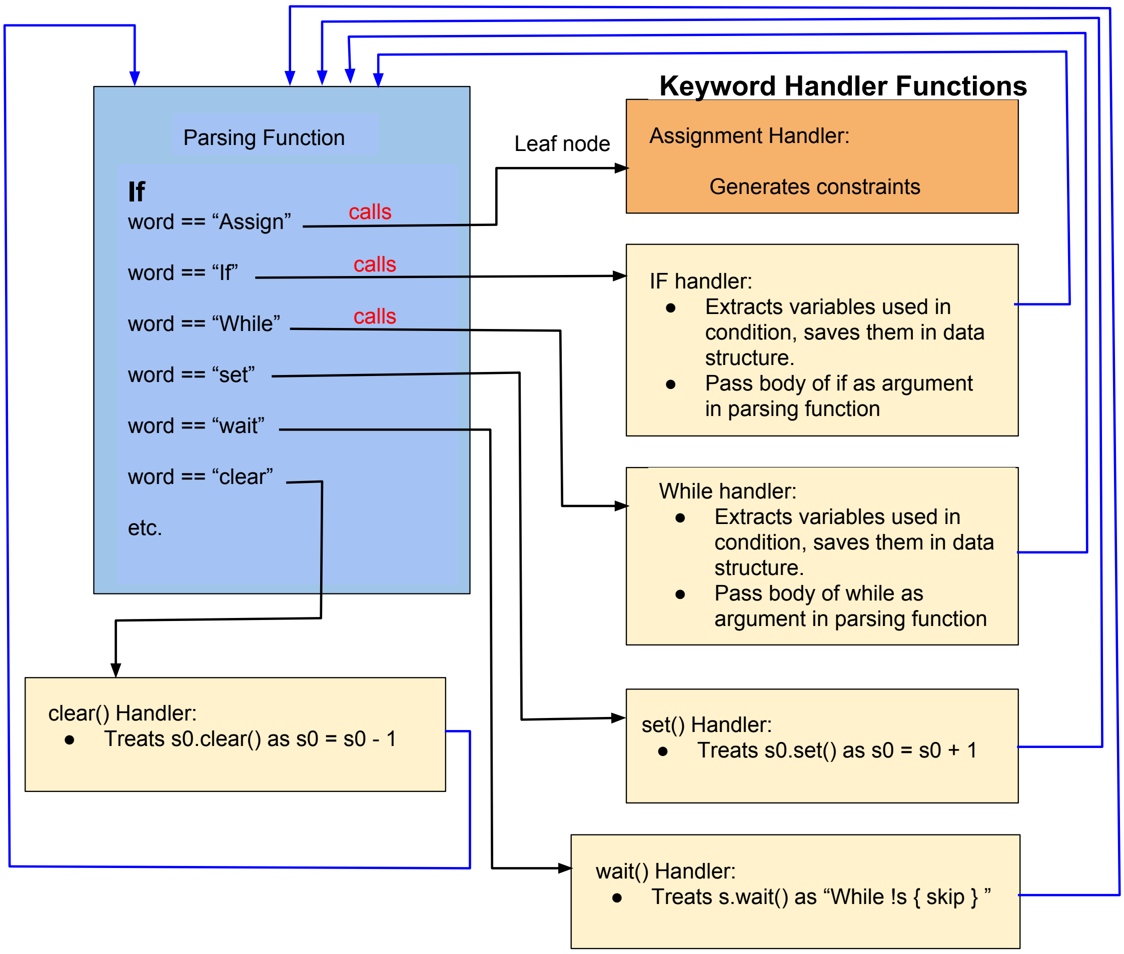

Figure 1.3: Block diagram of parsing function

I implemented fully automated certification platform for python source code using two python scripts, given in Appendix A and B. Block diagram in figure 1.2 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 1.3 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 5.1 of chapter 5.

Chapter 2

Related work

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 [3], 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 → denotes information flow).

- x = y : y → x
- if e then x = y : e → x
- while w if e then x = y w =false : w ⊕ e → x
- infinite loop: while w ; x = y :- w → x etc.

Chen et al. [4](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. [2] introduce a new model to work with subjects and my work will be focused on this. Conti et al. [5] provide library support in python for information flow analysis in explicit flows only.

