Concurrent Interprocedural Dataflow Analysis

by

Di Zou

Thesis submitted to the

Faculty of Graduate and Postdoctoral Studies

In partial fulfilment of the requirements

For the Master in

Computer Science

School of Electrical Engineering and Computer Science

Faculty of Engineering

University of Ottawa

Abstract

Detecting bugs plays a significant role in software development. Bugs may lead to unexpected behaviors. An attacker can gain control over a system by exploiting its bugs. Usually, an attack can be triggered by user's input. Unchecked user input can cause serious problems in a program. In order to prevent this situation, user's input must be checked carefully before it can be used. To provide the information of where user's input can affect a program, the taint dataflow analysis is being considered.

In this thesis, we introduce a concurrent solution to perform static taint dataflow analysis. The goal is to find the statements of the program dependent on user input and inform the developers to validate those. We provides a method for the static concurrent taint dataflow analysis based on sequential static taint dataflow analysis.

Static dataflow analysis is time consuming. This research addresses the challenge of efficiently analyzing the dataflow. Our experimental shows that our concurrent taint dataflow analysis improves the speed of analyzing complex programs.

Acknowledgements

Foremost, I would like to give my sincerest gratitude and appreciation to my supervisor Dr. Gregor v. Bochmann and my co-supervisor Dr. Guy-Vincent Jourdan for the continuous support of my Masters study and research, for their patience, motivation, enthusiasm and kindness. I attribute the level of my Masters degree to their encouragement and effort. I could not finish my Master study and this thesis without them. One simply could not wish for better or friendlier supervisors.

I could not be more thankful to my dear parents, Yongtong Zou and Zhuqing Cong for all their love and support. Without them, studying abroad in a postgraduate level would be impossible for me.

I would also like to show my greatest appreciation to my colleague Xinghao Xu and the members of the Software Security Research Group at University of Ottawa, as well as Dr. Vio Onut (IBM) for all their helpful suggestions and discussions.I owe a very important debt to Seyed M. Mirtaheri for helping me not only doing the research but also adapting to the new life in a new place half the earth away from my home.

In addition, I thank to IBM for all their financial support of my graduating study.

Contents

List of Figures

Chapter 1

Introduction

In recent years, complexity and scale of software development has increased tremendously compared to the early days of computing. As software gets larger and more complex, ensuring its reliability and security only get more difficult [9,36]. A *vulnerable software* is software that does not protect resources available to it the way it should. In this context, a *security vulnerability* is a weakness that allows the user or attacker to intentionally or unintentionally bypass granted privileges.

A common example of a security vulnerability is *buffer overflow*. A software that is vulnerable to buffer overflow attacks can potentially allow an attacker to run arbitrary code on the machine that runs the vulnerable software [7, 15, 21]. In this attack, the attacker inserts and runs code through the input chain of software. Another well-known example of software vulnerability is called *SQL injection* [10,25,44]. Vulnerable software in this case, can possibly allow the attacker to run arbitrary SQL commands, and by-pass the intended access control.

Detecting software vulnerabilities such as buffer overflow and SQL injection manually is not a trivial task. There have been extensive studies in the literature of computer science dedicated to the detection of software vulnerabilities automatically [14, 20, 27].

Program analysis [23, 43] is powerful tool to detect software vulnerabilities. This field addresses automated process of analyzing the behavior of a program. This technology is widely used for validation, correctness, optimization and vulnerability detection of programs.

Program analysis has two aspects:

- *Dynamic analysis* [45] is the analysis that is done at runtime given a set of inputs.
- *Static analysis* [23], is the analysis that is done on the source code, without executing the application.

Dynamic analysis is able to find run-time information while it is often hard to cover all possible states of the program. On the other hand, static analysis is done without executing the code and can consider all possible states of the program. However, static analysis is not able to detect dynamic behaviours such as *pointer-function* and *pointers*. In practice, static analysis is more likely to be used to understand the behaviours of a program [19, 40, 41].

User-input dependencies analysis, also known as *Taint analysis*, is one of the static analysis approaches that can efficiently and effectively detect many security vulnerabilities [33]. Unfortunately, taint analysis is complex and time consuming for a large application. In this thesis, we address this problem and aim at improving the speed of taint analysis by running the algorithm on several cores on the given computer concurrently.

The rest of this chapter is organized as follow:

- In Section 1.1, I review the history of program analysis and the challenges faced.
- In Section 1.2, I briefly describe the motivations of the project from the security point of view.

- In Section 1.3, I summarize the contributions we made in this thesis.
- Finally, in Section 1.4, we provide the organization of the whole thesis.

1.1 Program Analysis

In order for the compiler to produce correct and efficient code, it had to optimize the program. To be able to optimize the program, compiler must understand the behaviours of the program. The field of program analysis, and more specifically static analysis, was thus born.

Program analysis studies automatic analysis of the behaviour of the program. The analysis is widely used for many purposes, such as security validation, compiler optimization and circuit design. The two most widely used method of program analysis are: *dynamic analysis* and *static analysis*.

Dynamic analysis is the understanding of the program's behaviour by executing it [5, 6]. This analysis relies on a set of inputs. The algorithm checks the output without knowing how it was produced. Thus, dynamic analysis is more often used for detecting bugs, and it often can not ensure that all bugs are found. The performance of dynamic analysis is largely contingent on the quality of the input set. In practice, it is very hard to generate a good input set, and different test cases may need different sets.

Unlike dynamic analysis, static analysis [2, 6, 8] is performed on the source code, without executing the program. This analysis is used to understand the control flow and the dataflow of given program. The static analysis is capable of considering all possible states of a program and, therefore, it is more help for developers to find potential bugs in their program. [1]

[1] Dynamic analysis is also required to find run-time bugs as the static analysis can not provide any run-time information

1.2 Motivation

This research project aims at performing taint dataflow analysis for large programs using concurrent threads - thus improving performance. As part of dataflow analysis, taint dataflow analysis tracks all possible points of a given program where the user's input can be used. User's input can cause unexpected behaviours of a program if it is not properly validated. The objective of taint dataflow analysis is to tell which variables in the program requires validation.

1.2.1 An example of buffer overflow

Listing 1.1: Sample Vulnerable C Code

```
#include<stdio.h>
int main(int argc,char **argv){
  char buf[10];
  strcpy(buf,argv[1]);
  printf("buf's 0x%8x\n",&buf);
  return 0;
}
```

As mentioned above, buffer overflow is a common vulnerability. In this section, we show an example of this vulnerability and demonstrate how the attack can be triggered by the user input. Listing 1.1, shows a sample *C* code with a buffer overflow attack vulnerability. The code copies the user input string to an array and prints it out. The array buf is assigned 10 elements initially. Before the *strcpy* function is called, memory space of 10 characters will be allocated to the buffer. If the function *strcpy* copies more than 10 characters to buf, then the buffer is overflowed.

Due to the lack of any sanity checks, if the user's input string has more than 10 elements, the stack will overflow. In such situation, some parts of memory will be overwritten by the user input. A malicious user often tries to take advantage of this scenario and overwrite the return address of the function. The malicious user, intentionally forces the program to return to a place where the intended executable code is stored. The malicious user can then control the program and potentially gain access the compromised computer. Fortunately, these kind of bugs can be detected based on the taint dataflow analysis. Taint dataflow analysis indicates whether a variable uses the user's input or not. Based on the information provided by taint dataflow, developers know which variables need to be checked before use.

1.2.2 Reducing the Cost

A major challenge in detecting such bugs is performance. To understand where the user's input can be used, knowing the dataflow information in a single procedure is not enough. Further, in *object-oriented* software dataflow analysis must be performed globally [11, 26, 28, 38]. Global dataflow analysis is also called *Interprocedural dataflow analysis*.

Interprocedural dataflow analysis is very time consuming due to its iterative nature which re-analyzes parts of the program many times before it finds all the traces of variables. Our goal is to address the challenge of analyzing complex programs by introducing *concurrent taint dataflow analysis*. Concurrent taint dataflow analysis parallelizes the task of analysis by breaking it down into independent tasks and performing the analysis concurrently with multiple threads.

1.3 Contributions

The main contributions of this thesis can be summarized as:

- A concurrent algorithm for performing static taint analysis. The concurrent algorithm is performed on multiple threads and each thread analyzes one call path. The result of analysis is merged at the end of the analysis.
- A methodology of sharing pointer-based global data between threads. The global data is shared with all threads which reduces the memory usage in the program.
- Implement concurrent static taint dataflow analysis based on a given sequential static taint dataflow analysis program. The given program is optimized to reduce memory usage. For instance, all objects used in the given program are not able to be duplicated. We implement a concurrent approach that duplicates smallest information to support multiple threading.
- Improvement of the speed of performing static taint analysis. The given program analysis the call paths one by one. Our approach analysis the call paths separately and simultaneously.

1.4 Thesis Organization

The rest of this thesis is organized as follows:

- in **Chapter 2**, we describe the basic concepts of concurrency in general and we also discuss the basic steps to perform the taint dataflow analysis. An idea of concurrent dataflow analysis is provided as well.

- in **Chapter 3**, we give a detailed description of the static taint analysis, its assumptions and difficulties. We discuss what the static taint analysis is and how to perform static taint analysis for complex source program.

- in **Chapter 4**, we describe the basic concepts and detailed steps of concurrent taint dataflow analysis program, as well as the difficulties involved.

- in **Chapter 5**, we provide our implementation based on IBM AppScan Source. This chapter also describes the experiment and results of our project.

- Finally, in **Chapter 6**, we provide conclusion and possible future directions for this research project.

Chapter 2

Literature Review

In this chapter, we briefly survey parts of computer science literature relevant to concurrency, taint dataflow analysis and concurrent taint data flow analysis.

The rest of the chapter is organized as follow:

- In Section 2.1, we talk about concurrency in computer science.
- In Section 2.2, we describe what is taint dataflow analysis in general.
- In Section 2.3, concurrent dataflow analysis is discussed.

2.1 Concurrency

In computer science, *concurrency* refers to multiple processes or threads that are running at the same time and may interact with each other. *Multiple Threading* is one of many ways to achieve concurrency. Multiple Threading can be achieved by distributing the tasks among several *threads* running simultaneously.

2.1.1 Multiple Threading

A *Process* in an operating system is the execution of a program [42]. A *process* is made up of at least one or multiple threads. A *Thread* is a basic unit of CPU utilization which has the smallest set of statements, a stack and a set of registers that can be managed by the operating system [42] scheduler. A thread is the minimum executable unit inside a process. Inside the process, one thread is called the *main thread*.

Threads in one process share the same virtual memory space. Threads can use their hosting process and resources. This includes system resources, global variables and direct communication between threads within the same process. Different threads of the same process can access global variables of the process. Therefore, communication between threads is easier than communication between process.

The higher overhead of communication between two processes compared to threads, makes multithreading a more appealing paradigm in the field of concurrent programming. This is particularly the case if computation happens on a single computer. Modern computers often have multiple cores, and sometimes multiple processors within each processor. Availability of multiple cores, allows multiple threads to run simultaneously, without scheduling and time-sharing.

Switching between threads consumes far fewer resources of the operating system, compared to switching between processes. To switch two processes, memory contexts need to be swapped. However, since threads share the resources of the process, to switch between two threads, only part of the memory contexts need to be swapped. For instance, local variables of the outgoing thread need to be stored, and the local variables of the incoming thread need to be read [37]. Therefore, switching between threads is more efficient than switch between processes.

Multi-threading is considered one of the best approaches to implement concurrency on a single computer. Ideally, the programmer breaks the tasks into several jobs, distributes

the jobs equally to threads, and runs all threads in parallel. A popular paradigm in multi-threading is to have all threads have the same functionality, with different data sets.

2.1.2 Mutual Exclusion

Multi-threading can increase the speed of the program, but it can also present the programmer with some challenges. Multiple running threads share process memory space and can access common objects. In this context, a *critical section* refers to a set of statements that access the shared resource, but multiple access to the resource at the same time is not allowed [4]. The critical section should not be entered by multiple threads concurrently. *Mutual exclusion* refers to the following requirement: There should be no more than one thread that accesses the same resources at the same time [4].

A popular solution to ensure mutual exclusion is to restrict threads such that only one thread at a time can access the shared resource. Using *locks* is a mechanism to enforce this policy [32]. A thread is required to acquire the lock before it is cleared to access the critical section. The thread gets the lock if and only if there are no other threads that are currently holding the lock. When a thread acquires the lock, it is required to relinquish the lock as soon as its interaction with the critical section is over.

Mutual exclusion may only be required for writing. Multiple readers may be allowed to read concurrently if no threads write to the resources. The mechanism to enforce mutual exclusion for writing is called *reader/writer lock*. The reader/writer lock can be set to work differently depending on the requirements of the system [31]:

- One approach is to give a high priority to readers. This can cause a long wait for the writer.
- Another approach is to give a higher priority to writers. High priority for writer is usually set to a system that need to update the information as soon as possible.

This can cause readers to wait a long time.

2.1.3 Deadlock Prevention

Enforcing mutual-exclusion properly is crucial to reliable multi-threading. Nevertheless, mutual exclusion is not the only challenge in multi-threading. Enforcing a mutual-exclusion policy can lead to a new problem called *deadlock*. A deadlock refers to a scenario where multiple processes or threads are waiting indefinitely for each other to release a resource before they can continue their operation.

As explained earlier, when multiple threads or processes are running simultaneously, it is not unusual that they can acquire locks. If an operation requires multiple locks to be acquired, threads or processes may fall into a deadlock. This happens when two thread or processes acquire locks, then wait for a lock to be freed to continue their operation and relinquish the locks that they currently hold. This can lead to a situation where none of the processes can release the resource, and neither can get a resource. In other words, deadlock refers to a situation where two or more processes or threads are waiting for a condition that will never be satisfied, and this makes the system stop working.

The necessary conditions for a deadlock are described below [37]:

1. **Mutual exclusion**: avoid the use the same resource simultaneously.

2. **Hold and wait**: a process or thread try to acquire a new resource before release the current resource it is holding.

3. **No preemption**: when a resource is held by a process or a thread that cannot be released before the process or thread finishes using the resource.

4. **Circular wait**: this is a situation where processes or threads are waiting for each other to release the resources.

For a deadlock to appear, all of the four conditions must be met. In other words, to avoid deadlock, it is sufficient to ensure at least one of the above conditions is not satisfied. Therefore, deadlock prevention is trying to avoid any of the four necessary conditions:

- The **mutual exclusion** condition can be removed if the resource is allowed to be accessed by multiple processes or threads concurrently. For instance, if a variable is shared in the memory, one approach to remove the exclusion condition is by making a copy of the variable before reading it so that the variable can be accessed concurrently [13].

- Removing the **hold and wait** condition can be accomplished by making a thread release all its resource when it cannot get a resource [16].

- To avoid **no preemption** condition, we can assign all required resources before processing starts. In this case, the thread will not ask for new resource before it finishes. However, in most cases, it is hard to foresee what resource are going to be required ahead of time, before running and assigning some of the required resources [13, 16].

- To avoid **circular wait**, a unique number can be given to each resource. A process or a thread can only access the resource with number x if it doesn't already have access to resource with number $y > x$. In this case, we can say that for each thread all the lower number resources are not possessed if the thread already have a greater number resource [13, 16].

2.2 Taint Analysis

In computer science, **Input Validation** is required to ensure that a program runs on correct data. To be able to detect which data requires validation, several approaches can be used. One of theses approaches is called taint analysis. Any variables modified by an untrusted source (generally speaking, it is user's input) are considered as tainted variables. The taint analysis tracks the use of tainted variables and propagates tainted information in the program. After the analysis, the result tells the programmer where (which statement) and how (trace of propagation) a variable becomes tainted. Taint analysis is able to tell which variables in the program need to be validated.

Taint analysis is performed in two ways: dynamic and static. A dynamic taint analysis labels untrusted sources and tracks them in the memory. Dynamic analysis is performed during the execution of the program. However, dynamic taint analysis has some weakness. Dynamic analysis only provides taint information of the current execution path instead of *Meet-Over-All-Paths*. Meet-Over-All-Paths is that all execution paths are considered instead of considering only the current execution path. On a call graph, meet-over-all-paths provides dataflow information of all possible paths from the entry node to any other node [29]. This gives false negative taint information. Static analysis, on the other hand, is able to provide *Meet-Over-All-Paths* taint information. The basic idea behind static taint dataflow analysis is to construct a *call graph* and then simulate the use of variables on all paths.

A call graph is a directed graph. A node in the graph represents a function in the source code and an edge represents a function call in the source code [30]. After the call graph is constructed, the propagation starts from the *entry* node of the call graph. An *entry* in the graph is a function that can be called from outside of the program. For instance, a *main*() function is a entry node, and any exported functions are also entry nodes in the graph.

The rule of taint propagation is simple. Each statement (instruction of the source code) is examined. If a variable stores a value from a tainted variable, it becomes tainted. After all paths are examined, and no taint information are changed, the taint propagation finishes. If the taint information is changed and those are the parameters passed to a function, then these functions are required to re-analyze. After that, we can tell at each statement of the program, which variables are tainted and how they become tainted (tainted by which variable).

Listing 2.1 shows an example code.

Listing 2.1: Example C Code

```
void main() {
  char buf[100];
  fget(buf, sizeof(buf), stdin);
  fput(buf, stdout);
}
```

Firstly, the call graph is constructed as shown in Figure 2.1.

