2009-01-01

An Optimization Approach for the Cascade Vulnerability Problem

Christian Servin
University of Texas at El Paso, christians@miners.utep.edu

AN OPTIMIZATION APPROACH FOR THE
CASCADE VULNERABILITY PROBLEM

CHRISTIAN SERVÍN MENESES

Department of Computer Science

APPROVED:

Martine Ceberio, Chair, Ph.D.

Eric Freudenthal, Chair, Ph.D.

Vladik Kreinovich, Ph.D.

Virgilio Gonzalez, Ph.D.

Patricia D. Witherspoon, Ph.D.
Dean of the Graduate School

to my

MOTHER, FATHER, SIBLINGS,
and
GRANDMOTHERS

with love

AN OPTIMIZATION APPROACH FOR THE CASCADE VULNERABILITY PROBLEM

by

CHRISTIAN SERVÍN MENESES

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

May 2009

Acknowledgements

I would like to express my deep-felt gratitude to my advisors, Dr. Martine Ceberio and Dr. Eric Freudenthal, of the Computer Science Department at The University of Texas at El Paso, for their advice, encouragement, enduring patience, and constant support. My two mentors helped me to think, understand, and explain ideas in a critical, professional, and articulated manner. I would also like to thank Dr. Stefano Bistarelli, for his helpful ideas in developing the CVP tool.

I also wish to thank the members of my committee, Dr. Vladik Kreinovich, of the Computer Science Department and Dr. Virgilio González, of the Computer Electrical Engineering Department, at The University of Texas at El Paso. Their suggestions, comments, and additional guidance were invaluable to the completion of my master thesis. Dr. Kreinovich was especially helpful in providing additional guidance.

Special thanks to Dr. Ben Flores, and the Louis Strokes Alliance for Minority Participation (LSAMP)/Bridge to the Doctorate program sponsored by the National Science Foundation under the grant HRD-0217691. This program was responsible for supporting me for three years of graduate education (including part of my Ph.D.).

Additionally, I want to thank the department of Computer Science at The University of Texas at El Paso faculty and staff for all their hard work and dedication, providing me the means to complete my degree, and for preparing me to become a computer scientist. This includes (but certainly is not limited to) the following individuals:

Dr. David Novick

> He made it possible to travel to Europe to establish collaboration with the Constraint Programming community at CP07. I would like to thank him for mentoring me to make things clear in what I really want to pursue in life.

Dr. Steve Roach

> For helping me think critically, and to find the counter-example from other peoples arguments. Also for supporting me in my final semesters through the Center for Science, Technology, Ethics and Policy (CSTEP).

Dr. Rodrigo Romero

> For the many interesting talks about life in general. Thank you for sharing your wisdom, experience, and ideas with me.

Finally, I must thank Sacnité, Rafael, and Alexis for putting up with me during the development of my thesis with continuous, loving support, and patience. Thank you for always being there, even during late nights.

NOTE: This thesis was submitted to my Supervising Committee on the May 5th, 2009.

Abstract

In inter-connected systems, where several computers share information with each other, problems may arise when inappropriate information starts to flow through. For example, let us consider a simple scenario of a university composed of three departments: payroll, financial aid, and academic services. We know that the payroll department deals with sensitive information, such as social security numbers, dates of birth, amounts of wages, etc. The financial aid department may use information that payroll owns. Similarly, the academic department communicates with the financial aid department. An intruder might take advantage of this network connectivity and create an inappropriate flow of information across the network, leading to the so-called Cascade Vulnerability Problem (CVP).

Several approaches have been proposed to solve this problem. Among them, the approach of Bistarelli et al. [9, 13] is of particular interest, as they express a solution of the problem using Constraint Programming, more specifically, soft constraints. This approach not only enables the detection of vulnerable paths in a network, but also eliminates the links in the network that provoke the security leakage.

This paranoid approach to network connectivity, although trivial to compute, is impractical because it neither considers the organizational value nor risk of each individual network link. When either is quantified and overall limitations are specified, CVP can be reduced to a constraint optimization problem.