Chapter 3

Information Flow Analysis using Denning's Lattice Model

Analysis In Local Scope

We started certification of python code at local scope initially. In figure 3.1 we considered

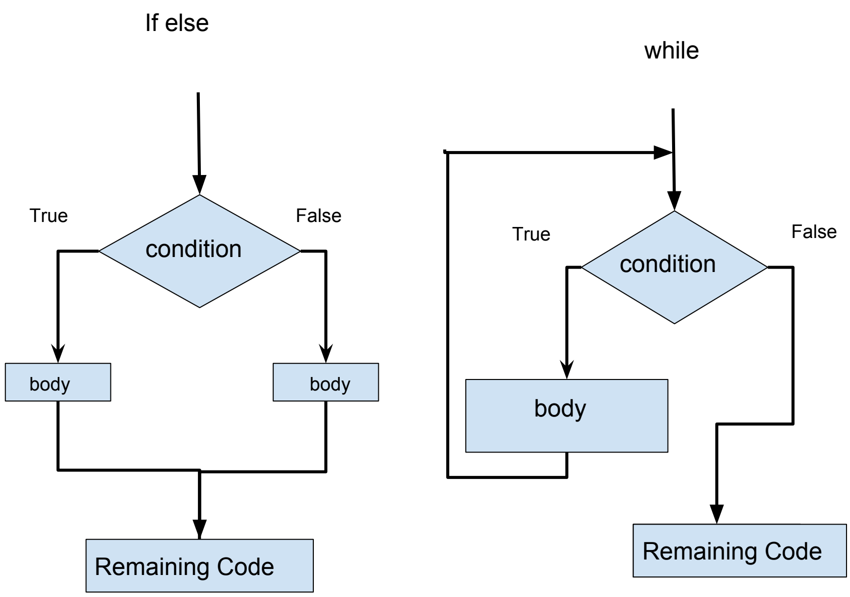

Figure 3.1: Control flow graph of If else and while loop

information flowing from variables used in condition to targets of assignment operation in body code only. Further analysis will be dealing with information flows to targets of assignment in

remaining code. Now a program with if else and terminating loops can be certified to be secure (no information leaks) by using basic rules of information theory implemented in my script.
Fifth chapter of Denning's book[1] used six examples to describe how can information flow through different ways in the program. We used python version of these examples as benchmark.

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

```
#Procedure copy1
x = 0
z = 0
y = 0
if x == 0:
  z=1
if z == 0:
  y=1
```

Listing 3.1: Python version of copy1 example in [1]. goal: information flow from x to y

Constraints for program in 3.1 generated by our script are:

1. low $\leqslant \underline{x}$
2. low $\leqslant \underline{z}$
3. low $\leqslant \underline{y}$
4. low $\leqslant \underline{y}$
5. $\underline{x} \leqslant \underline{z}$
6. $\underline{z} \leqslant \underline{y}$

Program in listing 3.1 shows how information can flow indirectly from x to y, for secure information flow $\underline{x} \leqslant \underline{y}$ must be true. Constraint 5 (x $\leqslant$ z) and constraint 6 (z $\leqslant$ y) with transitivity property of lattice model satisfies x $\leqslant$ y.

```
# Procedure copy2
x = 0
z = 1
y = -1
while z == 1:
  y = y + 1
  if y == 0:
    z = x
  else:
```

```
    z = 0
```

Listing 3.2: Python version of copy2 example in [1]. goal: information flow from x to y

Constraints of program in Listing 3.2 generated by our script are:

1. low $\leqslant \underline{x}$
2. low $\leqslant \underline{z}$
3. low $\leqslant \underline{y}$
4. $\underline{y} \oplus \underline{z} \leqslant \underline{y}$
5. $\underline{y} \oplus \underline{x} \oplus \underline{z} \leqslant \underline{z}$

Information flows from x to y using iteration of the while loop in listing 3.2. One iteration of while loop changes the value of y to -1 to 0 by incrementing it by 1, and two iteration of same while loop changes the value of y to -1 to 1. Number of iterations of while loop depends on value of x so this program is transmitting information from x to y indirectly using both explicit and implicit information flows.

Proof:

$\underline{x} \leqslant \underline{y}$ must be true for secure information flow. This can be proved using constraint 4 ($\underline{y} \oplus \underline{z} \leqslant \underline{y}$) and constraint 5 ($\underline{y} \oplus \underline{x} \oplus \underline{z} \leqslant \underline{z}$) in two ways. First: $\underline{z} \leqslant \underline{y}$ is given in constraint 4 and $\underline{y} \leqslant \underline{z}$ is given in constraint 5 so $\underline{y}$ must be equal to $\underline{z}$ using this new constraint ($\underline{y} \equiv \underline{z}$) and reduced constraint 5 ($\underline{x} \leqslant \underline{z}$) desired constraint $\underline{x} \leqslant \underline{y}$ is proved.