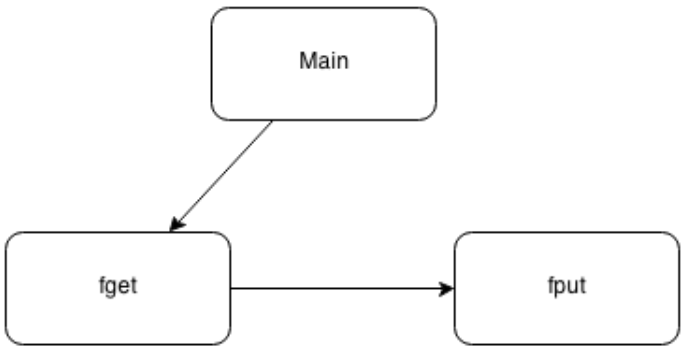

Figure 2.1: Call Graph: Example of C Code

We define that any user's input is untrusted source and thus it is a tainted source. After the taint analysis, we can get:

1. The variable buf in function $main$ is tainted by function $fget$.

2. The variable buf in function $fget$ is tainted by user's input.

3. The variable buf in function $fput$ is a tainted variable which is passed from function $main$.

2.3 Concurrent Dataflow analysis

Dataflow analysis was originally designed for compiler optimization. Dataflow analysis involves three basic stages: transformation into intermediate representation, control-flow analysis and data propagation. In general, the transformation into intermediate representation transforms the source code into an intermediate representation for further analysis. The control-flow analysis produces hierarchical flow of control within procedures and between procedures. The dataflow analysis uses the result of control-flow analysis to analyze the use of data.

The dataflow analysis is time consuming due to its nature of re-analyze. In order to improve the speed of analysis, concurrency has been considered. To perform a concurrent dataflow analysis, there are two different approaches. The first approach uses pipeline concurrency. Each stage of the analysis is performed by multiple processes/threads [3]. However, this approach has its limitations. As it is a pipeline, the speed of the analysis is limited by the slowest stage [34]. The second approach is to spilt the source code at the beginning of the analysis and performs an analysis on each code segment concurrently [35]. This approach gives rise to some difficulties in the implementation. The largest difficulty is that multiple analysis processes need to share some resource (e.g. symbol tables) and communicate with each other to update the necessary information.

In this thesis, we only work on the data propagation stage. To apply concurrency to data propagation, we choose the second approach to spilt tasks at the beginning of data

propagation and perform data propagation concurrently on multiple threads.

Chapter 3

Taint Dataflow Analysis

In this chapter, we provide an overview of taint dataflow analysis. A tainted variable is a variable that stores user's input. A taint dataflow analysis tells whether a variable is tainted after the execution of a sequence of statements.

The rest of this chapter is organized as follow:

- In Section 3.1, we briefly describe the assumption about taint dataflow analysis.
- In Section 3.2, we briefly introduce the basic concepts of taint dataflow analysis.
- In Section 3.3, we delve into more details of the taint dataflow analysis algorithm.
- In Section 3.4, we show two examples on how to perform the taint analysis. The first example is an intraprocedural analysis performed. The second example shows an interprocedural analysis.
- Finally, in Section 3.5 we conclude this chapter.

3.1 Assumption

Taint dataflow analysis attempts to track any untrusted source in the system. The result can be used to perform a security analysis which tells whether there are security problems caused by untrusted source in a system. In this thesis, user's input is the only untrusted source. A variable is called **tainted** if and only if this variable is assigned by a untrusted source (user's input) or a tainted variable.

Some issues involved in the taint dataflow analysis are **Alias analysis** and **Availability analysis**. Alias analysis tells if several variables refer to the same memory location in memory. Availability analysis tells if two variables that have the same name are referring to the same memory location in memory. For instance, in a C++ program, we have $a = C(); a = B()$, the two a are not the same as the first a is destroyed after the second a is constructed.

The alias analysis and availability analysis have already been done when transforming source code into intermediate representation. In this thesis, we assume that those variables that are the same are replaced with the same name, and those variables which have the same name but do not refer to the same memory location are replaced with different names.

3.1.1 Terms

All terms used later in this thesis are defined below.

- **Procedure** and **Routine**: Procedure or routine are functions in a program. We use these terms interchangeably through this thesis.
- **Formals** and **Arguments**: The parameters i are called **Formals**. The parameters which are used when call a function are called **arguments**.

- **ValueProperties**: ValueProperties is a property of $Variables$ which indicate whether this $Variable$ is tainted or not. It also contains a trace of tainting. The trace shows where it is tainted and where this tainted variable is used.

- **Root**: Root is a node on the call graph which does not have any incoming edges or all incoming edges are originally from itself.

- **Link**: Two or more $ValueProperties$ can be linked together. The link has its direction. When a is said to be linked to b, that means the taint value changed in b will affect the taint value of a. However, the taint value changed in a will not affect the taint value of b. For instance, if a function call is passed by value, then we say that formals are linked to arguments. If the parameters are passed by reference/pointer, then we say the formals and arguments are linked to each other.

3.2 Problem Statement

There are two kinds of taint dataflow analysis. One is known as **intraprocedural taint dataflow analysis**. The intraprocedural taint dataflow analysis indicates which variables become tainted within a procedure. In other words, intraprocedural taint dataflow analysis gives the taint values of variables before and after being used in a procedure. This includes parameter variables passed and local variables used in a procedure. The second one is known as **interprocedural taint dataflow analysis**. The interprocedural taint dataflow analysis tells which variables become tainted after one or more function calls.

Listing 3.1 shows an example of source code. Table 3.1 shows the result of intraprocedural taint dataflow analysis of the example code. Before the intraprocedural taint dataflow analysis is performed, all variables are defined as **untainted**. After the intraprocedural analysis, the variable *buf* becomes **tainted** and the result also shows that

buf becomes **tainted** at statement $S2$ where *buf* get an input from user.

Listing 3.1: Example of Intra-Code

```
Procedure main(){
   S1:char *buf;
   S2:fgets(buf);
   S3:fputs(buf);
}
```

Variables	Before Analysis	After Analysis
buf	Untainted	Tainted (at S2 by user's input)

Table 3.1: Intraprocedural Taint Dataflow Analysis Result

Listing 3.2 shows an example of source code. Table 3.2 shows the result of interprocedural taint dataflow analysis of the example code of Listing 3.2. In Table 3.2, a variable $M.N$ means that N is a local variable of procedure M. For instance, $main.buf$ means the local variable *buf* that is defined and used in procedure *main*.

In the example below, all variables $main.buf$, $input.x$ and $output.x$ are all initially defined as **untainted** before the analysis. After the analysis, the results show that $main.buf$ becomes **tainted** in procedure **input**, then returns back to **main** and later is used in **output**. $input.x$ becomes **tainted** as it gets an input from user in **input** and returns to **main**. $output.x$ also becomes tainted because it uses the value of a tainted variable passed from **main**.

Listing 3.2: Example of Inter-Code

```
Procedure main(){
   S1:char *buf;
   S2:input(buf);
```

```
   S3:output(buf);
}

Procedure input(char *x){
    S4:fgets(x);
}

Procedure output(char *x){
   S5:fputs(x)
}
```

Variables	Before Analysis	After Analysis	Tainting Trace
main.buf	Untainted	Tainted	input-main-output
input.x	Untainted	Tainted	input-main
output.x	Untainted	Tainted	main-output

Table 3.2: Interprocedural Taint Dataflow Analysis Result

3.2.1 Taint Dataflow Analysis: Basic Steps

Taint dataflow analysis consists of three basic steps as was discussed in the previous chapter. The first step is transformation into intermediate representation. This step reads the source code and transforms the source code into a language-independent representation. After transformation is done, control flow analysis is performed to generate flow graphs for each procedure as well as a call graph to represent relationship between procedures. Propagation analysis is performed to propagate taint values on the call graph.

Taint analysis in its core relies on tainting any untrusted source used by the system[1].

[1]In this thesis we focus on user input as the only source of untrusted input. Any variables contain

After performing taint dataflow analysis, all tainted variables and their *tainting trace* in the program are produced. The tainting trace indicates that for a particular tainted variable, in which procedure it became tainted and in which procedures it was used. Formally, a variable is set to be tainted if and only if that variable is on the left-side of an assignment statement and a tainted source (tainted variables or user's input) is used in the expression on the right-side of the same assign statement. An exception to that are pointer variables. Pointer variables allow two or more variables to share the same memory space. In this case, if one of the variables becomes tainted then all variables that share the same memory location become tainted.

3.2.2 Background

Dataflow analysis includes two different types of analysis: *intraprocedural analysis* and *interprocedural analysis* [22]. Intraprocedural analysis is performed within a procedure. It determines how data is used in a procedure. Interprocedural analysis is performed between procedures. It determines how a variable is used when passing between procedures (procedure calls). The interprocedural analysis is capable of providing how the data is used among the entire program while the intraprocedural analysis only provides dataflow information in individual procedures.

Interprocedural analysis can be performed in two ways: *flow-insensitive* and *flow-sensitive*. Flow-insensitive analysis is the analysis that does not consider the control flow in a program while the flow-sensitive analysis cares about control flow. For instance, a flow-sensitive analysis can tell that routine B is always executed after routine A. A flow-insensitive analysis may analyze routine A first even though routine A is called only after routine B is called in the program.

In this thesis, we only consider the flow-sensitive analysis as the flow-insensitive anal-

user's input will be marked as tainted variables

ysis produces false positive/negative results.

3.2.3 Definition of Tainted Values

The data propagation analysis gathers the information whether variables in a procedure become **tainted** after execution. For a single variable var, in the following, we assume that a **boolean** variable var^* represents the taint values of that variable. The boolean value **false** represents **untainted** and **true** represents **tainted**.

We define a taint assignment function $TA : V \Longrightarrow Boolean$, which tells the taint values of a set of variables V. We then define a taint transmission function $TT_s : TA \Longrightarrow TA$ which tells the taint values of the variables V after the execution of a sequence of statements s.

The detailed definition of a TA and TT is given in Table 3.3.

Name	**Taint Values**	**Statements**
Initial taint values	$TA(v) = false$, for all $v \in V$	
User's input	$TA(v) =$ true	v = getchar()
Assignment	$TT_s(TA)(x) = \begin{cases} \vee_{i=1,..,n} TA(v_i), \text{ if } x = v \\ TA(x), \text{ if } otherwise \end{cases}$	S: $v = v_1 + .. + v_n$
Multiple Statements	$TT_S(TA)(v) = TT_{S2}\{TT_{s1}(TA)(v)\}$, where S is a sequence of statements, s1 is the first statement of S, S2 is the remaining statements of S	S: { s1: S2: { } }
IF-statement	$TT_S(TA)(v)$ $= TT_{S1}(TA)(v) \vee TT_{S2}(TA)(v)$	S: if (c_1) then: S1 else: S2

Table 3.3: Taint Dataflow Analysis Result

- *Initial taint values*: At the beginning of taint dataflow analysis, all variables are considered untainted. In other words, the initial taint values of any variable is false.
- *User's Input*: When a user's input is assigned to a variable, that variable becomes tainted.
- *Assignment*: When a statement is an assignment, the taint values of the right-side variables propagate to the left-side variable. If one of right-side variables is tainted, then the left-side variable becomes tainted
- *Multiple Statements*: When TT is applied on a sequence of statements, TT is applied to the each statement in the same sequence. The output of each TT

function is the input of next TT function.

- *IF-statement*: When the statement is an if-statement, the taint value is the disjunction of the $TT_S1(TA)(v)$ and $TT_S2(TA)(v)$.

Loop

Loop statements can be seen as a combination of **if-statements** and **jump** statements. To analyze loop statements, it is necessary to determine when the loop terminates. This also indicates when the analysis of a loop is finished. There are two approaches to determine when a static analysis of a loop is finished. Sometimes, a loop statement has an integer-valued variable that tells how many iterations are needed for this loop [22]. This variable is called the ***induction-variable***. The first approach analyzes the loop body several times until the loop counter reaches the upper bound value of the induction-variable.

However, there are some loops that do not have an induction-variable. Listing 3.3 shows an example of an infinite loop that does not have an induction-variable.

Listing 3.3: Infinite Loop

```
S1:while(true){
   S2: a = b;
   S3: b = c;
   S4: c = a;

}
```

As we can see in Listing 3.3, the first approach is not able to tell when the analysis is finished as there is no such induction-variable.

To address this issue, a second approach uses the idea of **fixed-point**. x is a fixed-

point of function f if $f(x) = x$. In taint dataflow analysis, a fixed-point taint assignment of a loop is $TT_{S_{loop-body}}(TA)(v) = TA(v)$ where v is any variables used in the body of the loop. In a analysis of a loop, a variable can become **tainted** if the loop may taint this variable.

As the example code shows in Listing 3.3, we show a detailed analysis using TT and TA functions defined before. The detailed analysis can also apply to a more common loop in which $S1$ is: *for (int i=0; i¡123;i++)*. The fixed-point approach simply ignores the **induction-variable**.

- 1st Iteration:

  $TA_{S1_{body}}(a, b, c) = 0, 0, 1$

  $TT_{S1_{body}}(TA)(a, b, c) = 0, 1, 0$

  At this point, we know that b can become tainted.

- 2nd Iteration:

  $TA_{S1_{body}}(a, b, c) = 0, 1, 0$

  $TT_{S1_{body}}(TA)(a, b, c) = 1, 1, 1$

  At this point, we know that a and c can become tainted as well. Like the if-statement, we consider all possible tainted executions so we get a, b and c can become tainted.

- 3rd Iteration:

  in: $TA_{S1_{body}}(a, b, c) = 1, 1, 1$

  out: $TT_{S1_{body}}(TA)(a, b, c) = TA_{S1}(a, b, c) = 1, 1, 1$

  At this point, we have $TT_{S_{loop_body}}(TA)(v) = TA(v)$. In other words, we reach the fixed point.

The fixed-point approach also has its weaknesses. In general, a function may have more than one fixed point depending on the input; this approach only gives the result of one fixed point. And, the fixed-point approach ignores the **induction-variable** which may lead to a loop being analyzed more than its maximum number ofiterations. In taint dataflow analysis, this may lead a false negative and/or positive result. The reason is that loop termination is a dynamic behaviour which can not be perfectly determined by static analysis.

The result of analyzing Listing 3.3 shows above, we can only find that if the variables may become tainted as we have no idea how many iterations are needed. In this case, all variables may be tainted. A dynamic analysis is required to correctly analyze a loop.

3.2.4 Interprocedural Taint Dataflow Analysis

We have given the definitions of TA and TT in a procedure. An issues that remains is how to apply TA and TT when there is a function call. In this section, we give detailed ideas of how to apply TA and TT depending on different type of function calls.

Non-Recursive Function Call

Listing 3.4: Example of Non-recursive Function Call

```
Procedure p(F,R){
   S:{
     ...
     S2: return k
   }
}

Procedure main(){
```

```
  ...
  S0: ....
  S1: a = p(A_F,A_R)
  ...
}
```

In Listing 3.4, procedure p is shown as a generally defined procedure. F is a set of variables which are passed by value. R is a set of variables which are passed by reference (or pointer). $S2$ is a return statement in p which returns the value of k to the calling procedure.

In procedure *main*, procedure p is called. A_F is a set of argument variables which are passed to procedure p by value. A_R is a set of argument variables which are passed to procedure p by reference (or pointer). a is assigned to the returned value of procedure call of p.

The TA and TT functions of $S1$ can be defined as follows:

$TA(F) = TA(A\_F)$

$TA(R) = TA(A\_R)$

$$TT_{S1}(TA)(x) = \begin{cases} TT_S(TA)(x), \text{ if } x \in A\_R \\ TT_{S0}(TA)(x), \text{ if } x \in A\_F \\ TT_{S2}(TA)(k), \text{ if } x = a \end{cases}$$

Recursive Function Call

A recursive call leads to an infinite analysis in the static dataflow analysis. This problem is similar to an infinite loop. The fixed point solution can be applied to address this issue.

In this thesis, we use a different approach. We only analyze a recursive call once. The reason is that, in practice, analysis of a function is memory consuming. Every

local variable used in the function needs memory space when the function is called. If the fixed-point approach is applied, the memory usage is going to exceed the memory limitation of the system. Instead, a call stack is constructed, and we skip a call if it has been already in the call stack.

In other words, when analyzing a function call, which has already been in the call stack, our strategy is to simply skip this function call and continue to analyze the next statement.

Pointer Functions

Pointer functions are considered as dynamic properties of the program. The static analysis can not provide runtime information of these functions and variables. A dynamic dataflow analysis is needed to address this issue.

3.3 Algorithm Elaboration

In this section, an algorithm for building the control flow graph is presented first and an algorithm of constructing call graph is discussed as well. After that, an algorithm of propagating taint dataflow information on control flow graph (intraprocedure taint dataflow analysis) and call graph (interprocedure taint dataflow analysis) is provided.

Build Control Flow Graph

As discussed earlier, the control flow graph of each procedure is constructed first. The control flow graph of a procedure can be seen as a flowchart of a procedure. The idea of building a control flow graph is that each statement is anaylzed and based on the type of a statement, a flowchart is applied accordingly [1]. For instance, the control flow graph of a **if-then-else** statement of Listing 3.5 shows in Figure 3.1 below.