In our proposed approach, we apply and implement the approach of Bistarelli et al, using soft constraints to model, detect and solve the cascade vulnerability problem, but we extend it by using the minimum weighted hitting set approximation algorithm to deal with the connection values. This way, we are able not only to detect the CVP, but also to make the least expensive cuts in the networks connections.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Managing security in an organization is a serious issue. To prevent inappropriate disclosure of sensitive information among an organization's departments, the disclosure rules must be carefully managed. In many practical situations, it is desired to combine information from different departments, i.e. information from different security contexts.

For example, consider the following scenario. A hospital's staff are aggregated into departments based on the hospital's function, e.g., pharmacists, nurses, doctors, fund-raisers, etc. A description of a set of connected links it is depicted in Figure 1.1a. Now, consider the following:

Celebrity Paulette arrives to the ER with a severe injury. After she was treated in the ER, Nancy the room nurse, is assigned to take care of Paulette's health. In the meantime, while Paulette is in the hospital's room, a reporter Ralph shows up at the hospital asking for information about Paulette's health to Nancy, who does not use discretion, provides details about Paulette's incident. Ralph obtains and publishes information from celebrity Paulette in a weekly magazine, specializing in celebrity news.

This is an example of how information can inappropriately "leak" through multiple intermediaries using available "network" connections, eventually leading to a loss of confidentiality. In [9], Bistarelli categorizes such leakage of confidential information as a Cascade Vulnerability Problem (CVP) [40].

In our ER's example, observe that the principal problem of information leakage is caused by Nancy. Nancy the nurse has many roles in the hospital. For instance, she has to inform doctors about the patient's condition, treat patients' preferences, interact with guests, and, in this scenario, it was not prevented that a guest could have access to information

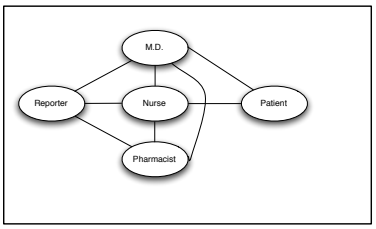
of a particular patient. Therefore, like all "network connected" entities, Nancy poses a potential information leakage risk to the hospital. If the organizations only objective is to eliminate risk of cascading information leakage, they can sever all communication links between workers, thus isolating them from each other as shown in Figure 1.1b We observe that, in this absurd scenario that ignores the value of communication, the hospital will fail to function.

Several approaches have been proposed to characterize and detect CVP. Our work extends the work of Bistarelliet al. [9, 13]; which provides a framework for quantifying and constraining aggregate risk through the selective severing of communication links. Bistarelli implicitly assumes that each communication link is of equal value and it expresses a solution to the problem in terms of constraints that can be solved using *constraint programming*.

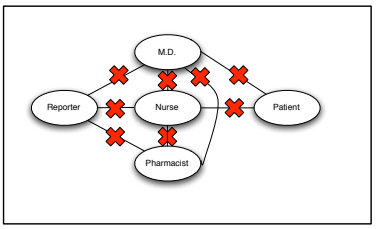
Bistarelli et al. provide an algorithm to determine the minimal number of link cuts necessary to solve the CVP. However, this approach overlooks the operational value of providing connectivity. For example, some connections among computers are more valuable than others (i.e., staff in the hospital that generates a CVP may be greater than other staff that generate this problem too.)

Therefore the solution provided by Bistarelli et al. may lead to choose to remove a link that is essential to providing services. For instance, in our hospital example, there is a need for nurses to talk with patient's guests, control the medication's amount to patients and report patient's status to the M.Ds. By applying the current approach, we determine that Nancy's connection is the responsible of causing a CVP. However, the latter "minimum number of cuts" solution could definitely disconnect Nancy's from Ralph, but also from Paulette, the pharmacist, and the MD as shown in Figure 1.1c.

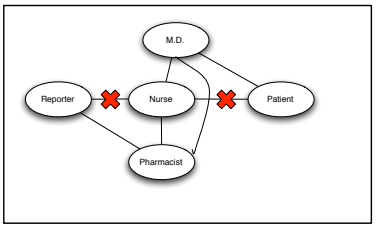
We notice that cutting this edge, we eliminate all benefits of having a nurse in the hospital's room. Clearly another approach to prevent Ralph obtaining information about patients and to keep the organization's resources benefited (i.e., Nancy the nurse) needed. By assigning values to the connections within the network, we constrain the elimination process proposed by Bistarelli et al. by considering the operational value of the connections



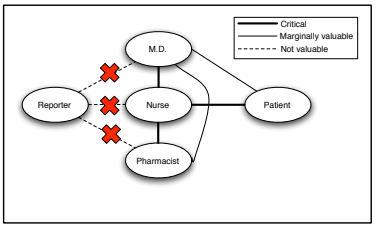
(a) Initial set of communication paths.