Further analysis: considering global influence of while

A program either terminates after a finite period of time or executes for an infinite period of time. The latter can be achieved through while loop and for loop. Such a non-terminating loop influences variables within the body of the loop as well as after the body of loop.

```
 x = 0
 z = True
 y = 0
 while z :
   y = 5
x = 5

```

Listing 3.3: Non terminating while.

In listing 3.3 execution of last statement x = 5 is conditioned on "while z" statement similar to "if" branching but "if" has limited body but in this case every statement that comes after "while z" share fate with x = 5, because of this property of "while" it is very crucial to know whether loop is terminating or nonterminating, so either there should be a mechanism to determine to terminating and nonterminating nature of loop or treat all while loop as nonterminating, latter approach is imprecise but secure, for now we are using this approach to handle while loops.

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

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

Listing 3.4: Python version of copy5 example in [1]. goal: information flow from x to y

Constraints generated by our script:

1. low $\leqslant \underline{y}$
2. $\underline{x} \leqslant \underline{y}$

so here our script able to track information flow from x to y and generating constraint $\underline{x} \leqslant \underline{y}$ for verification.

```
#Procedure copy4
import thread
import time
import threading

def thread1():
global x
global e0
global e1
if x==0:
   e0 = False
else:
   e1 = False

def thread2():
global e0
global e1
global y
while e0
   pass
y = 1
e1 = False

```

```
def thread3():
global e1
global e0
global y
while e1:
  pass
y = 0
e0 = False

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

Listing 3.5: Python version of copy4 example in [1]. goal: information flow from x to y

Constraints generated by our script for program in listing 3.5 are:

1. $\underline{x} \leqslant \underline{e0}$
2. $\underline{x} \leqslant \underline{e1}$
3. $\underline{e0} \leqslant \underline{y}$
4. $\underline{e0} \leqslant \underline{e1}$
5. $\underline{e1} \leqslant \underline{y}$
6. $\underline{e1} \leqslant \underline{e0}$

$(\underline{x} \leqslant \underline{e0})$ and $(\underline{e0} \leqslant \underline{y}) \quad \equiv \quad \underline{x} \leqslant \underline{y}$ (transitivity)

$(\underline{x} \leqslant \underline{e1})$ and $(\underline{e1} \leqslant \underline{y}) \quad \equiv \quad \underline{x} \leqslant \underline{y}$

So script able to track hidden information flow from x to y and generating constraint for this flow $\underline{x} \leqslant \underline{y}$ by using transitivity.

Chapter 4

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().