Listing 3.5: Example of if-then-else

```
{
  S0:if (condition) then
      S1:{
      }
    else:
      S2:{
      }
  S3:...
```

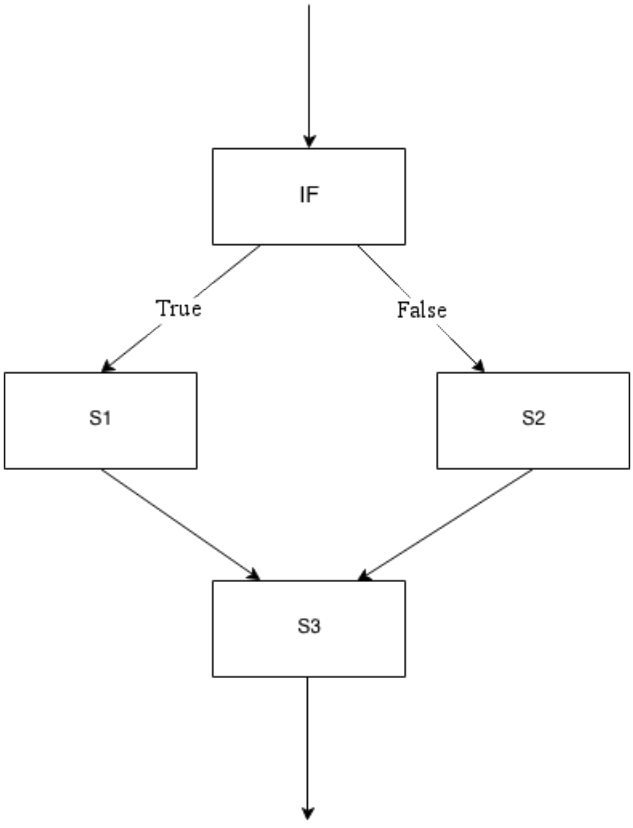

Figure 3.1: Control Flow graph

Loop statements can be more complicated. However, in this thesis we only focus on dataflow analysis, we assume that the control flow analysis is done before the dataflow analysis is performed.

Build Call Graph

In this step, each procedure is analyzed. Based on each function calls being called in each procedure, a breadth-first algorithm is used to build the call graph of the entire program. A call graph is a multiple rooted graph. Each node represents a procedure of the source program. Edges are directed edges which represent a call from one procedure to another. Figure 4.4 shows an example of call graph.

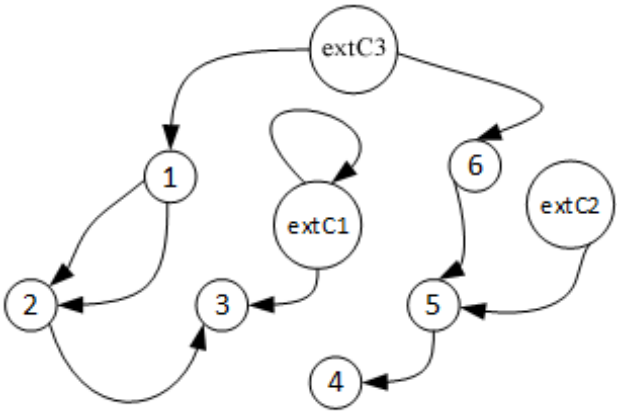

Figure 3.2: Call Graph

A root node represents a procedure which has an external caller. An external caller means that this procedure can be called from outside of the program. A common example would be the *main* function in a program or any exported function.

Propagate taint values

The propagation of taint values is performed on the call graph. The propagation starts from a root node on the graph. The basic idea is that all the procedures (nodes) on the

graph are recursively visited, and taint values are propagated through these visits.

Intraprocedural To analyze each procedure, an intraprocedural taint analysis is performed first. Each statement is analyzed using depth-first-serach based on the control flow graph of this particular procedure. The TT and TA functions are applied for each statements unless that statement is a call statement. Eventually, the analysis tells which variables of this procedure are tainted.

Interprocedural If a statement is a function call or procedure call, the propagation between procedures is performed (interprocedure analysis). When a procedure p_i is called by p_j, the arguments in p_j are linked to the formals in p_i. As a result, the procedure p_i's formals have inherited all tainted information from its calling p_j's arguments. Any taint values updates to the p_j's arguments have effects on p_i's formals. p_i must be analyzed again if the taint values passed to p_i have changed.

The analysis continues until no taint values of formals are changing. This is similar to the fixed-point approach when dealing with loops.

The algorithm of propagating the tainted information is given below:

1. Recursion Check

   (a) If this procedure has already been in the *callStack*, do not analyze this procedure. Go to **step 4**.

   (b) Otherwise, push this procedure into *callStack*. Go to **step 2**.

2. Analyze each statement based on the control flow graph using depth-first-search.

   (a) If the statement is not a return statement nor a function call, TT and TA functions are applied. All variables involved change their tainted information accordingly. **Continue** to the next statement.

(b) If the statement is a return statement, return the tainted value of the returning variable to its callsite as well as all parameter variables passed by reference or passed by pointer. Go to **step 4**.

(c) If the statement is a function call, Go to **step 3**

3. Propagate among procedures

(a) Link arguments to formals of the called procedure (if the arguments are passed by reference or pointer, link the formals to arguments as well).

(b) Propagate taint values on the called procedure

(c) Unlink arguments from formals from the called procedure. Go to **step 2** and analyze next statement.

4. If any taint values of formals changed during this analysis, go back to **Step 2** and re-analyze this procedure until taint values of formals do not change (fixed-point).

5. Pop this procedure out of the *callstack*

3.4 Examples

In this section, we give an example of performing intraprocedural taint dataflow analysis as well as an interprocedural taint dataflow analysis. The sample codes are provided in Listing 3.6 and Listing 3.7 below.

Listing 3.6: Sample Code 1

```
void main() {
   char buf[100];
   fgets(buf, sizeof(buf), stdin);
   fputs(buf, stdout);
```

```
}
```

Listing 3.7: Sample Code 2

```
void main() {
   char buf[100];
   input (buf, sizeof(buf);
   output(buf);
}

static void input(char *x, int len) {
   fgets(x, len, stdin);
}

void output(const char *y) {
   fputs(y, stdout);
}
```

3.4.1 Intraprocedural Example

The intraprocedural taint dataflow analysis is using the sample code in Listing 3.6. This example is intend to show how analysis will take place within a singe procedure.

The first step involved is transforming source code to a intermediate representation. The high-level intermediate representation looks something like:

Procedure: **main**

Formals:

Variables: **buf**, blocksize=100

Operation:

Call: **fgets**

Argument 1: **buf**

Argument 2: 100

Argument 3: stdin

Call: **fputs**

Argument 1: **buf**

Argument 2: stdout

Table 3.4: Intermediate Representaion

The second step produces a control graph shows below in Figure 3.6

Figure 3.3: Control Flow Graph

After that, taint values are propagated on the control flow graph. The $fgets$ is analyzed first. As we can see, this function reads user's input and stores the value to first argument. As the first argument is passed by pointer in this case, *buf* becomes tainted. Following the control flow graph, $fputs$ is analyzed. As *buf* has already become

tainted, we can say $fputs$ use a tainted variable.

Finally, we can say that, in procedure *main* the local variable *buf* is tainted by $fgets$ and later used by $fputs$.

3.4.2 Interprocedural Example

The interprocedural taint dataflow analysis is using the sample code in Listing 3.7. This example is intend to show how analysis will take place between different procedures.

The first step and second step are the same as intraprocedural. Intermediate representation and control flow graphs are generated for each procedure. The tables and figures below show the result of intermediate representation transformation and control flow analysis.

Procedure: **main**
- Formals:
- Variables: **buf**, blocksize=100
- Operation:
  - Call: **input**
    - Argument 1: **buf**
    - Argument 2: 100
  - Call: **output**
    - Argument 1: **buf**

Table 3.5: Intermediate Representaion (main)

Procedure: **input**

Formals: **x, len**

Varialbes:

Call: **fgets**

Argument 1: **x**

Argument 2: **len**

Argument 3: stdin

Table 3.6: Intermediate Representaion (input)

Procedure: **output**

Formals: **y**

Varialbes:

Call: **fputs**

Argument 1: **y**

Argument 2: stdout

Table 3.7: Intermediate Representaion (output)

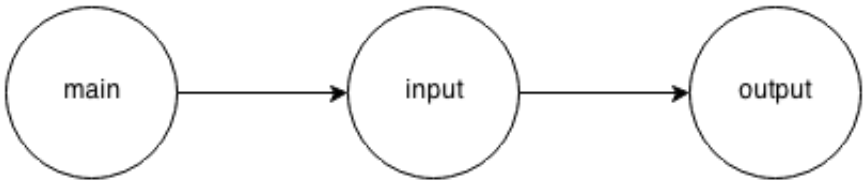

Figure 3.4: Control Flow Graph (main) and Call Graph

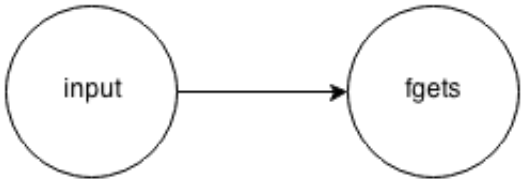

Figure 3.5: Control Flow Graph (input)

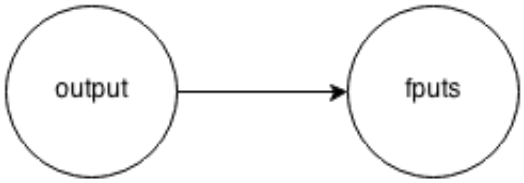

Figure 3.6: Control Flow Graph (output)

After that, taint values are propagating between procedures on the call graph.

1. The *main* procedure is analyzed first as it is the only root node in the call graph.
2. Check recursion and push *main* into call stack
3. The intraprocedure taint dataflow analysis is perform on *main*. The *input* is being analyzed first
   (a) **buf** and **x** are linked to each other as the parameter is passed by pointer
   (b) Check recursion and push *input* onto call stack
   (c) Analyze $fgets$ and we find that **x** and **buf** become tainted
   (d) The parameter passed to *input* has been changed its taint value. Re-analyze is required of *input*.
   (e) Analyze *input* again until reaching fixed-point
   (f) pop *input* out of the call stack

4. The intraprocedure taint dataflow analysis continues on *main*. The *output* is being analyzed.
   (a) **buf** is linked to **y**
   (b) Check recursion and push *output* onto the call stack
   (c) Analyze $fputs$ and we find that a tainted variable is used
   (d) The parameter passed to *output* has not been changed its taint value. Re-analysis is not required of *output*.
   (e) pop *output* off of the call stack
5. No parameters passed to *main* have been changed its taint value. Re-analyze is not required of *main*.
6. pop *main* out of the call stack
7. finished

Finally, we can say that the variable *buf* of *main* is tainted in *input* by $fgets$ and then later is used in *output* by $fputs$.

3.5 Conclusion

In this chapter, we described the problem of taint flow analysis in detail. The definition of taint analysis, and a solution to solve the taint dataflow problem on a call graph were presented. Based on these algorithms, we then presented algorithms to propagate taint values within and between procedures. Finally, we introduced two examples to show how taint values are propagating within and between procedures.

Chapter 4

Introducing Concurrency into Taint Dataflow Analysis

4.1 Introduction

In Chapter 3, we showed how to perform taint dataflow analysis by traversing the call graph, and propagating the taint information. In this chapter, we describe how to parallelize the algorithm, and perform the taint dataflow analysis concurrently. This chapter is organized as follows:

- In Section 4.2, we describe the basic ideas to perform taint dataflow analysis concurrently.
- In Section 4.3, we give needed data structures before discussing the algorithm.
- In Section 4.4, we describe an architecture to perform taint dataflow analysis concurrently.
- In Section 4.5, we describe our concurrent taint dataflow analysis algorithm.

- In Section 4.5.3, two different approaches for sharing information between threads are provided. We also discuss the reason we chose one approach instead of the other.
- In Section 4.5.4, we explain the merging process of results among the threads.
- Finally, in Section 4.6, we conclude this chapter.

4.2 Static Taint Concurrency

As discussed in Chapter 3, we have shown that taint dataflow analysis is started from the root node in a call graph. During an analysis, an intraprocedure analysis is performed for each procedure and when a function call is made, an interprocedure analysis is involved. Eventually, the analysis can tell which variables become tainted and how they become tainted (in which procedure).

In Chapter 3, we only give an example which has only one root node in the call graph. The root node is either the *main* function of the program, or a function that can be called externally from an external caller. For example, in Windows Operating System, a library usually has a field called export table. The table contains functions that are able to be called by other programs. These functions are considered as roots. In a program, there are usually more than one root nodes. Therefore, each root node is going to be analyzed until all root nodes are analyzed. In a sequential analysis, the root nodes are analyzed one by one.

To parallelize taint dataflow analysis, we simply start from different root nodes simultaneously. We split the call graph into several sub-graphs and analyze each sub-graph concurrently. After splitting the graph into sub-graphs, each sub-graph has only one root node. The root node is considered to be the starting point of each analysis.

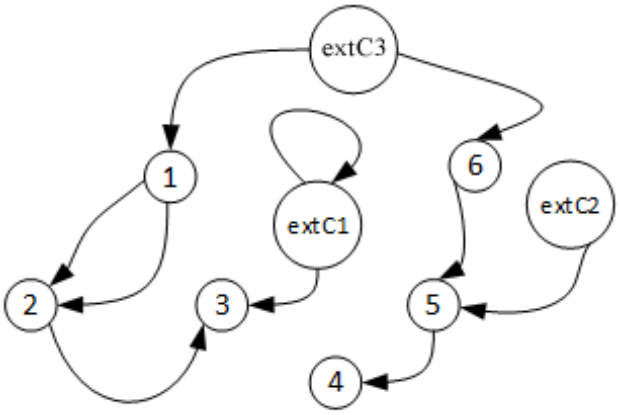

Figure 4.1: Call Graph

Figure 4.1 shows an example of call graphs. As we can see in the figure, the $extC1$, $extC2$ and the $extC3$ functions are the root nodes.

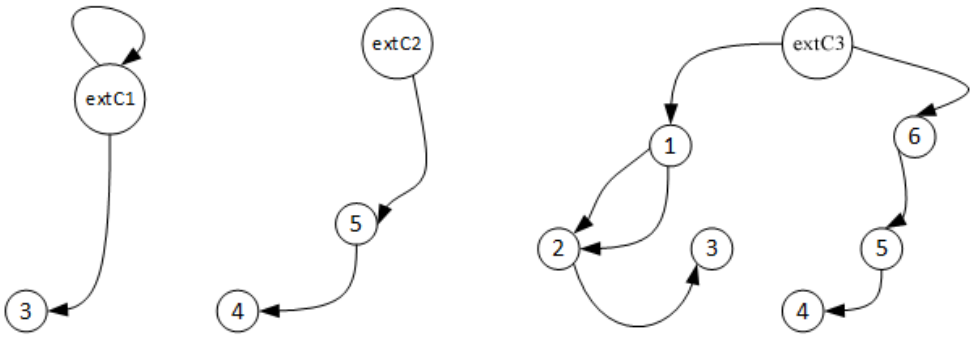

Figure 4.2: Sub-graphs

Figure 4.2 shows the sub-graphs after splitting the call graph. As the figure shows, each sub-graph has only one root node. Theses sub-graphs can be analyzed concurrently. Also, these sub-graphs usually have some overlaps and the overlaps are analyzed serveral times once, for each root.

To analyze a sub-graph, the algorithm introduced in Chapter 3 can be applied. In a concurrent analysis, each sub-graph is analyzed separately using the algorithm discussed

in Chapter 3. However, issues like deadlock and sharing information between each analysis have not been taken into consideration. The details of such issues is going to be discussed in the later sections.

4.3 Background

4.3.1 ValueProperties

During taint dataflow analysis, a *ValueProperties* is a container to represent tainted values of a variable. It also stores memory size, data size, and the data type contained (e.g. heap, stack, static or constant) of the variable. Instead of operating on a variable, any operation on a variable is modelled as an operation on the $ValueProperties$ of that variable. By the end of the operation, the $ValueProperties$ has the properly tainted value of the variable.

$ValueProperties$ of different variables can be *linked* and *unlinked* from each other. By linking a $ValueProperties$ A to $ValueProperties$ B, operations on A will effect B. For example, a caller procedure p_1 has a variable v_1 which is passed to the called procedure p_2 by value. p_2 uses v_1 and assigns v_1 to its own variable v_2. When the taint information in v_1 is changed in p_1, v_2 in p_2 also changes as they are linked. This enables propagating taint information across calls and variables.

$ValueProperties$ can also store trace taint propagation. Figure 4.3 shows the structure of trace information in a $ValueProperties$. As the figure shows, a is a variable in the routine A, b is a variable in the routine B and d is a variable in the routine D. Routine A calls routine B and routine B calls routine D. a is assigned to b and b is assigned to d. The $ValueProperties$ of d is linked to the $ValueProperties$ of a and b. If a is tainted, then the tainting trace of d is ABD

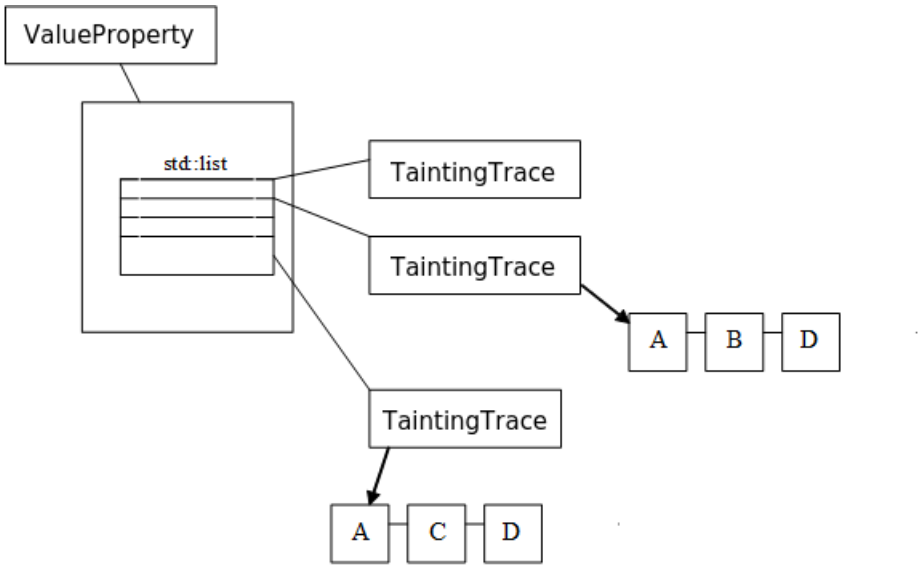

Figure 4.3: Tainting Trace Information in a ValueProperties

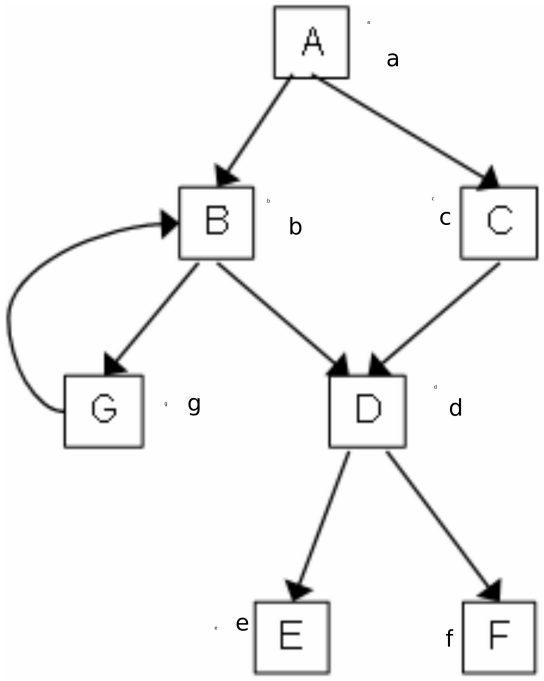

Figure 4.4: Call Graph

4.3.2 Routine

In our concurrent taint dataflow analysis approach, a routine data structure is used to represent a call in the program. On the call graph, each call is represented by a node. A routine contains several objects that are used for the analysis. Figure 4.5 shows the class diagram of a routine.