(b) Conservative solution, where all possible paths are eliminated.



(c) Algorithm proposed proposed by Bistarelli et al. , select minimal path cuts. Observe that as selected above, the communication of patient is eliminated.



(d) Proposed different types of communication links between parties in the network.

Figure 1.1: These figures depict four different hospital's network configurations.

that are generating the CVP. Considering these values, we are interested in finding the least expensive connections to eliminate in the network that are causing the CVP. These least expensive connections are called solutions, and it is desirable to find the best solution, among the available solutions. The process of finding the best solutions among the possible available considering constraints is called *constrained optimization problem*. For instance, by considering different values of the links, e.g., critical, marginally valuable, and not valuable, at the time of eliminating the links, we remove the solution that generates the lowest cost in the network as shown in Figure 1.1d.

Constraint Programming (CP) is a paradigm in which the problem at hand is expressed in terms of *what* goal should be achieved and not in terms of *how* to achieve it. In CP, problems are modeled in terms of their constraints (or requirements), the variables the constraints bind, and domains for each variable: this formulation is called a *Constraint Satisfaction Problem* (CSP). A solution of a CSP is a complete instantiation of the variables of the CSP, such that all constraints are satisfied. Most of the constraint solvers provide all

solutions: such solvers are called complete solvers. In this thesis, we consider only complete solvers.

In some cases, the traditional constraint framework is limited. It may happen that a CSP has no solution, i.e., when there are conflicting constraints. Such problems are called over-constrained problems and cannot be solved as CSPs. When dealing with over-constrained problems, we might be able to relax some conflicting requirements in order to obtain solutions. For example, suppose that a person must meet with five people who have different schedules. There might not be a time that fits everybody's schedule. In this situation, some schedules can be modified to make the meeting possible, or maybe holding the meeting with fewer people might offer a solution. Either of these options constitutes a relaxation of the original constraints. There is a specific research field dedicated to studying such relaxed problems: it is the research area on so-called *soft constraints*, also known as *flexible constraint*.

Bistarelli et al. used a soft constraint framework [11, 12] to solve the cascade vulnerability problem. This approach modeled the problem in terms of valid/invalid connections between computers, and the information flows that these connections generate. These flows of information are called paths. The approach of Bistarelli et al. enables the detection of vulnerable paths in a network, and it also identifies the computers links in the network that are causing a security leakage.

In this thesis we present an extension to the algorithm proposed by Bistarelli et al. considering valued connections into account. Our contribution in solving the CVP consists in determining the cost of the communication between computer systems by quantifying the value associated with their connections (we call these values weights). A number of connections in a network are more valuable than others and we are interested in the impact of removing the least expensive and vulnerable connections in the network.

1.1 Thesis' Overview

The thesis is composed of five chapters. Chapter One (Introduction) describes the problem we are solving, the motivation behind this work, and the difficulties of solving this problem.

Chapter Two (Background) presents the concepts, definitions, and notation that I used throughout the thesis. The preliminary concepts related to the fields of Constraint Programming, Optimization, and Computer Security are summarized.

The first section in Chapter Two presents Constraint Programming (CP.) It defines constraints and Constraint Satisfaction Problems (CSP), discusses the stages of the constraint solving process. Also, it discusses different frameworks for soft constraints, and presents widely-known methods and techniques for optimization. Subsequently, mention what a constrained optimization problem is. The second section introduces Computer Security concepts. Computer Security is a wide and large area to discuss, however, our work focuses in fundamental concepts and defines what is considered to be "secure." We also mention the problem we are solving and its impact on Computer Security. The third section in Chapter Two discusses existing approaches that solve the Cascade Vulnerability Problem (CVP), the techniques and tradeoffs of each approach. The shortcomings of the existing constraint programming approach that serves as a basis to this thesis are identified.