```
s = threading.Event()
s.wait()
x = 1
```

Listing 4.1: Example of wait() operation on binary semaphore.Info Flow: s→x

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

Listing 4.2: Infinite while loop, Information Flow: x→y

Listing 4.1 and Listing 4.2 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 4.1 and 4.2 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 4.3: Python version of copy3 example in [1]. goal: information flow from x to y

To certify the multi-threaded program in Listing 4.3 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 4.3 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 5

Constraint Verification with RWFM Labels

Information flow policies require some data related to objects and subjects to enforce any security protocol on them. Each object and subject needs to specify information related to permission like who can read, who can write etc. Label or security class is a way to represent this information. Information flow policies add some prerequisite condition with each operation on objects based on labels of objects and subject.

RWFM label [2]

Reader Writer Flow Model label has three tuple (s,R,W) the first s is a subject, R is a set of subjects allowed to read object, W is a set of subjects allowed to write and subjects who influenced this object so far.

Secure Information Flow

All information flow security models follow property of lattice because, for secure flow, information must flow from less secure class or label to more secure class or label, any reverse flow is a violation of security. So each information flow model redefines $\leqslant$ operator to check validity of flow.

Label1 = (s_1, R_1, W_2), **Label2** = (s_2, R_2, W_2)

information can flow from Label1 to Label2 if and only if $R_1 \supseteq R_2$ and $W_1 \subseteq W_2$.

Definition of Join and Meet operations [2]

Join: Label1 $\oplus$ Label2 = $(s_3, R_1 \cap R_2, W_1 \cup W_2)$

Meet: Label1 $\otimes$ Label2 = $(s_3, R_1 \cup R_2, W_1 \cap W_2)$

Create, Read and Write operations on objects with RWFM label [2]

If a subject $s^{(s,R_1,W_1)}$ creates object o, o will be assigned a default label derived from subject is $(s, R_1, W_1 \cup \{s\})$. Clearance of subject s is assumed (s,R_s,W_s), clearance is used to set an upper bound on labels of a subject.

Read: A subject $s^{(s_1,R_1,W_1)}$ can read object $o^{(s_2,R_2,W_2)}$ if and only if s $\in R_2$ and $(R_1 \cap R_2) \supseteq R_s \wedge (W_1 \cup W_2) \subseteq W_s$. after read operation label of subject s changes into $(s_1, R_1 \cap R_2, W_1 \cup W_2)$

Write: A subject $s^{(s_1,R_1,W_1)}$ can read object $o^{(s_2,R_2,W_2)}$ if and only if s $\in W_2$ and $R_1 \supseteq R_2 \wedge W_1 \subseteq$ W2.

Constraint Verification

First of all constraint checker script processes inputs and populate data structures. Inputs are two files, one has all constraints generated by our constraint generator script, other has labels of each object provided by user. Figure 5.1 shows modules of script2 with flow of control. After preprocessing of given constraints and labels, script provides answer to following queries.

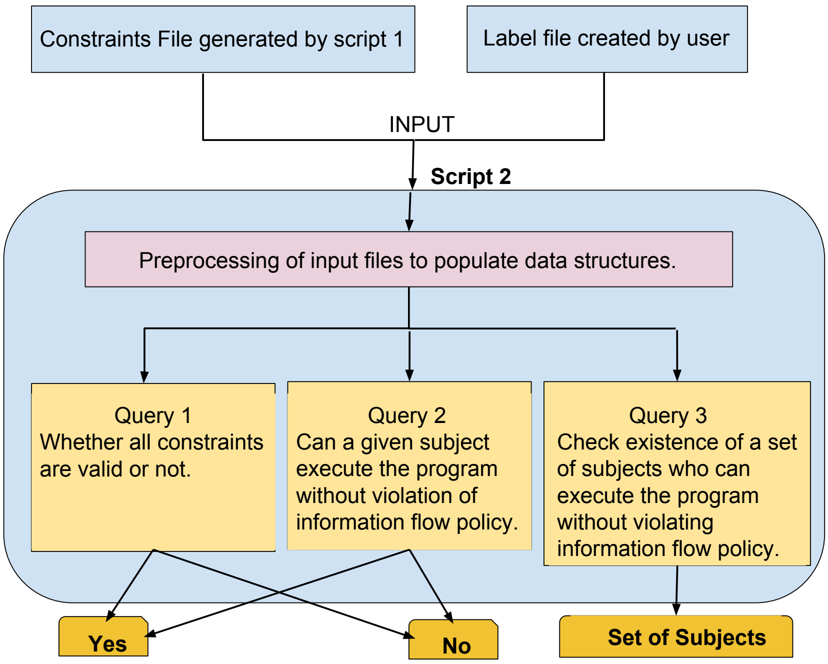

Figure 5.1: Block diagram of script2

1. **Whether all constraints are valid or not.**

   Constraint format in input file : $\underline{x} \leq \underline{y}$.

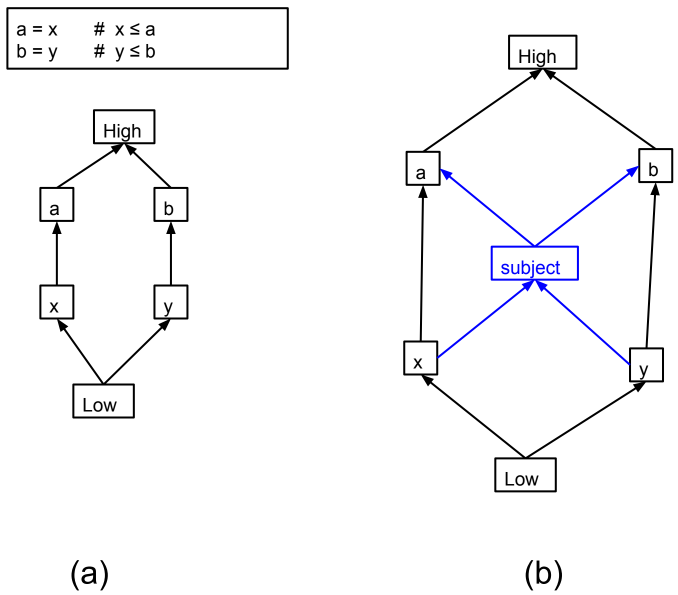

Figure 5.2: Lattice for information flow in a =x and y =b,(a)without subject (b) with subject.

Format of RWFM labels : label(x) (s_1, R_1, W_1)
label(y) (s_2, R_2, W_2)

Definition of $\leqslant_{RWFM}$ operator in RWFM model : label(x) $\leqslant_{RWFM}$ label(y) if only if $R_1 \supseteq R_2$ and $W_1 \subseteq W_2$ [2]

Script reads all constraints from input file one by one then converts all objects involved in constraint into RWFM label and checks whether they follow $\leqslant_{RWFM}$, if any constraint fails to follow $\leqslant_{RWFM}$ then returns false otherwise true.

2. **Can a given subject execute the program without violation of information flow policy.**

   In the previous case, we are checking constraints without considering the subject, but now a subject is given as input and we have to check whether given subject has all required permissions to execute all statements without violation of information flow policy. Figure 5.2 shows that subject needs to read x and write a in order to execute a = x, so it must follow label(x) $\leqslant_{RWFM}$ label(subject) and label(subject) $\leqslant_{RWFM}$ label(a) to maintain secure information flow. If the subject does not have required permissions for any statement then script returns false otherwise true.

3. **Check existence of a set of subjects who can execute the program without violating**

information flow policy.

In this case, we need to find a maximal set of those subjects who can pass 2nd case. A naive way to implement it: for each subject run the 2nd test and if it passes, add it in a set; after checking all subjects this set will be desired output. There is an efficient way to do it, any subject can pass 2nd test if and only if it is present in reader set of all objects which lie on the left side of $\leqslant$ operator in constraints and it is present in writer set of all objects which lie on the right side of $\leqslant$ operator in constraints . So we take the stepwise intersection of all reader sets of objects tht lie in left side of $\leqslant$ operator in constraints and writer sets of all objects that lie on the right side of $\leqslant$ operator in constraints.

Example: constraint is given in equation 5.1

Conversion of security class of objects into RWFM labels: $x^{(s,R_1,W_1)}$, $y^{(s,R_2,W_2)}$, $z^{(s,R_3,W_3)}$.

Output of script: $R_1 \cap R_2 \cap W_3$

$$\underline{x} \oplus \underline{y} \leqslant \underline{z} \tag{5.1}$$

Chapter 6

Conclusion & Future work

We have implemented python scripts to track and certify(for secure flow) information flow in a given python source code with new concept RWFM labels[2]. We considered the subject in information flow analysis, it helps to deal with real world problems related with information flow.

Future work

We will implement the following for completion of information flow analysis in python.

- Use of dynamic labels.
- Handling of control flows of functions.
- Implementation of steps to determine terminating condition of loops, to handle terminating and non-terminating loops differently.
- Tracking of information flow through covert channels.
- Information flow analysis on python data structures list, dictionary etc.
- Information flow analysis related to dynamic types and objects in python.

Appendices

Appendix A

Python Script 1: Constraint Generator