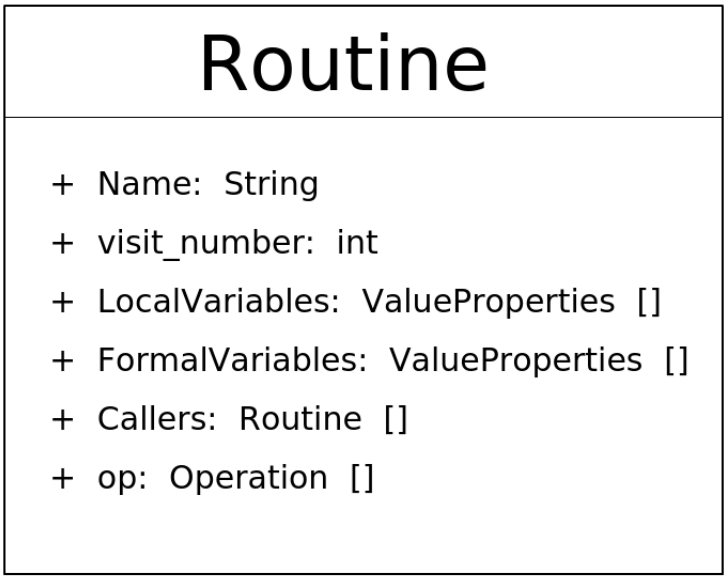

Figure 4.5: Class Diagram of Routine

The attributes of the routine object are explained below.

- *Name*: the function name of the routine.
- *visit_number*: A time stamp that stores the last time any *ValueProperties* in the routine was modified.
- *LocalVariables* stores the local variables used in this routine.
- *FormalVariables* are variables that are passed to this routine.

- *Callers* lists all the routines call to this routine.
- *op* lists all operations in a routine. An operation can be a call statement, a return statement or an assign statement in the source code.

4.4 Architecture

This thesis is a study based on a given taint dataflow analysis program. The concurrent taint dataflow analysis we provide is more complex to adapt to the given program. However, the idea of introducing concurrency into taint dataflow analysis can still apply to any other cases. In this section, a basic architecture needed to perform a concurrent taint dataflow analysis is introduced.

The concurrent taint dataflow analysis consists three basic components: *Intraprocedural analysis*, *Taint Analysis* and *Post Analysis*. Figure 4.6 shows the overall architecture of our concurrent dataflow analysis.

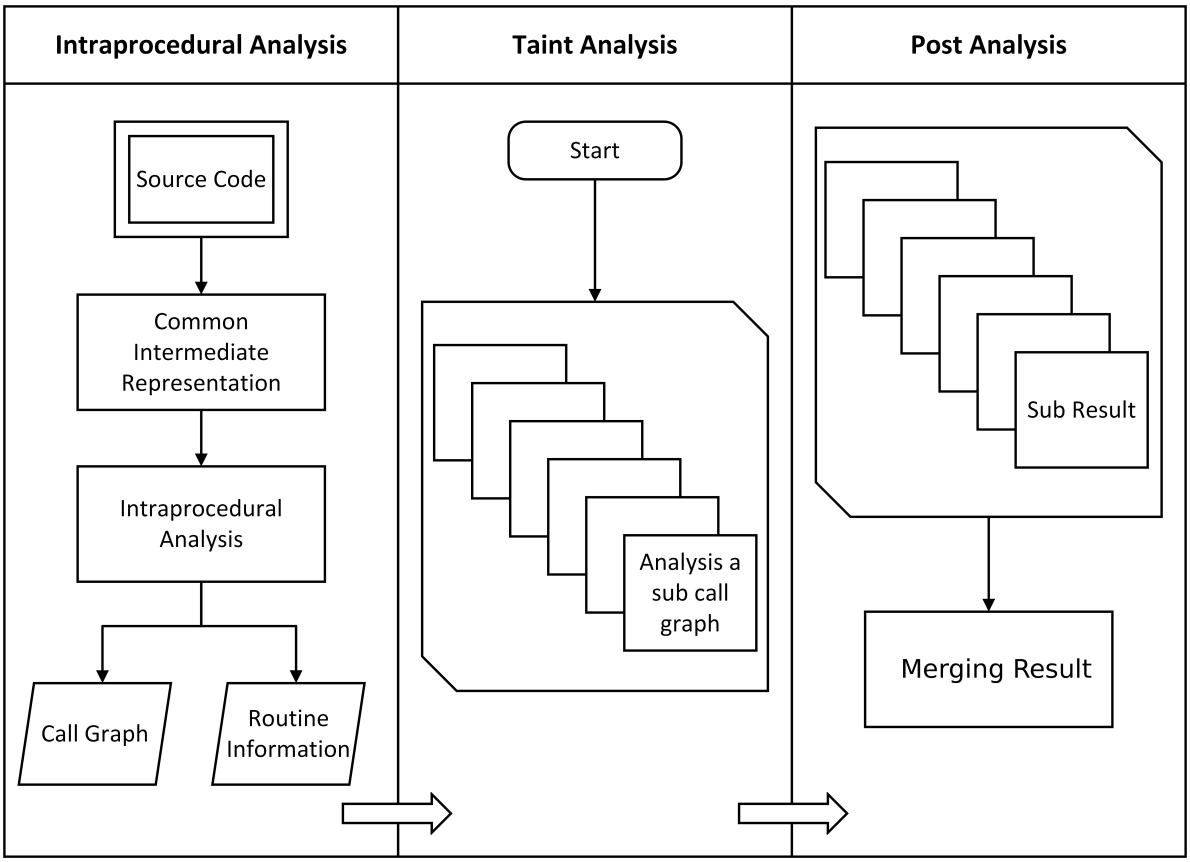

Figure 4.6: Overview of Concurrent Taint Dataflow Analysis

The *intraprocedural analysis* starts by reading the source code. An intermediate representation of the source code is produced. An intraprocedural dataflow analysis is performed on the intermediate representation. The intraprocedural analysis analyze all procedures individually. It produces a control flow graph of each procedure as well as a call graph. The control flow analysis produces the order in which the individual statements are executed [1] in a procedure. The call graph provides the order in which each function is called.

The *taint analysis* starts with creating a thread manager object. The thread manager object creates a master thread and working threads. The master thread is responsible for splitting the call graph, and assigning tasks to working threads. The working threads

are responsible for performing the analysis on a given sub-graph.

Ideally, the same number of working threads as the number of sub-graphs are created, each thread gets a sub-graph and performs analysis on it. When the analysis is done, working threads merge the results of each sub-graph by using the union operation to get the result of the entire call graph.

4.5 Algorithm

The concurrent taint dataflow analysis algorithm consists of three steps. The first step is to split the call graph and assign tasks to some threads. This is done by the thread manager. The thread manager splits the tasks and creates some working threads. Each working thread will be assigned with the task of analyzing one sub-graph. In the second step, the working threads perform the analysis concurrently. Finally, in the third step, results are merged together, and the analysis is complete.

The rest of this section is organized as follow:

- In Section 4.5.1, we describe algorithm of splitting call graph and managing threads in the thread manager.
- In Section 4.5.2, we describe a detailed algorithm for propagating taint values on a sub-call graph.
- In Section 4.5.3, we discuss two different approaches to exchange information between threads.
- In Section 4.5.4, we give an algorithm to merge the result from different threads

4.5.1 Splitting Graph

In this stage, the call graph is split into several sub-graphs. As discussed before, the call graph is a graph which has multiple root nodes. Each sub-graph is rooted by one of those root nodes. In other words, a sub-graph has only one root node and each sub-graph has a different root node.

This process is called *split graph*. A breadth first traversal on the original call graph is performed and sub-graphs are built at the same time. Listing 4.1 shows the algorithm to split the graph.

Listing 4.1: Algorithm of Split Call Graph

```
Procedure splitCallGraph (callGraph)
   //returns value, list of sub-graphs
   subGraphs = [];
   //list of all root nodes in callGraph
   rootList = callGraph.getRoots();
   //start to build sub-graph for each root node,
   //until all root nodes are visited
   while ( (root=rootList.getNext()) !=NULL)
      //initialize this sub-graph as empty
      subGraph = {}
      //add root node to this sub-graph
      subGraph.addRoot(root)
      //start BFS to construct this sub-graph
      while ( (r = subGraph.getNextNode()) != NULL)
            For (int i=0; i< r.numOfCalls(); i++)
               //add a new node to this sub-graph
                subGraph.addNode(r.Calls[i])
               //add a new edge to this sub-graph
```

```
            subGraph.addEdge(r, r.Calls[i])
    end while
    //add this sub-graph to return list
    subGraphs.add(subGraph)
end while
//return all sub-graphs
return subGraphs
```

In Listing 4.1:

- *callGraph* is the original call graph.
- *subGraphs* is the set of sub-graphs.
- *callGraph.getRoots*() returns all root nodes in callGraph.
- *rootList.getNext*() returns the next root node in the list
- *r.numOfCalls*() returns the number of called procedures of a node (routine)

4.5.2 Concurrent taint dataflow analysis

After the sub-graphs are built, each sub-graph is analyzed separately and concurrently. The analysis is performed by traversing the sub-graph. The analysis starts from the root node of the sub-graph.

To traverse a graph, there are many options. For instance, depth first or breadth first traversals. In this thesis, depth first is chosen because it is capable of maintaining the order of calls in the analyzed program. The next node to visit is always the node called by the current node. A *call path* is a path from a root node to a node on the sub-graph.

Listing 4.2 shows the overview of the traversal algorithm. The term *routine* also refers to *node* on the sub-graph.

Listing 4.2: Overview of Traversal Algorithm

```
Procedure ProcessRoutine(Routine r, int VisitNumber)
   if r is on the call stack
      // if r has already been on the call stack, it means this r has been
          analyzed. In other words, it is a recursion call to r
      Return
   push r on the call stack
   r->visit_number = starting_visit_number = VisitNumber
   do
         starting_visit_number = r->visit_number
         for each Operation op in r
            ++global_visit_number;
            if op->isNotACall
               // if this op is not a call, simply applied TT and TA functions
               PropagateTaint(op,global_visit_number)
            else if op->isCall
               subr = op->routine;
               if (op->modified_number > op->visited_number OR
                   op->isNotVisited)
                        linkArgToFormal(op)
                        if op->isNotVisted
                           for each Formal f in op
                              Propagate_Upward(f, global_visit_number);
                           op->isNotVisited = FALSE;
                        ProcessRoutine(subr, global_visit_time);
                        unlinkArgFromFormal(op)
            op->visited_number = ++global_visit_number;
   while(r->visit_number != starting_visit_number) //if re-visit needed, keep
```

```
    looping
pop r off the call stack
```

On entry to analyzing a routine, a unique number $VisitNumber$ is given. This number is used as a flag to tell whether a routine needs to be re-analyzed or not. Then, this routine is checked whether it has been analyzed for this call path or not. This is intended to avoid processing a recursively called routine twice as was discussed in **Chapter** 3.

After the recursion check, this routine is pushed onto the *callstack* for further recursion check. r's current visit number and this analysis's starting visit number are set to $VisitNumber$.

A *do-while* loop is entered to examine all operations of r until there is no re-analyze flag raised. A re-analyze flag is raised if and only if the taint values of parameters passed to r are changed. When entering the *do-while* loop, the *starting_visit_number* is set to r's current visit number. In other words, we assume that taint values of parameters passed to r are not being changed at the beginning of the analysis. Then the analysis loops through every operation of r and analyzes each operation in turn.

The steps involved to analyze each operation are the following:

1. Increase the global visit number *global_visit_number*. This generates a new unique visit number.

2. If this operation *op* is not a function call, TA and TT are applied to propagate the taint value accordingly. If the tainted value of a variable is changed to tainted, then all variables it links to are changed to tainted. In addition, this *op*'s $modified\_number$ is set to the current $global\_visit\_number$.

3. If this operation *op* is a function call, then the interprocedure propagation is involved.

(a) If *op* has been analyzed and *op* uses data that has not changed its tainted value, then an analysis is not needed. This makes sure that a routine is not analyzed unnecessarily twice.

(b) If *op* has not been analyzed or *op* uses the data that has changed, then an analysis of this *op* is required.

i. The argument variables are linked to the formal variables.

ii. If this is the first visit of this *op*, then a process called *Propagate_Upward* is performed to notify all upper nodes on the call path. This tells all upper nodes that there are new formal variables that are linked to argument variables. Details of *Propagate_Upward* are given in next paragraph.

iii. This *op* is analyzed as a common routine *subr* by calling $ProcessRoutine(subr, global_visit_number)$. The current *global_visit_number* is passed as a starting visit number for processing *subr*.

iv. The argument variables are unlinked from the formal variables.

(c) This *op*'s visited number is set to a new global visit number

4. If this routine uses data that has changed its taint value during this analysis, a re-analysis is required.

After all necessary re-analysis is performed, the analysis of this routine is done. This routine is popped from the call stack

Propagate Upward The upward propagation propagates taint information from a called routine to the caller routine and upper routines on the sub-graph. The upward propagation happens when a routine is analyzed for the first time. This tells its upper nodes that all linked variables' taint value may have been changed by the called routine.

Listing4.3 shows a sample source code.

Listing 4.3: Sample Source Code 1 (C++)

```
int main()
    char s = f;
    input(&f)

void input(char *x)
    fgets(x)
```

In the sample source code, the parameter passed to function *input* is passed by reference. Before analyzing the *input* function, all variables linked to the argument f need to be linked to the formal parameter x as well. A upward propagation is necessary.

As described in Listing 4.2, the function *Propagate_Upward* is called to inform a routine's upper nodes that some variables' taint value may have been changed by this routine.

The algorithm to propagate tainted information upward is shown below in Listing 4.4.

Listing 4.4: Propagate Upward

```
bool Propagate_Upward(ValueProperties vp,int VisitNumber)
   if vp is not associated with a variable or formal
      return False
   ValueProperties arg_vp := vp->argument;
   if arg_vp is not associated with an argument in the caller
      return false
   ValueProperties vf_vp := arg_vp->formal;
   if vf_vp is not from root
      status := Propagate_Upward(vf_vp, VisitNumber);
   else if vf_vp is variable
```

```
   link(vf_vp, vp)
   Status := (vf_vp is changed by link) ? true : false;
   else //vf_vp is a formal
      Status := false;

if (!Status)
   return false

// update the visit number for everything involved
vf := vf_vp->variable;
vf->visit_number := VisitNumber;
r  := vf->routine;
r ->visit_number := VisitNumber;
op := arg_vp->argument->operation;
op->modified_number := VisitNumber;
r := op->routine;
r ->visit_number := VisitNumber;
return true;
```

The first step to propagate upward is to check if this variable is associated with any local variables or formal variables. For instance, if this variable is a constant value, then there is no need to propagate upward. Besides, if *vp* is in the root routine then there is also no need to propagate upward. After that, all argument variables and formal variables associate with this *vp* are detected. If the associated formal variables are still associated with any other variables in an upper caller, then the *Propagate_Upward* notifies the upper callers until it reaches the top caller.

When the propagation reaches the top caller, it checks what kind of variable is reached. A formal variable is not propagated because the formal variable is only used by

the top caller as a local variable. If it reaches a non-formal variable, then the propagation should reach this variable as well. The propagate is done by linking the vp to the variable vf_vp. If this linking changes the taint value of vf_vp, then a reset of $modified\_number$ and $visit\_number$ for everything involved is required. As the $Propagate\_Upward$ is recursively called, eventually, all routines that have variables associate with the vp are updated.

All variables involved are linked to upper callers. If a taint value has changed, all the routines involved mark their last modified number as current visit number.

Example

In Listing 4.5, a sample source code is shown. We show now an example of performing taint dataflow analysis on this sample in a working thread.

Listing 4.5: Sample Code

```
void main() {
   char buf[100];
   input (buf, sizeof(buf));
}

static void input(char *x, int len) {
   fgets(x, len, stdin);
}
```

After per analysis of this sample code, the routine information is listed below in Table 4.1 and Table 4.2.