```
import sys
import re
import shutil

line = 0

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

def make_lub_string(llist):  # assumption list containing string elemnts
    if len(llist) == 0:
        return 0
    if len(llist) == 1:
        return llist[0]
    tmp = set(llist)
    uniq_list = list(tmp)
    if len(uniq_list) == 1:
        return str(uniq_list[0])
    ret = ""
    ret += uniq_list[0]
    i = 1
    while i < len(uniq_list):
        ret += " " + const.oplus + " "
        ret += uniq_list[i]
        i = i + 1
    return ret
```

```


def make_glb_string(llist):  # assumption list containing string elemnts
    # type: (list) -> string
    if len(llist) == 0:
        return 0
    if len(llist) == 1:
        return llist[0]
    tmp = set(llist)
    uniq_list = list(tmp)
    if len(uniq_list) == 1:
        return uniq_list[0]
    ret = ""
    ret += uniq_list[0]
    i = 1
    while i < len(uniq_list):
        ret += " " + const.oplus + " "
        ret += uniq_list[i]
        i = i + 1
    return ret


def split_through_orelse(if_str):
    # find first body word
    i = if_str.find("body=[")
    i = parse_square_br(i + 5, if_str)[1] + 1
    return ["{" + if_str[1:i] + "}", if_str[i + 7:]]

def parse_keyword(i, data):
    if i + 4 < len(data) and data[i:i + 5] == 'Expr(':
        return "Expr("
    if i + 9 < len(data) - 1 and data[i:i + 9] == 'AugAssign':
        return "AugAssign"
    if i + 6 < len(data) - 1 and data[i:i + 6] == 'Assign':
        return "Assign"
    if i + 2 < len(data) - 1 and data[i:i + 2] == 'If':
        return "If"
    if i + 5 < len(data) - 1 and data[i:i + 5] == 'While':
        return "While"
    if i + 11 < len(data) - 1 and data[i:i + 11] == 'FunctionDef':
        return "FunctionDef"
    return "none"
```

```


def parse_square_br(i, data):
    if data[i] != '[':
        print "Error: [ is missing"
        return []
    ret = "["
    count = 1
    i += 1
    while count > 0 and i < len(data) - 1:
        if data[i] == '[':
            count += 1
        if data[i] == ']':
            count -= 1
        ret += data[i]
        i += 1
    return [ret, i]


def parse_parenthesis(i, data):
    # type: (int , string) -> string
    if data[i] != '(':
        print "Error: ( is missing"
        return []
    ret = "("
    count = 1
    i += 1
    while count > 0 and i < len(data) - 1:
        if data[i] == '(':
            count += 1
        if data[i] == ')':
            count -= 1
        ret += data[i]
        i += 1
    return [ret, i]


def extract_variavle_name(startpos, line):
    # string->string
    var = ""
    while line[startpos] != "'":
        var += line[startpos]
```

```
        startpos = startpos + 1
    return var


def target_of_assignment(str):  # find all targets
    # string->list
    targets_ptrn = r"targets=\[.*?\]"
    ctargets_ptrn = re.compile(targets_ptrn)
    temp_list = ctargets_ptrn.findall(str)
    ret = ''.join(temp_list)  # converting to string
    return ret


def parse_variables(line):
    # type: (string) -> list
    id_index = [m.start() for m in re.finditer('id=', line)]
    var_list = []
    for it in id_index:
        vname = extract_variavle_name(it + 4, line)
        if vname == "False" or vname == "True":
            continue
        var_list.append(vname)
    return var_list


def multiple_assign(parent_list, global_while_list, assign_str,
   target_id_index):
    global line
    tmp = assign_str.split("value", 1)
    rvalue = parse_variables(tmp[0])
    lvalue = parse_variables(tmp[1])

    # printing denning's rule
    for it in rvalue:
        if len(lvalue) == 0:
            print "low " + const.lt + " " + it
            line += 1
        else:
            print make_lub_string(lvalue), const.lt, it
            line += 1


```

```
def assign_denning(parent_list, global_while_list, called_by_fun,
    fun_global, assign_str):  # applying dennig's model on assignments
    global line
    ss = assign_str.split("value")
    target_id_index = [m.start() for m in re.finditer('id=', ss[0])]

    if len(target_id_index) > 1:
        multiple_assign(parent_list, global_while_list, assign_str,
    target_id_index)
        return 0

    if "id='" in ss[0]:
        left = extract_variavle_name(0, ss[0].split("id='")[1])
        # print "lvalue", left," ",
    else:
        left = ['const']
    id_index = [m.start() for m in
                re.finditer('id=', ss[1])]  # list of starting index of
    variables in right part of string
    rvalue = []
    if len(id_index) == 0:
        # rvalue.append("low")
        pass
    else:
        for it in id_index:
            startpos = it + 4
            vname = extract_variavle_name(startpos, ss[1])
            """Exclusion of False keyword"""
            if vname == "False" or vname == "True":
                continue
            rvalue.append(vname)

    """Update local parent_list """
    parent_list += rvalue

    """Printing denning rules for stmts"""

    # print "= rvalues", rvalue, "parent_list", parent_list
    """Renaming all local variable of function"""
    mod_parent_list = []
    if called_by_fun != "":
        if left not in fun_global:
```

```
            left += "_" + called_by_fun
        for it in parent_list:
            if it not in fun_global:
                it += "_" + called_by_fun
            mod_parent_list.append(it)
    if called_by_fun != "" and len(mod_parent_list) > 0:
        parent_list = mod_parent_list

    ret = ""
    if len(parent_list + global_while_list) == 0 :
        ret = "low" + " " + const.lt + " " + left
    else:
        ret = make_lub_string(parent_list + global_while_list)
        ret += " " + const.lt + " " + left
    print ret
    line += 1

def augAssign_denning(parent_list, global_while_list, called_by_fun, fun_global, augAssign_str):
    i = augAssign_str.find("id=")
    var_name = extract_variavle_name(i + 4, augAssign_str)
    parent_list.append(var_name)
    assign_denning(parent_list[:], global_while_list, called_by_fun, fun_global, augAssign_str)


def track_while(while_str):
    while_list = []
    while_index = [m.start() for m in re.finditer("While", while_str)]
    for it in while_index:
        count = 0
        i = it
        while count < 2 :
            if while_str[i] == "(":
                count += 1
            i += 1
        i -= 1
        test_str = parse_parenthesis(i, while_str)[0]
        while_list += parse_variables(test_str)
    return while_list

def if_denning(parent_list, global_while_list, called_by_fun, fun_global,
```

```
if_str):
    # type: (list, list, string, dict, string) -> print rules
    if "orelse=" not in if_str:
        # print "termination",if_str
        if if_str[0:2] == "[]":  # absence of else part
            return []
        else:  # handling else part
            else_str = if_str
            continuous_parse(parent_list[:], global_while_list,
called_by_fun, fun_global, else_str)
            return []

    tmp = split_through_orelse(if_str)
    if_half = tmp[0]
    ladder = tmp[1]

    if if_str[1:5] != "test":
        print "Error test not found in if"