Routine	Main
Variables	v1.1 **buf** – vp1.1 **buf_vp**
Operations	op1.1 call: **input** arg.1.1.1: **buf** – vp1.2 – vp1.1 arg.1.1.2: **100** – vp1.3

Table 4.1: Routine Main

Routine	input
Formals	f2.1 **X** – vp2.1 **X** f2.2 **len** – vp2.2 **len**
Operations	op2.1 call: **fgets** arg.2.1.1: **x** – vp2.3 – vp2.1 arg.2.1.2: **len** – vp2.4 – vp2.2 arg.2.1.3: **stdin** – vp2.5

Table 4.2: Routine input

The taint dataflow analysis of this sample code is shown below in Table 4.3.

Step	Action (routine's time stamp)
1	**global_visit_number := 1** Process Routine **main (1)** Routine **main**.visit_number := 1
2	**global_visit_number := 2** process op1.1 Call: **input** link vp2.1 – vp1.2 – vp1.1 for **buf** link vp2.2 – vp1.3
3	Process Routine **input (2)** **push** Routine **input (2)** on the callstack Routine **input**.visit_number := 2
4	**global_visit_number := 3** process op2.1 Call: **fgets** Propagate Upward: vp2.3 – vp2.1 – vp1.2 – vp1.1 Routine **input**.visit_number(2) != starting_visit_number(3): **Re-visit**
5	starting_visit_number = global_visit_number (3) **input**._visit_number = global_visit_number (3) **global_visit_number := 4** process op2.1 Call: **fgets** op2.1 Call: modified_visit_number (3) $\leq$ visited_visit_number (3): **No re-visit**
6	**pop** Routine **input (2)** off the callstack
7	unlink vp2.1 – vp1.2 – vp1.1 unlink vp2.2 – vp1.3 **global_visit_number := 5**
8	**main**.visit_number(3) = starting_visit_number (1), we loop **main**._visit_number = global_visit_number (3) **global_visit_number := 6** op1.1 Call: modified_visit_number (3) $\leq$ visit_number (5): **No re-visit**
9	**main**.visit_number(3) == starting_visit_number (3), finished

Table 4.3: Routine input

4.5.3 Exploration Phase

We have shown before that the propagation happens on the call graph simultaneously with different threads. Multiple threads can perform the analysis either on the same graph or their own copy of the graphs. The first approach is called *shared memory* and the second one is called *duplicated memory*. The steps of concurrent taint dataflow analysis are the same and only the use of data (call graph, variables, routines and ValueProperties) is different in these two approaches.

Shared Memory

The shared memory approach starts by assigning root routines to threads. However, instead of splitting the call graph into several sub-graphs, the analysis in each thread is performed on the same call graph. Figure 4.7 shows this concept.

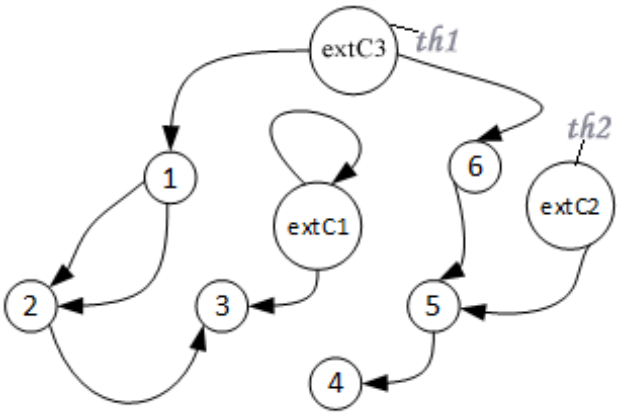

Figure 4.7: Call Graph

In the example of Figure 4.7, it is assumed that there are two threads, called th_1 and th_2. The root routine $extC3$ is assigned to th_1, and the root routine $extC2$ is assigned to th_2. These two threads perform analysis simultaneously on the same call graph.

As the example shows, when th_1 and th_2 visit the same node 5, they both need to link some $ValueProperties$ to node 5's $ValueProperties$. As the analysis is performed

on the same call graph, if those $ValueProperties$ of node 5 are linked to both threads' $ValueProperties$, when a $ValueProperties$ changes in th_1, it will also have an effect on the linked $ValueProperties$ in th_2. To avoid this scenario, a *lock* can be applied.

The lock is set to ensure that only one thread can access the data of a given node. When a thread wants to visit a node, it checks if the node is locked. If the node is not locked, the thread locks the node. This prevents other threads from reading or writing to the node. By utilizing locks, when multiple threads visit a node, only one of them can gain access to the node, and other threads wait until the lock is released.

While a thread locks a node, other threads continue to perform the analysis. A lock list is maintained by each thread. This list stores the nodes locked by the current thread. After finishing analysis, the current thread releases all its locks.

However, this approach is not applicable in many situations. For instance, the shared concurrent taint dataflow analysis may fail to achieve multi-threading when the program to be analyzed is a web application. The reason is that web applications usually haves a database function to connect to the database. If we build the call graph, almost all the paths go through the database connection procedure. By applying the shared concurrent taint dataflow analysis, only one thread can perform the analysis because all other threads are blocked by the lock setting to this procedure. Figure 4.8 shows an example. In this example, if one thread has blocked the node 3, then all other threads have to wait until the node 3 is released. This makes only one thread perform the analysis.

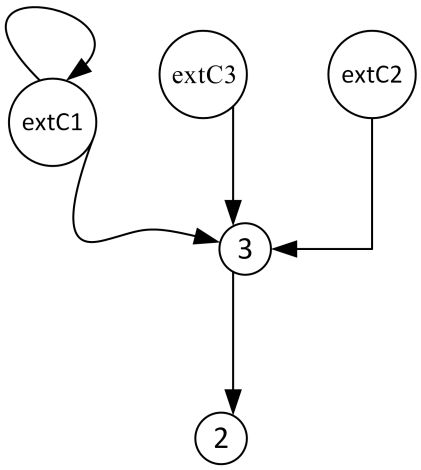

Figure 4.8: Single Node Block

Duplicated Memory

To address the issue of blocked by one node and deadlock, the duplicated memory approach is taken into consideration. To avoid using locks during the analysis, each working thread works on its own copy of the graph. An example is shown below in 4.1 and 4.2.

We assume that we have three threads: th_1, th_2 and th_3. The root routine $extC1$ is assigned to th_1 with a copy of the sub-graph of $extC1$. The root routine $extC2$ is assigned to th_2 with a copy of the sub-graph of $extC2$. The root routine $main$ is assigned to th_3 with a copy of the sub-graph of $main$.

These three threads perform the analysis simultaneously on their own sub-graphs. As the example shows, when th_2 and th_3 visit the same node 5, two different copies of node 5 are used in different threads. With duplication of memory, there is no need to take mutual exclusion into consideration. The propagating phase is done independently in each thread.

After the analysis in each thread is done, each thread contains its own result. A merging phase puts the result from each thread together to produce the global result of

the taint dataflow analysis.

4.5.4 Merging Phase

In this thesis, we chose the duplicated memory approach to achieve multithreading. However, for all threads, when they finish the analysis, each thread has only the result of its own sub-graph. It is necessary to merge the results so that the concurrent taint dataflow analysis provides exactly the same result as the sequential taint dataflow analysis does.

When we duplicated memory, one node on the *callGraph* may have two different instances on two different sub-graph. Figure 4.2 shows that the node 3 has two different instances on sub-graph rooted by $extC1$ and sub-graph rooted by $extC3$. If there is a variable *var* used in node (routine) 3, two different instances of *var* are created.

When a thread analyzes the sub-graph rooted by $extC1$, the taint trace is appended to the $ValueProperties$ of *var*. Similarly, another thread inserts another taint trace of *var* on $extC3$. At the end of analysis, as *var* is independent in two threads, the analysis is not able to know that *var* has two taint trace. The merging phase is needed to perform this work.

A thread goes back to its own sub-graph to find all the variables involved after finishing analyzing a sub-graph. This thread then starts to merge the taint information of all involved variables back to the original call graph (it is a union operation). The taint trace information of each variable is appended to the variables' $ValueProperties$ on the original call graph. The details of the algorithm of merging the result is shown below in Listing 4.6.

Listing 4.6: Algorithm of Merging Result

```
Procedure MergeResult()
```

```
for each variable var used in this analysis
  //find the ValueProperties of 'var' from the original call graph
  ValueProperties vp = callGraph.node[var].ValueProperties
  // ask and wait until we can read
  vp->acquireReadLock();
  //read again to make sure we have the latest information of this
     ValueProperties
  vp = callGraph.node[var].ValueProperties
  //check if the trace is all ready inserted by other thread
  if (vp->traceInfo->find(var->ValueProperties->traceInfo))
     vp->releaseReadLock();
     Continue;
  else
     //wait until we can write
     vp->acquireWriteLock();
     vp->traceInfo->append(var->ValueProperties->traceInfo);
     //release the locker
     vp->releaseWriteLock();
```

Each variable analyzed by this thread is checked iteratively. The merging of results consists of 4 steps:

1. Find the reference of the $ValueProperties$ of a analyzed variable var on the original call graph.

2. Check if there is another thread merging result to this $ValueProperties$. This is done by using the reader/writer lock. When $aquireReadLock()$ is called, the working thread waits until no writer lock is set to this $ValueProperties$. Then the working thread gains access to this $ValueProperties$. The working thread sets a

reader lock to this $ValueProperties$ to prevent other working thread writing.

3. Checks if the same taint trace of *var* has already been merged into this $ValueProperties$. If so, there is no need to insert the same trace again. The reader locker of this *vp* is released, and this thread continues processing the next analyzed variable. If the taint trace of *var* does not exist in *vp*, the merging starts.

4. A writer lock is set to *vp*. After that, this thread appends the taint trace of *var* to *vp*'s trace field. The writer lock of *vp* is released after the appending finished and this thread continues to process the next used variable.

After all threads finish merging, all variables on the original call graph will have the final results.

4.6 Conclusion

In this chapter, we have shown that taint dataflow analysis can be solved concurrently by traversing the call graph and propagating the taint information simultaneously. Algorithms for constructing the call graph, propagating taint information among routines and propagating taint information within a routine are studied. Two approaches of sharing information between threads are introduced. Then we explained why duplicated memory approach takes better advantages of the multithreading in our study.

Chapter 5

Implementation and Performance

In this chapter we describe in detail an implementation of concurrent taint dataflow analysis. We also provide experimental results to show the performance improvement of our approach. The chapter is organized as follows:

- In Section 5.1, we provide some background of our prototype which is a sequential program for taint data flow analysis. We also describe how we changed this program to integrate our approach to concurrency.
- In Section 5.3, we provide the methodology we are using during the implementation. Two methods are introduced to help implement multiple threading and memory sharing.
- In Section 5.4, we describes the difficulties encountered during our implementation. We also explain why we are not able to deal with some of these difficulties.
- In Section 5.5, we provide the experiment of two test cases. We show the performance improvement and explain the result as well.

5.1 IBM AppScan Source

IBM Security AppScan Source [18] is a tool written in **C/C++** to identify and to help dealing with the vulnerabilities in web, desktop and mobile applications. This tool is integrated into the developing process of the lifecycle by analyzing the source code of the software. It provides a cross platform and language independent analysis for the source code. The tool is also able to provide the trace information of the vulnerabilities and the type of threat. The analysis is done by using the taint dataflow analysis which tracks user's input and validates whether there is a risk of using it. Our prototype is implemented based on IBM Security AppScan Source [18] and intends to improve the performance of the analysis.

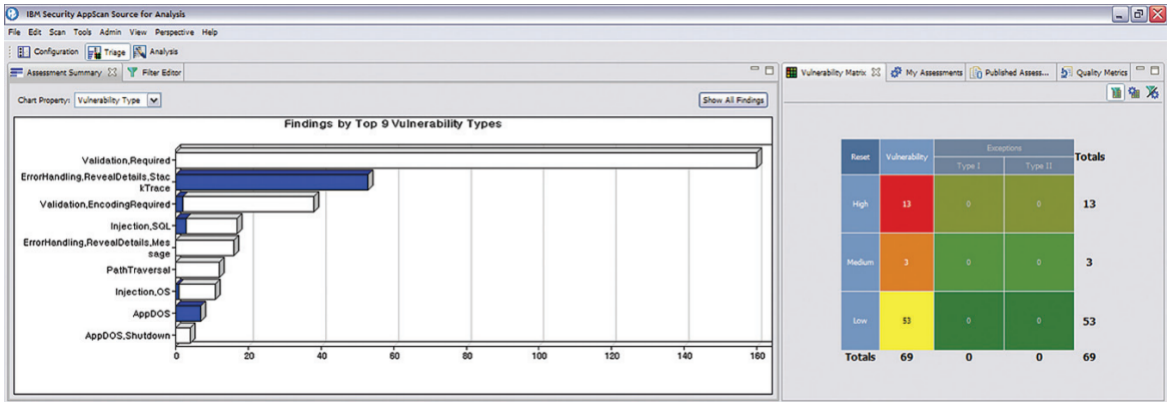

Figure 5.1: Assessment Summary of AppScan

5.1.1 Architecture

To perform the analysis, there are four stages involved in our concurrent dataflow analysis prototype. The architecture is shown in Figure 5.2.

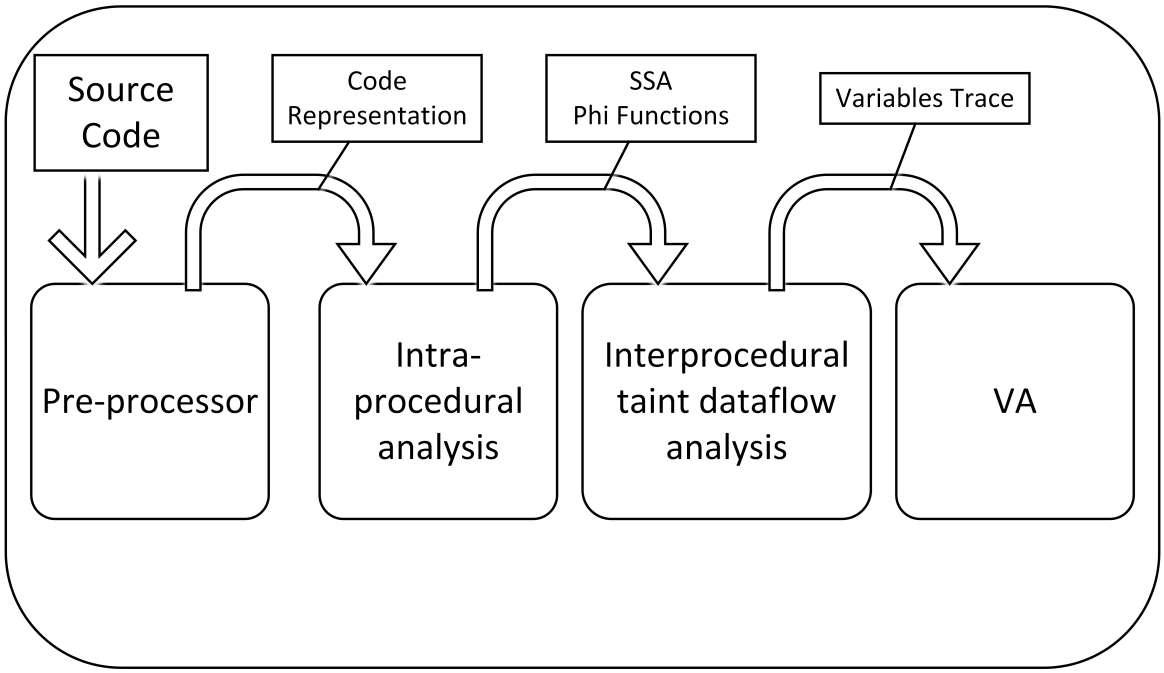

Figure 5.2: Components of the Analysis Engine

In ***Pre-processor***, the source codes are translated into intermediate representation. This stage provides language independence. After the intermediate representation is generated, all function procedures are examined to construct the data structures for all routines.

After the routine information is constructed, the ***Intraprocedural Analysis*** is performed. The intraprocedural analysis is a combination of control flow analysis and data flow analysis. The control flow analysis builds control flow graph for each routine which indicates the order of execution of statements in a procure. The dataflow analysis is performed by transforming all functions to static single assignment(SSA) [12] form. In SSA form, variables are renamed and **phi** functions are applied. When the value of a variable can be assigned by more than one source, and it is impossible to tell at the compile time which source has assigned that variable, the dataflow analysis creates a artificial definition of that variable named *phi* to indicate that the variable can be assigned by

different sources. An example of SSA form transformation is showing below in Figure 5.3.

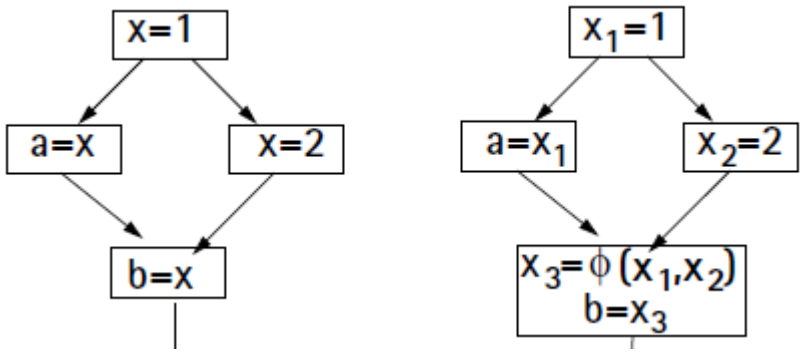

Figure 5.3: SSA

After that, the completed data structure of each routine is generated. Listing 5.1 shows definition of completed routine representation.

Listing 5.1: Definition of Routine Representation

```
class Routine
{
   String name;
   int visit_number;
   Variable localVariables[];
   Variable formalsVariables[];
   Routine callers[]
   Operation op[];
}

class Variable
{
   String name;
```

```
   ValueProperties vp;
   Routine routine;
}

class ValueProperties
{
   Routine routine;
   int visit_number;
   int modified_visit_number;

   ValueProperties arguments;
   Operation op;
   ValueProperties formal;
   ValueProperties variable;
}

class Operation
{
   boolean isCall;
   boolean isReturn;
   boolean isAssign;

   boolean isNotVisited;

   Routine routine;
   int modified_visit_number;
   int visited_number;
```

```
}
```

- **Routine**: This object contains the *name* of a procedure, last *visit_number*, a list of all local variables $localVariables$, a list of all formals $formalVariables$, a list of callers of this procedure *callers* and a list of *op* representing all statements in this procedure.

- **Variable**: This object contains the *name* of a variable, associated $ValueProperties$ and a reference to its *routine*.

- **ValueProperties**: This object contains a reference to its *routine*, last *visit_number*, last $modified\_visit\_number$ which indicates the visit number of the last tainted value changing and three reference to its linked variables. If its linked variable is an argument, then *arguments* is a reference to that variable and *op* is a reference to the function call. If its linked variable is a formal variable, then $formal$ is the reference to the formal variable. If its linked variable is not a formal or an argument, then *variable* is the reference to that variable.

- **Operation**: This object contains three flags: $isCall$, $isReturn$ and $isAssign$ which indicate the type of this operation. Another flag $isNotVisited$ is used to mark whether this operation has never been analyzed. A reference to this operation's routine *routine* is also stored. $modified\_visit\_number$ and *visited_number* are used to indicate when this operation changes some tainted value and when this operation is analyzed.

The third stage uses the completed routine data structure to perform the ***interprocedural taint dataflow analysis***. The interprocedural taint dataflow analysis first builds a call graph and starts analyzing from each root node of the call graph. A tainting trace is generated for each tainted variable.

The last stage in our prototype is to use the tainting traces of variables to perform the **Vulnerability Analysis (VA)**. The VA tells whether a user's input is going to cause security issues. For instance, using a user's input directly in a SQL query is considered as insecure because a potential SQL injection can be launched.

5.1.2 Original Flow

Our approach focuses on the Interprocedural Taint Dataflow Analysis (ITDA) stage of the prototype. The ITDA process consists of two phases. The first phase is to build the call graph. The second phase is to iterate over the call graph from each root node and to propagate the taint information while visiting the nodes on the graph.

Our concurrent analysis is introduced into the second phase. The original implementation of the second phase is using a single thread and analyze each sub-graph (rooted by a root node) at a time. The work flow of the original implementation is shown below in Figure 5.4.

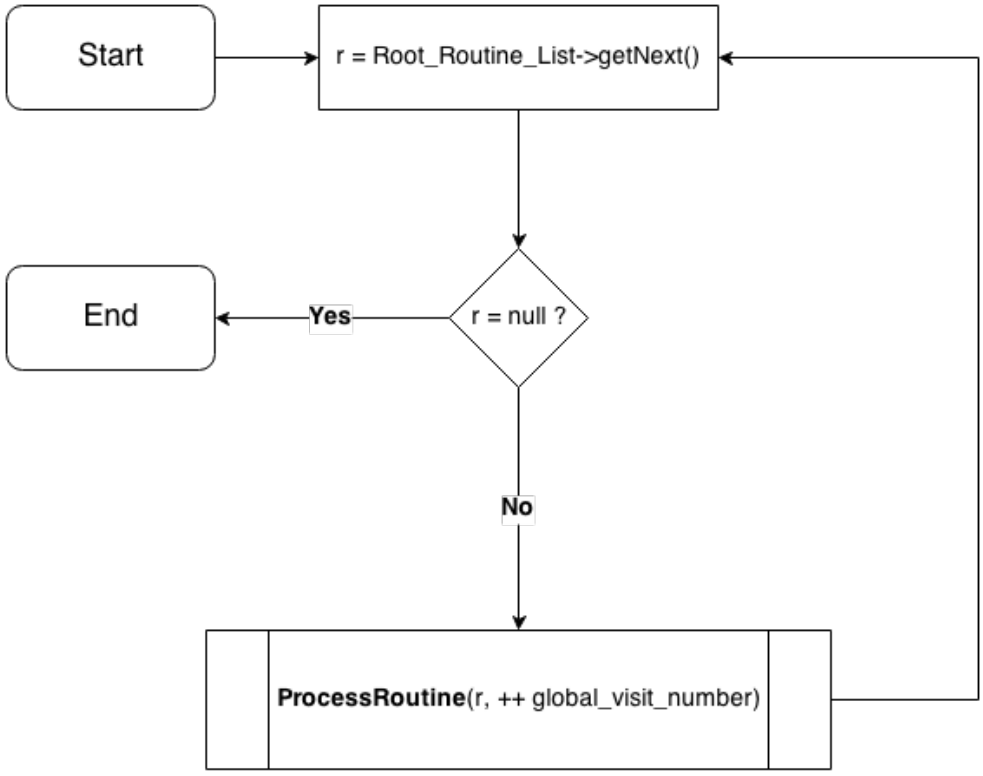

Figure 5.4: Flow chart of Original Implementation

The original work flow starts with getting a root routine r from the *Root_Routine_List*. The *Root_Routine_List* is a list contains all root nodes on the call graph. Then it takes root routine r as an entry point of analysis. The routine r and its rooted sub-graph are analyzed. After this analysis is finished, it continues to analysis the next root routine until all the root routines and their sub-graphs are analyzed.

5.1.3 Concurrent Flow

As discussed in Chapter 4, instead of analyzing each sub-graph one by one, a concurrent analysis creates multiple threads to analyze sub-graphs at the same time. Applying the idea, we modify the original flow as shown in Figure 5.5 below.

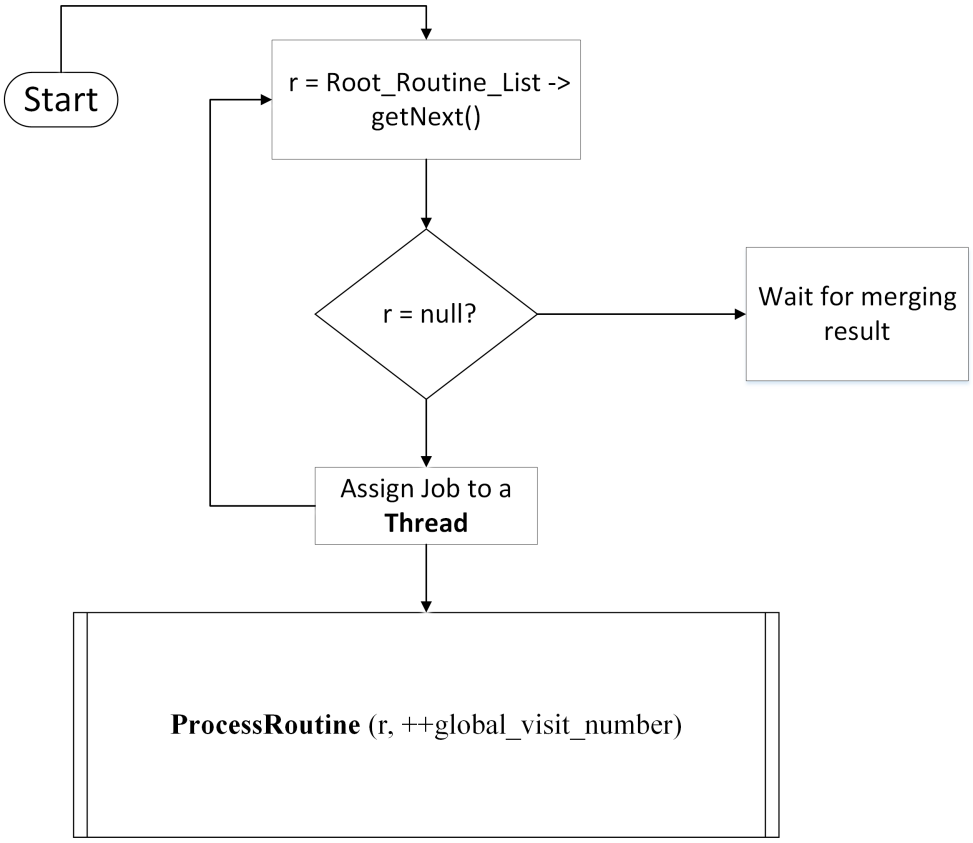

Figure 5.5: Flow chart of Concurrent Implementation

The concurrency is achieved by multiple threading. In our approach, there are two kinds of threads.

- **Master Thread**: The master thread is responsible for getting the next root routine from the root list and create a copy the associated sub-graph. Then the master thread assigns the task to a working thread. The master thread goes back to create a new job and assigns it to a new working thread until all root routines and their sub-graphs are handled by working thread.

- **Working Thread**: The working threads start analyzing the given job independently. The analysis flow of each working thread remains the same. After a working thread finishes a analysis, it merges its result back to the global result.

5.2 Implementation

The implementation is based on the algorithm described in Chapter 4. The implementation is separated into two tasks. In Section 5.2.1, a detailed implementation of the master thread is provided. In Section 5.2.2, details of performing the analysis and solving race conditions are discussed.

5.2.1 Master Thread

The master creates jobs for each working thread. It also manages shared resources, synchronization of the work threads, communication among the work threads and takes the merged result.

The master thread is created at the beginning of the ITDA stage. It initializes all the resources needed for the analysis. The initialization includes creating a thread pool with some working threads. After all necessary resources are prepared, the master thread goes to the iteration step and works as a thread manager. The iteration step runs within a loop. In the loop, the master thread gets the next root routine from the root list and copies the associated sub-graph from the call graph. Then it finds an idle working thread in the thread pool and assigns the job (root routine and its sub-graph) to the working thread to perform the analysis of the routine.

Thread Pool

Multiple threads running concurrently can improve the performance. However, it does not mean that one can create unlimited threads to improve the speed. Too many threads will led to a performance degradation. There are three reasons that the performance can be effected negatively by too many threads:

- **System Limitation**: Modern computers are not able to support too many threads

running concurrently. For example an i7 dual core may only support up to 4 threads running concurrently. That means even there are more threads being created only 4 of them can run simultaneously.

- **Thread Switching**: If there are too many threads running, thread switching happens more frequently. Thread switching is computationally intensive. To switch between threads, a certain amount of time is required for saving and loading registers and memory [39]. With too many threads the switching consumes more time than the time saved by multiple threads.

- **Thread Initialization**: When a thread is created, the system needs to initialize the stack, heap, copy global variables and initialize the local variables. To create too many threads, the initialization of threads takes a certain amount of time.

Therefore, the master thread should be able to control the number of threads running in the system and to avoid creating threads frequently. In our implementation, a thread pool is used to address these issues.

The thread pool is created by the master thread with a certain number of threads at the beginning. Each thread is called a working thread. A job, which is an infinite loop, is assigned to each working thread. This job is simulated as an idle job. The master thread works as a manager of the thread pool. When a working thread is required, the master thread gets an idle working thread and replaces the idle job with a actual job. After the working thread finishes its job, the master thread assigns the idle job again to that working thread.

In this case, we can maintain the number of threads running in our system while creating them only once, thus to achieve the best performance of multiple threading.

Job Distribution

After the thread pool is created, the master thread starts to create and assign jobs to working threads. Before a job is assigned, the master thread goes to the thread pool to find a working thread running an idle job. If there exist an idle working threads, a job is created and assigned to this idle working thread. If there is no idle working thread, the master thread waits until an working thread finishes its work and becomes idle.

Listing 5.2 shows the implementation of job distribution.

Listing 5.2: Algorithm of Master Thread

```
Procedure Master_Thread(Routine[] root_list)
  for each root in root_list
       if isExceededMemoryLimition()
       MemoryManager->cleanUp();
       while(true)
          threadId := threadManager.aquireThreadIndex();
          //calling a win32 API, wait for a working thread's release event
          if (threadId == -1)
             WaitForSingleObject(wReleseEvent, INFINITE);
             continue;
          else
             break;

       // build the sub-graph which is rooted by this root routine
       subGraph = buildSubGraphRootedBy(root);
       getThreadById(threadId).threadRun(subGraph, root);

  While(threadManager.haveJobs())
  End while
```

There are 6 steps to distribute a job:

- **Step 1**: The master thread iteratively gets an unprocessed root routine *root* from the root list. If all roots are processed, then go to **Step 6**.
- **Step 2**: The master thread checks the memory space to ensure there is enough memory space for this analysis. A *cleanup* function is called to make more space for this job if necessary. The *cleanup* function simply copies some random data in memory to disk. The reason is that the duplicated approach requires large amount of memory space to duplicate information for each working thread.
- **Step 3**: The master thread enters the spinlock to acquire an idle working thread. The spinlock is a loop which makes the master thread keep trying to get a working thread until it succeeds. A non negative $threadId$ is returned to represent an idle working thread and a negative $threadId$ means that there is no idle working thread at the moment.
- **Step 4**: The master thread constructs the sub-graph *subGraph* which is rooted by *root*.
- **Step 5**: The master thread assigns the *subGraph* and *root* to a working thread with id equals $threadId$. After that, the working thread starts to perform the analysis and the master thread goes back to **Step 1** to create the next job.
- **Step 6**: The master thread terminates when all working threads are finished. When a working thread finishes its task, it modifies a idle counter in the thread manager. If the value of idle counter equals the number of working thread, then that means all working threads are finished. In other words, the analysis of a program is finished.

5.2.2 Working Thread

The working thread is in charge of performing the analysis. The task is assigned by the master thread. The task contains the sub-graph and the root of the sub-graph. A number is also given as the index of the *resource container*.

Resource Container

The resource container is shared among all working threads and the master thread as discussed above. A resource container is created by the master thread. This is intended to provide the ability of communication between threads. The container is implemented as a vector. The number of the elements in the vector is the same as the number of the working threads. There is a one-to-one relationship between the container and the working thread.

Instead of copying the data into a working thread's local memory, the master thread copies data to the container. An index is assigned to each working thread, when a working thread need to access its data, it uses the index to access the element of the container. The index is the same as $threadId$.

Perform Analysis

The analysis starts from the root routine of the sub-graph. The root routine is different from other routines because the root routine doesn't have an actual caller. Therefore, for the root routine a process called "generate the external context" is called before the root routine is analyzed. The "external context" is generated such that the root routine can be analyzed as a common routine. In other words, the root routine will also have some arguments passed.

When the analysis starts, the routine is pushed onto a call stack which is used to deal with the recursions as discussed in Chapter 4. For each routine to be analyzed,

its last modified visit number is used as a new starting visit number to indicate a new visit. Figure 5.3 and Figure 5.4 show the detailed algorithms of the working thread. The algorithm contains two parts: process a root and process a routine.

Listing 5.3: Algorithm of Processing Root Routine

```
Procedure Process_root(Routine root, int ThreadId)
   //no need to analyze if there is no operations(statements) in this root
       routine
   if (root->numberOfOperations ==0)
      return
   //generate the external call context
   root->generateExternalCallContext()
   //set the start visit number to the root's last modified visit number
   resetStartingVisitNumber(threadId)
   //clear the call stack
   clearCallStack(threadId)
   //process the root routine as a common routine
   Process_Routine(root, incrGlobalVisitNumber(), threadId)
   //merge the result
   MergeResult()
   //clear the external context
   root->clearExternalCallingContext()
```

To process a root routine, there are 7 steps:

- **Step 1**: Check if there is any operations in this routine. If there are no operations, then there is no need to perform the analysis of this routine because no taint information will be changed by calling this routine in the program.
- **Step 2**: If this root routine has some operations, the external context is generated

such that the working thread can process this root routine as it is a common routine.

- **Step 3**: The start visit number is set to this routine's last modified visit number. This resets the starting visit number of this routine. The *threadId* is used so that the working thread knows which element in the container should be used.
- **Step 4**: The call stack is cleared to provide a fresh environment for the current analysis.
- **Step 5**: This root routine is analyzed as a common routine.
- **Step 6**: The result is merged after this sub-graph is analyzed.
- **Step 7**: The external context is cleared.

Listing 5.4: Algorithm of Processing Routine

```
Procedure Process_Routine(Routine r, VisitNumber visit_number, int threadId)
  pushCall(r, visit_number, threadId)
  do
    for each Operation op in r
      incrGlobalVisitNumber()
      if (op->lastModifiedVisitNumber > op->lastVisitedNumber)
        Switch(op->kind)
          Case simple_call:
            called_Routine = op->getCalledRoutine()
            call = op->getCall()
            //recursive call to process the called
            //routine, the following function links
            //necessary arguments and formals
            //and then analysis the call as a routine
```

```
      Process_Call(call, called_Routine, threadId)
      break
    Case virtual_call:
      //process all possible actuall calls as
      //Process_Call()
      Process_Virtual_Call(op->getVirtualCall(),threadId)
      break
    Case return:
      //merge the return values tainted
      //information into the formals
      //tainted information
      op->mergeReturnToFormal()
      //if the result is a reference we
      //need to merge the referenced object
      //as well
      if op->return_Val->isReference()
        op->mergeReferencedToFormal()
      break;
    Case assignment:
      op->dst->taintInformation = op->src->taintInformation
      if ( op->dst->isLocalVariable() || op->dst->isFormal)
        //as this tainted information
        //will not propagate upward to a
        //higher routine, we mark it to
        //restore after this routine is
        //analyzed
        Op->dst->markToRestoreTaint(r)
op->setOperationLastVisitedNumber()
```

```
while(r->lastModifiedNumber != starting_visit_number)
popCall(r, threadId)
```

To perform the analysis of each routine, the routine is pushed into the call stack at the beginning. Each operation is analyzed by applying the TT and TA functions discussed in Chapter 3. The routine is analyzed recursively until nothing is changed. This is indicated by the last modified number of this routine. Whenever any taint information changes during a analysis of this routine, the last modified number will be different from the start visit number. After the analysis of this routine is finished, this routine is popped off the call stack.