    """extract test=...() from if_half"""
    i = if_half.find("(")
    tmp = parse_parenthesis(i, if_half)
    test_str = tmp[0]
    parent_list += parse_variables(test_str)
    i = tmp[1]

    """then extract body part and process like normal AST text """
    # body processing
    body_onward_str = if_half[i:]  ### Asumption : Compare string always
followed by body=[...] imediatly
    # setting i to location of [ in body_str: ,body=[..
    i = body_onward_str.find("[")
    body_str = parse_square_br(i, body_onward_str)[0]

    continuous_parse(parent_list[:], global_while_list, called_by_fun,
fun_global, body_str)
    continuous_parse(parent_list[:], global_while_list, called_by_fun,
fun_global, ladder)

def while_denning(parent_list, global_while_list, called_by_fun, fun_global
, while_str):
    compare = "()"
```

```
    if while_str[6:10] == "Name":
        tmp = parse_parenthesis(10, while_str)
        compare = tmp[0]
        i = tmp[1]
    elif while_str[6:10] == "Comp":
        tmp = parse_parenthesis(13, while_str)
        compare = tmp[0]
        i = tmp[1]

    parent_list += parse_variables(compare)
    global_while_list += parse_variables(compare)

    # body processing
    body_onward_str = while_str[i:]  ### Asumption : Compare string always
   followed by body=[...] imediatly
    # setting i to location of [ in body_str: ,body=[..
    i = body_onward_str.find("[")
    body_str = parse_square_br(i, body_onward_str)[0]

    continuous_parse(parent_list[:], global_while_list, called_by_fun,
   fun_global, body_str)

def set_clear_denning(parent_list, global_while_list,called_by_fun,
   fun_global, expr_str):
    global line
    i = expr_str.find("Call(")
    i += 4
    call_str = parse_parenthesis(i, expr_str)[0]
    i = call_str.find("Attribute(")
    i += 9
    attribute_str = parse_parenthesis(i, call_str)[0]
    i = attribute_str.find("Name(")
    i += 4
    # name_str = parse_parenthesis(i,attribute_str)[0]
    i = attribute_str.find("id=")
    var_name = extract_variavle_name(i + 4, attribute_str)
    """Renaming local var"""
    if var_name not in fun_global:
        var_name += "_"+called_by_fun
    i = attribute_str.find("attr=")
    attr = extract_variavle_name(i + 6, attribute_str)
    if attr == "set":
```

```
            # treat it like AugAssign s0 += 1
            print make_lub_string(parent_list + [var_name] + global_while_list) + " <=  " + var_name
            line += 1
        elif attr == "clear":
            # treat it like AugAssign s0 -= 1
            print make_lub_string(parent_list + [var_name] + global_while_list) + " <= " + var_name
            line += 1
        elif attr == "wait":
            global_while_list.append(var_name)

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]
        ss = sq_str.strip("[").strip("]")
        sslist = ss.split(",")
        for it in sslist:
            if it == '':
                continue
            globals[(it.strip("'"))] = 1
    return globals


def fun_denning(global_while_list, fun_str):
    fun_name = extract_variavle_name(fun_str.find("name=") + 6, fun_str)
    fun_globals = extract_Globals(fun_str)
    fun_globals.update(supreme_global)
    fun_global_while = []
    continuous_parse(global_while_list, fun_global_while, fun_name, fun_globals, fun_str)


# global var for counting
ww = ww1 = ww2 = 1
nested_while = 0
supreme_global = {}

def continuous_parse(parent_list, global_while_list, called_by_fun,
```

```
fun_global, data):
    # type: (object, object) -> object
    length = len(data)
    i = 0
    while i < length - 1:
        # checking for keyword
        if parse_keyword(i, data) == "FunctionDef":
            i += 11
            tmp = parse_parenthesis(i, data)
            fun_str = tmp[0]
            i = tmp[1]
            global_while_list = []
            fun_denning(global_while_list,fun_str)
        if parse_keyword(i, data) == "Expr(":                    #NEED
generlization
            i += 4
            tmp = parse_parenthesis(i, data)
            expr_str = tmp[0]
            i = tmp[1]
            set_clear_denning(parent_list[:], global_while_list,
called_by_fun, fun_global, expr_str)
        if parse_keyword(i, data) == "AugAssign":
            i += 9
            tmp = parse_parenthesis(i, data)
            augAssign_str = tmp[0]
            i = tmp[1]
            augAssign_denning(parent_list[:], global_while_list,
called_by_fun, fun_global, augAssign_str)
        if parse_keyword(i, data) == "Assign":
            global ww
            # print "Assign found",ww
            ww += 1
            i += 6
            tmp = parse_parenthesis(i, data)
            assign_str = tmp[0]
            i = tmp[1]
            if "value=Name(id='threading'" in assign_str:
                var_name = parse_variables(target_of_assignment(assign_str)
)[0]
                supreme_global[var_name] = 1
                continue
            assign_denning(parent_list[:], global_while_list, called_by_fun
```

```
    , fun_global, assign_str)
        elif parse_keyword(i, data) == "If":
            global ww1
            # print "If found", ww1
            ww1 += 1
            i += 2
            tmp = parse_parenthesis(i, data)
            if_str = tmp[0]
            i = tmp[1]
            if_denning(parent_list[:], global_while_list, called_by_fun,
    fun_global, if_str)
        elif parse_keyword(i, data) == "While":
            global ww2
            global nested_while
            # print "while found", ww2
            ww2 += 1
            i += 5
            tmp = parse_parenthesis(i, data)
            while_str = tmp[0]
            i = tmp[1]
            if nested_while == 0:
                global_while_list += track_while(while_str)
            nested_while += 1
            while_denning(parent_list[:], global_while_list, called_by_fun,
     fun_global, while_str)
            nested_while -= 1
        i += 1


############# main ###################
with open(sys.argv[1], "r") as inputfile:
    # data = inputfile.read().replace('\n', '').replace(' ','')
    data = "".join(inputfile.read().split())
orig_stdout = sys.stdout
f = file('tmp_constraints.txt', 'w')
sys.stdout = f
llist = []
dummy = []
outside_while_list = []
continuous_parse(llist[:], outside_while_list, "", dummy, data)

sys.stdout = orig_stdout
```

```
f.close()

from_file = open('tmp_constraints.txt','r')
to_file = open('constraints.txt','w')
to_file.write(str(line)+"\n")
shutil.copyfileobj(from_file, to_file)

# print "#while: ",ww2-1
# print "#assign: ",ww-1
```

Appendix B

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:
```

```
                constraints = int(line)
                i+=1
                continue
            tmp = line.split("<=")
            left = tmp[0]
            right = tmp[1]
            left_list = left.split("+")
            cons.append([left_list,right])
    return cons

def process_label_file():
    labels = {}
    with open(sys.argv[2], "r") as inputfile:
        for line in inputfile:
            line = "".join(line.split())
            line = "".join(line.split("'"))
            if line == "":
                break
            tmp = line.split("=")
            left = tmp[0]
            right = tmp[1]
            tmp_s = right.strip("[").strip("]").split("],[")
            owner = tmp_s[0]
            first_list = tmp_s[1].split(",")
            second_list = tmp_s[2].split(",")
            labels[left] = [set(first_list),set(second_list),owner]
    return labels


def join(label1,label2):
    R = label1[0].intersection(label2[0])
    W = label1[1] | label2[0]
    return [R,W]

def can_flow(label1,label2):
    return label1[0].issuperset(label2[0]) and label1[1].issubset(label2[1])

def sat(cons,labels):
    for constraint in cons:
        tmp = set(constraint[0])
        left_list = list(tmp)
```

```
        right = constraint[1]
        for it in left_list:
            if not can_flow(labels[it],labels[right]):
                return False
    return True

def can_perform(subject,constraint,labels):
    if subject not in labels[constraint[1]][1]:
        return False
    tmp = set(constraint[0])
    left_list = list(tmp)
    for it in left_list:
        if subject not in labels[it][0]:
            return False
    return True

def can_perform_set(cons,labels):
    # type: (list, dict{set,set,"str"}) -> set
    s = labels[cons[0][1]][1]
    for constraint in cons:
        tmp = set(constraint[0])
        left_list = list(tmp)
        right = constraint[1]
        for it in left_list:
            s = s.intersection(labels[it][0])
            if len(s) == 0:
                return s
        s = s.intersection(labels[right][1])
    return s

def can_perform_all(subject,cons,labels):
    for constraint in cons:
        if can_perform(subject,constraint,labels) == False:
            return False
    return True

########################## main #####################
print """Enter Choice
        1.Satisfied or not
        2.Can given subject perform all constraints
        3.Checking existance of subjects who follows constraints"""
choice = raw_input()
```

```

cons = process_constraint_file()
labels = process_label_file()

if choice == '1':
    print sat(cons,labels)
elif choice == '2':
    print "Enter subject"
    subject = raw_input()
    print can_perform_all(subject, cons, labels)
elif choice == '3':
    print can_perform_set(cons,labels)
```

References

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

[2] 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.

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

[4] 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.

[5] 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.