5.3 Methodology

This implementation is based on the prototype which has already implemented the taint dataflow analysis with a single thread in C++. The taint dataflow analysis is an intermediate component of the software. To transform the existing sequential program to a concurrent program some methodologies that we used are explained in this section.

5.3.1 Moving Locks

The first methodology to introduce concurrency into the sequential program is called *moving locks*. Moving locks enable the program to be partially concurrent and by moving the locks we eventually make the whole dataflow analysis work concurrently.

To transform a sequential program into a concurrent program, the basic idea is to find an entry point and duplicate all the calls after that point in multiple threads. For instance, a sequential program which deletes all files in a folder can be simply written as:

Listing 5.5: Sample Code: Sequential Delete Files

```
Procedure deleteFiles(Folder fo)
  For each File f in fo
    Delete(f)
  End For
```

To transform the example above to a concurrent program, we simply duplicate the delete() function in multiple threads as shown in Listing 5.6.

Listing 5.6: Sample Code: Concurrent Delete Files

```
Procedure deleteFiles(Folder fo)
  For each File f in fo
    thread.run(Delete(f))
  End For
```

Listing 5.7 shows a more complex example.

Listing 5.7: Sample Code: Delete Files

```
Procedure deleteFiles(Folder fo)
  For each File f in fo
    Delete(f)
  End For


Procedure delete(File f)
  Switch f->type
    Case image:
      deleteImage(f)
    Case document:
      deleteDoc(f)
```

```
    Case video:
      deleteVideo(f)
  End Switch
```

When we duplicate the *delete*() function, *deleteImage*(), *deleteDoc*() and *deleteVideo*() are also duplicated. In other words, all functions called by the duplicated function are duplicated as well. In our implementation, there are several functions that are called by the duplicated function. To be able to validate each of them, moving locks is introduced.

The moving lock is implemented using *EnterCriticalSection*() and *LeaveCriticalSection*(). The *EnterCriticalSection*() and *LeaveCriticalSection*() make sure that only one thread runs the code between them. An example is shown below using the code in Listing 5.7.

We assume there are two threads $th1$ and $th2$. The *delete*() is duplicated. So $th1$ and $th2$ have the same code:

Listing 5.8: Sample Code: Delete Files In Each Thread

```
Procedure delete(File f)
  Switch f->type
    Case image:
      deleteImage(f)
    Case document:
      deleteDoc(f)
    Case video:
      deleteVideo(f)
```

Apply the moving locks, we have:

Listing 5.9: Sample Code: Delete Files With Moving Locks 1

```
Procedure delete(File f)
  EnterCriticalSection()
```

```
  Switch f->type
    Case image:
      deleteImage(f)
    Case document:
      deleteDoc(f)
    Case video:
      deleteVideo(f)
  LeaveCriticalSection()
```

In this case, when $th1$ and $th2$ are running, only one of them can run the "switch" statement. By moving the locks down, we have:

Listing 5.10: Sample Code: Delete Files With Moving Locks 2

```
Procedure delete(File f)
  Switch f->type
    Case image:
      deleteImage(f)
  EnterCriticalSection()
    Case document:
      deleteDoc(f)
    Case video:
      deleteVideo(f)
  LeaveCriticalSection()
```

In this case, $deleteImage()$ can run concurrently in $th1$ and $th2$. $deleteDoc()$ and $deleteVideo$ are allowed to run in only one thread.

By moving the locks down, we can parallelize the program function by function. Thus, we can validate one function at a time. If there exists an error after moving the

locks by one function, we know where the error happens. This is useful to debug and validate complex concurrent programs.

The moving locks mechanism can also be applied to variables. It also helps us to understand which variables are shared between working threads and may cause a race condition so that we can duplicate these variables in each working thread. When the moving locks finally moves to the $LeaveCriticalSection()$, the program is running fully concurrently.

5.3.2 Shared Container

The original implementation of the prototype is highly optimized to minimize duplicate objects. This is done by using smart pointers to make sure when we call the copy constructor of an object only the counter of that instance increases instead of making a new instance in the memory. When we duplicated the analysis function in multiple working threads, the master thread duplicates all necessary information in each thread independently. To be able to copy an object, there are two options:

1. **Overwrite the Constructor**

   One option is that we can overwrite the constructor of those objects and call the overwritten constructors to copy them in memory.

   This is hard to do due to inheritance. For instance, if we want to modify the copy constructor of *Class* A and A contains some pointers pointing to an object from *Class* B then we have to modify the copy constructor of B as well. Failing to do so will leave the instance of B in A not duplicated.

   When we use the information of the B object in A in different working threads, the instances of A is independent but the information of B remains the same in all working threads.

2. **Shared Container**

   The second option is called shared container. The container is constructed by the master. When the master thread constructs the container, it simply asks for memory space without assigning any information. When a working thread is initializing, the *threadId* is assigned as the identifier to the container to access the memory area that belongs to this working thread. Also, instead of modifying the copy constructors, the master simply copies the whole memory into the working thread's container space.

In our implementation, we chose the shared container to duplicate the memory for each working thread. This is more efficient to implement. To avoid consuming too much memory, we only duplicated the dependency graph and its associated data. All other data used by working threads is not duplicated.

5.4 Challenges

During our implementation, there were several challenges. Some of them prevented us from implementing the existing taint dataflow analysis concurrently. We had no choices but to disable some features of the original implementation.

5.4.1 Memory management

The memory management component is always used in the program. As this product is very memory consuming and to avoid potential memory leaks in C++ with pointers, the memory management is implemented as a fundamental component of the program. Every single instance created in the program, no matter whether it is belongs to the taint dataflow component or not, is controlled by the memory management component.

The memory management monitors the memory usage and always attempts to release memory space. For example, if the memory management finds that the program is reaching the memory limitation it automatically tries to clear the memory.

By using the smart pointer, the memory management knows which instances are not in use. When the memory management does the cleaning, all the instances that have no reference will be released. Additionally, the instances that are not recently used are also swapped to the disk instead of staying in the memory.

The memory management in the taint dataflow analysis checks the memory usage before any function is called. It also tries to clean the memory after any processes are finished. As our duplicated approach still keeps some pointers, when a working thread finishes one process of analysis, some data may be removed by the memory management. To address these issues in our implementation, we first chose to modify the memory management for the taint dataflow analysis.

Unfortunately, all data types except the basic data types are defined in the memory management. In other words, to modify the memory management function for the dataflow analysis turns out to rewrite the memory management component. As our primary goal is to make a concurrent taint dataflow analysis instead of making a concurrent memory management component, we decided to disable the memory management by setting the memory limit to unlimited.

5.4.2 Race Condition

A race condition is an important issue to be solved in concurrent programming. With the moving locks, we can restrict the race condition to occur within a certain scope. Also, by duplicating the memory, we ideally avoid the shared resources altogether. But in practice, during implementation, it is impossible to make everything correct at once. For instance, when we deal with the shared resources, because we want to duplicate only

the needed information, we don't copy all the information. With the moving locks, we can tell which resource should be duplicated to avoid a race condition. If we find some resources facing a race condition, we duplicate them and validate before moving the locks down.

In practice, the error cannot be reproduced easily so that we were not able to find some bugs during the implementation. In other words, when we believed that all the race conditions has been solved before the locks there may be still some bugs hiding. This situation required us to go back to check something we previously thought was correct.

5.4.3 Exception Handling

The original implementation we got was a beta version under development. There are some bugs existing in the code. As an enterprise product, the exception handling is a major concern of the code.

During our implementation, we found that it is very hard to debug our code with the exception handling. The original implementation of exception cleans the memory at the very beginning. In this way, all debug information such as stack information, variable's values, pointers are cleared out in the memory.

To Disable the exception handling is also impossible. Due to the limitation of the program, there are some cases it can not analyze. These cases throw an exception.

5.5 Performance

In this section, we compared the performance of concurrent taint dataflow analysis to the original taint dataflow analysis prototype. The sub-sections below are organized as follows:

- in Section 5.5.1, we provide the details of how we measure the performance and

how we compare the performance between the original program and our approach.

- In Section 5.5.2, we discuss the test cases we are using in our experiment.
- Finally, in Section 5.5.3, we give the result of the comparison and summaries our experiment.

5.5.1 Performance Measurements

The performance of the program is compared according to different criteria. We compare the total cost of the whole analysis and the cost in each thread. We ran each experiment three times and the presented results are the average of the three runs. The aspects are discussed below:

- **Cost of Total Analysis**

  The most important performance is the total time of the analysis. As our goal is to reduce the time to perform the analysis, this cost is our main concern.

  We get the timestamp when the taint dataflow analysis starts and finishes. We calculate the duration of the entire analysis.

- **Average number of processed routines**

  This cost aims to tell how much extra work load is given to the overall analysis.

  We count the number of visiting to routines, including re-visiting, in each working thread. The cost is the sum of the number in each thread.

Our experiment is running on an i7 CPU which has 2 cores and supports 4 threads running concurrently. As we explained before, only up to 4 threads are necessary on this test computer. In other words, we are running with 1 master thread and up to 3 working threads during the experiment.

5.5.2 Test Cases

Altoro Mutual

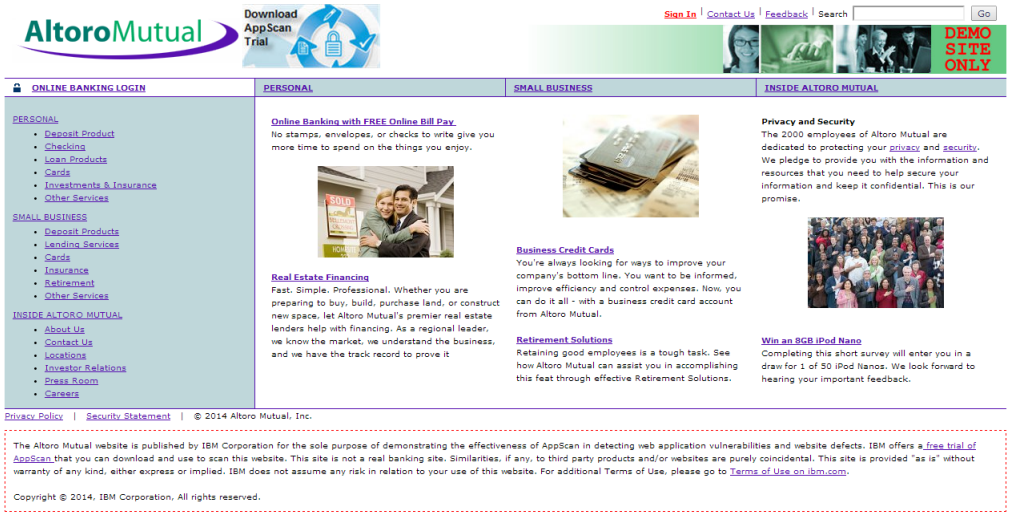

Figure 5.6: Altoro Mutual Website

The first test case we are using is a test application built by IBM AppScan team as a mock website for security testing. This application is a demo of a fictional bank website [17].

This application contains **463** routines.

WebGoat

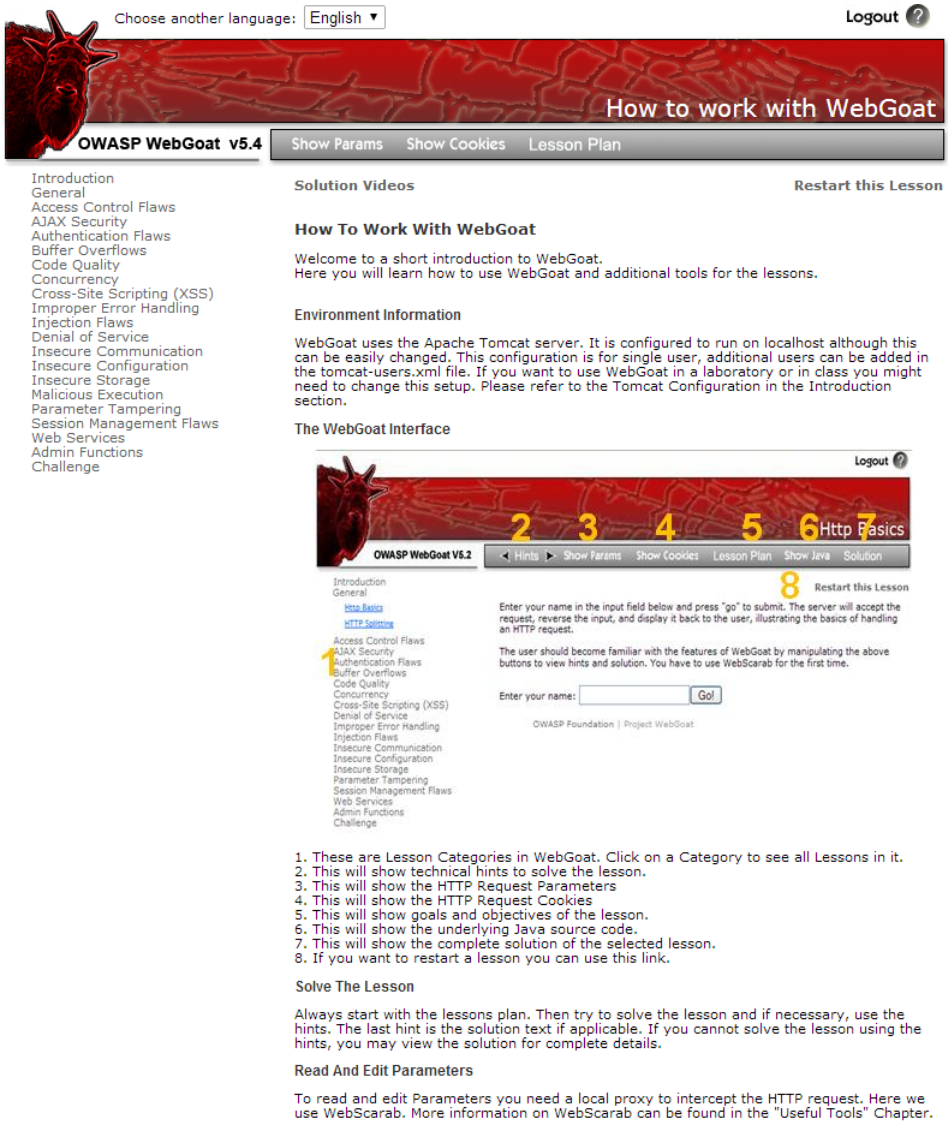

Figure 5.7: WebGoat Website

The second application is a real application called WebGoat [24]. This application is written in Java and aims to teach web application security lessons.

This application contains **12,219** routines.

5.5.3 Results

Altoro Mutual

- **Cost of Total Analysis**

  The total analysis time cost is shown in Table 5.1 and Figure 5.8.

  With only one working thread running, the total time of the analysis is 6.61 seconds. The time goes down to 3.95 seconds when we run the experiment with two working threads. The time decreases a little when we increase the number of working threads to three.

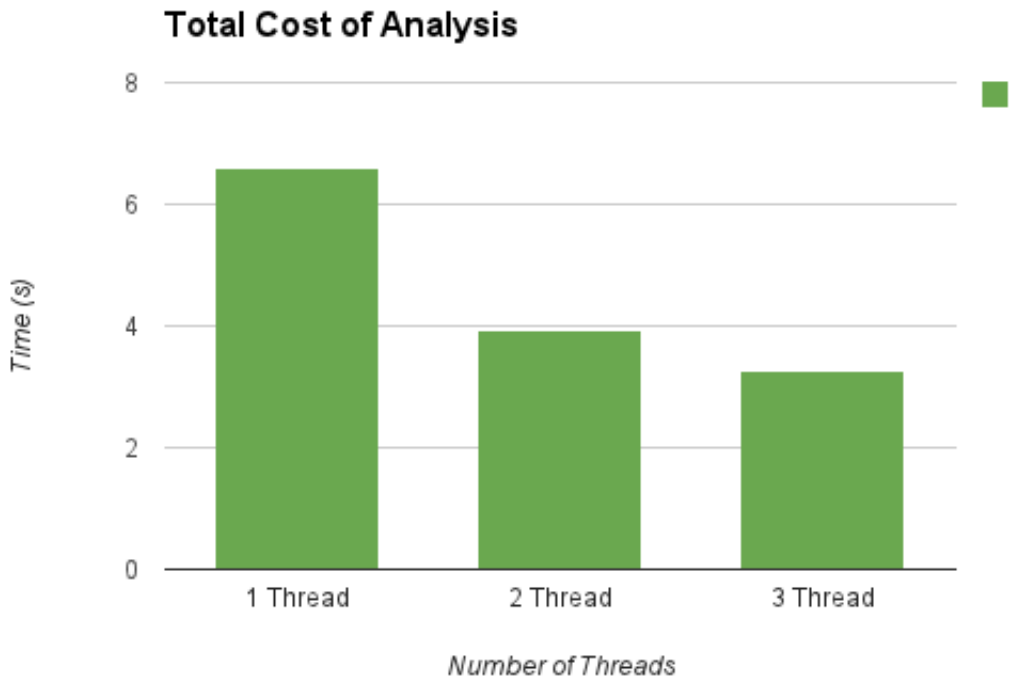

Figure 5.8: Total Cost of Analysis (Altoro Mutual)

The variances are 0.0049, 0.0024 and 0.0013 for one thread, two threads and three threads, respectively. The confidence interval is difficult to show in the graph. For

one thread, the 90% confidence interval is [6.56,6.65]. For two threads and three threads, they are [3.91,3.98] and [3.24,3.29], respectively.

Number of Threads	Total time cost in each thread
1 Thread	6.61
2 Thread	3.95
3 Thread	3.27

Table 5.1: Total cost of analysis(Altoro Mutual)

The reason is that as this test case is small, the master thread becomes the bottleneck. The working thread finishes its analysis faster than the master thread prepares new tasks. In other words, most of the time, there are only two working threads running concurrently and the third one is waiting for the master thread to assign a new task.

- **Average number of processed routines**

  The result of average number of processed routines is shown in Table 5.2 and Figure 5.9.

  As we can see, the average number of processed routines increases a little when the number of working threads increases.

  In the sequential approach, some routines and its sub routines may not need to be re-visited because the previous analysis has already provided the result. This is called result reuse. However, in a concurrent approach, the result may not be able to be reused. In concurrent taint dataflow analysis, two working threads may analyze the same routine at the same time. Neither of them is aware of the other's result. This may cause unnecessary duplicate analysis of same routines and increase

the average number of processed routines. This is a side effect of our concurrent taint dataflow algorithm.

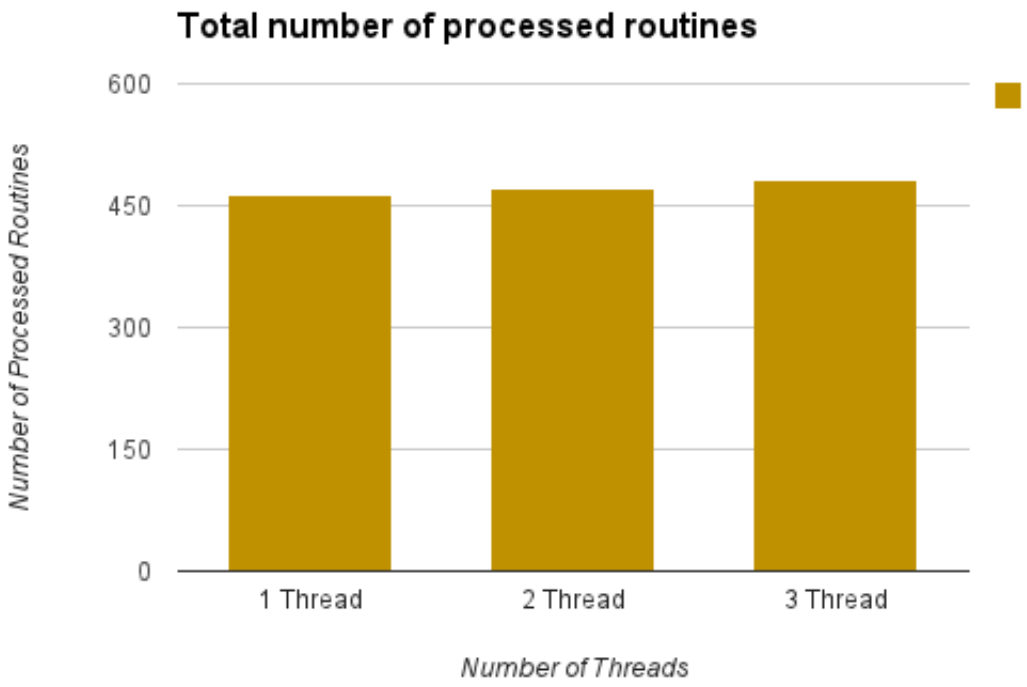

Figure 5.9: Average number of processed routines (Altoro Mutual)

Number of Threads	Total Number of Processed Routines
1 Thread	463
2 Thread	470.67
3 Thread	480.67

Table 5.2: Average Number of Processed Routines(Altoro Mutual)

WebGoat

- **Cost of Total Analysis**

The total analysis time cost is shown in Table 5.3 and Figure 5.10.

With only one working thread running, the total time cost of the analysis is 214.2 seconds. The time cost goes down to 152.6 seconds when we run the experiment with two working threads. The time cost decrease to 117.7 seconds when we increase the number of working threads to three.

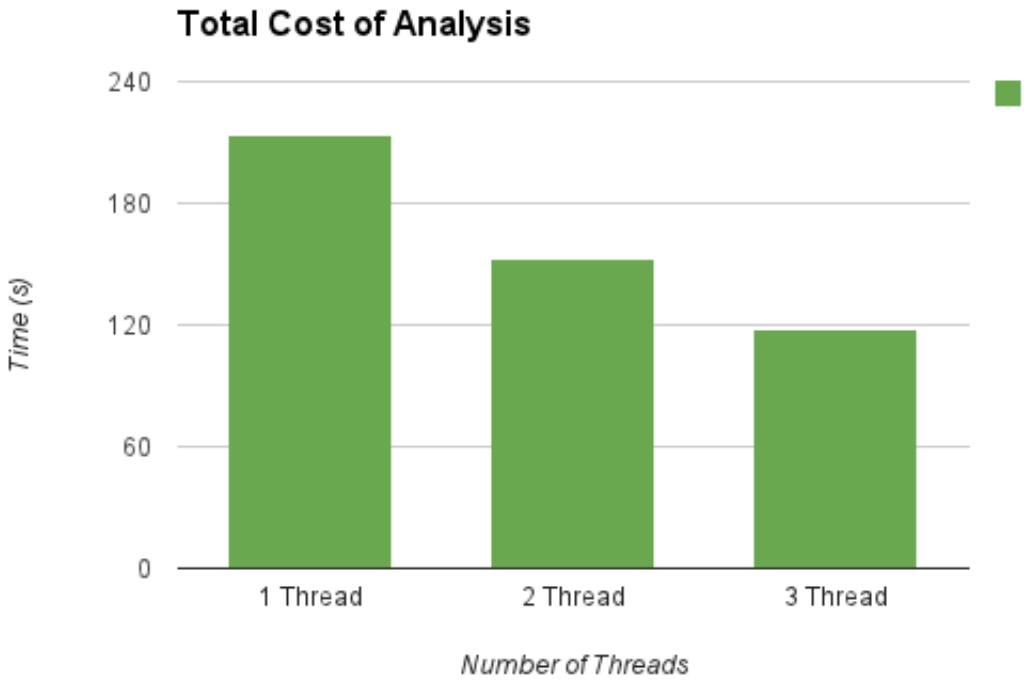

Figure 5.10: Total Cost of Analysis (Web Goat)

The variances are 10.66, 4.69 and 5.95 for one thread, two threads and three threads, respectively. The confidence interval is difficult to show in the graph. For one thread, the 90% confidence interval is [212.09,216.28]. For two threads and three threads, they are [151.23,154.00] and [116.16,119.29], respectively.

The result also shows that the master thread is not a bottleneck when analyzing

Number of Threads	Total time cost in each thread
1 Thread	214.2
2 Thread	152.6
3 Thread	117.7

Table 5.3: Total cost of analysis(Altoro Mutual)

a large program. To analyze a call path in a working thread takes more time than creating a new task in the master thread. However, a larger application means longer call paths. As we duplicated the call graph, the memory usage is significantly increased when dealing with a larger application. In this case, the memory limitation of the test computer becomes the bottleneck.

- **Total number of processed routines**

  The result of the total number of processed routines is shown in Table 5.4 and Figure 5.11.

  As we can see, the increase in the total number of processed routines is very similar to the small test case. Compared to the result in Figure 5.9, we can conclude that the increase in total number of processed routines is small even when analyzing a large program.

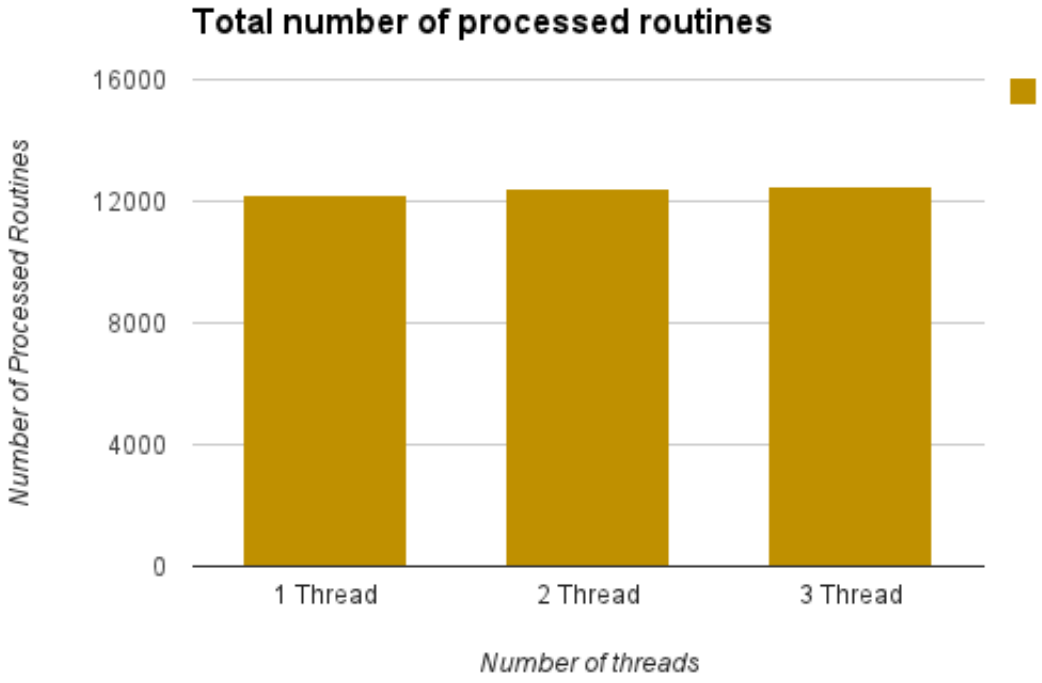

Figure 5.11: Average number of processed routines (Web Goat)

Number of Threads	Total Number of Processed Routines
1 Thread	12219
2 Thread	12413.33
3 Thread	12476.33

Table 5.4: Average Number of Processed Routines(Web Goat)

Chapter 6

Conclusions and Future Directions

This thesis provides algorithms and implementations for concurrent taint dataflow analysis. By integrating our concurrent analysis into a IBM product AppSacn Source, we have successfully improved the performance of the analysis by reduce the time of analysis. The contributions of this thesis are discussed in the first part of this chapter.

The work may be enhanced in several ways in future work. Both the enhancement of concurrency and support of other features are discussed as the final section of this document.

6.1 Summary of Contributions

- Improvement of the performance of static taint analysis: As shown in Chapter 5, the performance of the concurrent taint dataflow analysis is improved by multi-threading.

- A concurrent taint dataflow analysis algorithm: The algorithm of concurrent taint dataflow analysis is introduced and implemented.

- A Methodology for sharing pointer type data between threads without modifying

class constructors.

6.2 Future Work

6.2.1 Static and Global Variables

Static variables are not tainted and tracked by the current implementation. This is because tracking the value for a static variable across the call graph, and associating the value with every routine and every call, rapidly becomes expensive unless a mechanism for pruning the information is available. The same applies to global variables.

To address this issue, a fixed-point approach can be taken into consideration. The fixed-point would be reached when analyzing all call paths in all orders, the tainted values of this kind of variables do not change.

6.2.2 Pointer-based Calls

Future work might involve supporting passing of addresses of routines as arguments, with the corresponding propagation to the call sites and hence context-specific enlargement of the call graph.

6.2.3 Memory Management

Future work might also involve memory management. A fully multithreading supported memory management is required. The memory management should be able to allow duplication of all objects and capable to clean memory within a thread.

6.2.4 Duplication

In our current implementation, some data is unnecessarily duplicated in every working thread. The duplication is very memory consuming. To avoid copying unnecessary data, future study of the original implementation is necessary. With more understanding of the original implementation, unnecessarily duplicated data could be avoided.

6.2.5 Scalability

In our current experiment setup, we were not able to test the scalability of this concurrent program for larger number of concurrent threads. The bottleneck in our system was the memory. In the future, we would like to test the scalability of our approach, using a larger number of CPU cores and much larger memory, in order to determine whether the performance is esentially proportional to the number of concurrently running threads.

References

[1] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[2] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and William Pugh. Using static analysis to find bugs. *Software, IEEE*, 25(5):22–29, Sept 2008.

[3] J. L. Baer and C. S. Ellis. Model, design, and evaluation of a compiler for a parallel processing environment. *IEEE Trans. Softw. Eng.*, 3(6):394–405, November 1977.

[4] Jean-Loup Baer. Mutual exclusion. In *Encyclopedia of Computer Science*, pages 1215–1216. John Wiley and Sons Ltd., Chichester, UK, 2003.

[5] Thoms Bell. The concept of dynamic analysis. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, pages 216–234, London, UK, UK, 1999. Springer-Verlag.

[6] Edward Berard. Testing object-oriented software (abstract). In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, pages 226–, New York, NY, USA, 1992. ACM.

[7] D. Byers and N. Shahmehri. Prioritisation and selection of software security activities. In *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, pages 201–207, March 2009.

[8] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, first edition, 2007.

[9] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, 4(3):178–187, 1978.

[10] World Wide Web Consortium. Sql injection. `http://www.w3schools.com/sql/sql_injection.asp`. [Online, Accessed 2014-10-11].

[11] Steve Cook. Languages and object-oriented programming. *Softw. Eng. J.*, 1(2):73–80, March 1986.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Language and Systems*, 13:451–490, 1991.

[13] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965.

[14] A.K. Ghosh, T. O'Connor, and Gary McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of Security and Privacy, 1998. 1998 IEEE Symposium on*, pages 104–114, May 1998.

[15] W.B. Glisson, L.M. Glisson, and R. Welland. Secure web application development and global regulation. In *Proceedings of The Second International Conference on Availability, Reliability and Security, 2007. ARES 2007.*, pages 681–688, April 2007.

[16] J. W. Havender. Avoiding deadlock in multitasking systems. *IBM Syst. J.*, 7(2):74–84, June 1968.

[17] IBM. Altoro mutual. `http://altoromutual.com`. [Online, Accessed 2014-10-11].

[18] IBM. Ibm security appscan source. `http://www-03.ibm.com/software/products/en/appscan-source/`. [Online, Accessed 2014-10-11].

[19] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, PLAS '06, pages 27–36, New York, NY, USA, 2006. ACM.

[20] K. Karppinen, M. Lindvall, and L. Yonkwa. Detecting security vulnerabilities with software architecture analysis tools. In *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*, pages 262–268, April 2008.

[21] Gary McGraw. Software security. *Security Privacy, IEEE*, 2(2):80–83, Mar 2004.

[22] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[23] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[24] OWASP. Owasp webgoat project. `https://www.owasp.org/index.php/Webgoat`. [Online, Accessed 2014-10-11].

[25] OWASP. Sql injection. `https://www.owasp.org/index.php/SQL_Injection`. [Online, Accessed 2014-10-11].

[26] B. P. Pokkunuri. Object oriented programming. *ACM SIGPLAN Not.*, 24(11):96–101, November 1989.

[27] S.J. Prowell, M. Pleszkoch, K.D. Sayre, and R.C. Linger. Automated vulnerability detection for compiled smart grid software. In *Innovative Smart Grid Technologies (ISGT), 2012 IEEE PES*, pages 1–5, Jan 2012.

[28] C. V. Ramamoorthy and Phillip C. Sheu. Object-oriented systems. *IEEE Expert: Intelligent Systems and Their Applications*, 3(3):9–15, September 1988.

[29] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[30] B.G. Ryder. Constructing the call graph of a program. *Software Engineering, IEEE Transactions on*, SE-5(3):216–226, May 1979.

[31] D. J. Rypka and A. P. Lucido. Deadlock detection and avoidance for shared logical resources. *IEEE Trans. Softw. Eng.*, 5(5):465–471, September 1979.

[32] André Schiper. *Concurrent Programming*. Halsted Press, New York, NY, USA, 1989.

[33] Bernard Scholz, Chenyi Zhang, and Cristina Cifuentes. User-Input Dependence Analysis via Graph Reachability. Technical Report TR-2008-171, March 2008.

[34] V. Seshadri, S. Weber, D. B. Wortman, C. P. Yu, and I. Small. Semantic analysis in a concurrent compiler. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 233–240, New York, NY, USA, 1988. ACM.

[35] V. Seshadri and D. B. Wortman. An investigation into concurrent semantic analysis. *Softw. Pract. Exper.*, 21(12):1323–1348, December 1991.

[36] Vibhu Saujanya Sharma and Kishor S. Trivedi. Architecture based analysis of performance, reliability and security of software systems. In *Proceedings of the 5th International Workshop on Software and Performance*, WOSP '05, pages 217–227, New York, NY, USA, 2005. ACM.

[37] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[38] Mark Stefik and Daniel Bobrow. Object-oriented programming: Themes and variations. *AI Mag.*, 6(4):40–62, January 1986.

[39] Richard Strong, Jayaram Mudigonda, Jeffrey C. Mogul, Nathan Binkert, and Dean Tullsen. Fast switching of threads between cores. *SIGOPS Oper. Syst. Rev.*, 43(2):35–45, April 2009.

[40] Jay-Evan J. Tevis and John A. Hamilton, Jr. Static analysis of anomalies and security vulnerabilities in executable files. In *Proceedings of the 44th Annual Southeast Regional Conference*, ACM-SE 44, pages 560–565, New York, NY, USA, 2006. ACM.

[41] Radha Vedala and Simhadri Anil kumar. Automatic detection of printf format string vulnerabilities in software applications using static analysis. In *Proceedings of the CUBE International Information Technology Conference*, CUBE '12, pages 379–384, New York, NY, USA, 2012. ACM.

[42] Wikipedia. Process. `http://en.wikipedia.org/wiki/Process_(computing)`. [Online, Accessed 2014-10-11].

[43] Wikipedia. Program analysis. `en.wikipedia.org/wiki/Program_analysis`. [Online, Accessed 2014-10-11].

[44] Wikipedia. Sql injection. `http://en.wikipedia.org/wiki/SQL_injection`. [Online, Accessed 2014-10-11].

[45] Wikipedia. Sql injection. `http://en.wikipedia.org/wiki/Dynamic_program_analysis`. [Online, Accessed 2014-10-11].