Chapter Three (Problem and Solution) introduces an extension to the Bistarelli et al. algorithm to solve the Cascade Vulnerability Problem that addresses the shortcomings presented in the previous chapter. This chapter defines the notation, then describes the problem and the solution in detail.

Chapter Four (Implementation) introduces the implementation of a CVP detection simulator tool. This tool graphically demonstrates the proposed solution as well as the previous approach. The user manual of the tool can be found in Appendix D.

Chapter Five (Results and Evaluation) reports the experiments we conducted on a real-world network configuration and the significance of its results.

Chapter Six (Conclusion) recapitulates the contribution of this thesis, discusses the

impact of the experiments and results, and proposes future work.

Chapter 2

Background

In this chapter, I present preliminary concepts in the fields of Constraint Programming (CP) and computer security. Both of these fields are broad and deep in scope. This chapter is devoted to providing a context of both fields that gives a reasonable understanding about why the cascade vulnerability problem is a computer security issue, and why it is reasonable to solve the problem using constraint programming.

In the CP subsection, the main concepts, vocabulary, and formal definitions for constraint programming are presented. In particular, the concept of Constraint Satisfaction Problem (CSP) is introduced, and solutions are defined. Different kinds of constraints (such as discrete or continuous constraints) are also described. More specifically, we introduce the notion of Soft Constraints. Such constraints are helpful when no solution is obtained when trying to solve a CSP. Indeed, in such a situation, it might be desirable to soften the constraints that are not considered as "crucial constraints", in order to find a solution. Several frameworks of soft constraints are presented, and we later narrow our scope to the *semiring-based soft constraints framework*, since it is the framework we use throughout this thesis.

Now, regardless of the kind of soft constraints used, the idea is to obtain the best possible solution among the available ones. Hence optimization can be used as a naturally way to find the best solution among the available ones. Optimization considering constraints is called *constrained optimization*. In this chapter we present constraints, soft constraints, and optimization problems. In Section 2.2, a general overview about optimization problems with and without considering constraints is discussed. Widely-known methods and techniques to solve optimization problems are discussed in Section 2.2.3.

Also, this chapter includes a section on computer security, and for the sake of simplicity, we limit this thesis' section to fundamental knowledge about what is considered to be secure and to security models that are designed for trusted operating systems. Section 2.3 is an overview of the three main properties of a secure system are described, i.e., confidentiality, integrity, and availability. This section defines these properties and explains why it is important to find a "correct" balance between these three properties. Section 2.3.2 delves further into the most outstanding security models that have been proposed through time. In particular, we describe the models that are focused on a particular network environment called the *Multilevel security* (MLS). One problem that may arise in the MLS is the so-called Cascade Vulnerability Problem (CVP), and in this chapter, we describe what a CVP is and why it is considered to be a security problem.

Finally, in Section 2.4, the *current approaches* for solving the CVP along with their limitations are presented. In this, we discuss the approach of Bistarelli et al. that we extend in this thesis.

2.1 Constraint Programming

Constraint Programming (CP) is a powerful programming paradigm that has been successfully applied in many applications: natural language processing in linguistics [14], industrial problems such as vehicle routing [3], operational control of water systems, optimization and scheduling [6, 26], and protein folding in bioinformatics [1, 4, 5].

CP has attracted the attention of researchers because of its potential to efficiently solve hard real-world Science and Engineering problems. At the present time, the research in the field consists in enabling computers to solve the given constraint problems. The range of research goes from programming languages to algorithm design, covering applications of constraint programming techniques to better identify the needs of real-world settings.

CP has been identified by the Association of Computer Machinery (ACM) as one of strategic directions in computing research. Eugene Freuder once said:

> "Constraint Programming represents one of the closest approaches computer science has yet made to the *Holy Grail of programming*: the user states the problem, the computer solves it" [24].

People from computer industry have expressed interest in applying CP to their future products. Recently, at the 2007 international conference on Principles and Practice of Constraint Programming, companies such as Google, IBM, ILOG, Nokia discussed the impact of solving their tasks using CP. These companies also discussed the need for training more constraint programmers, for designing constraint solvers, and developing techniques to solve optimization problems. Recently IBM bought ILOG.

2.1.1 Constraint Programming: preliminary concepts

Constraint Programming is a paradigm that allows the user to express the problem's relations in terms of *what* goal should be achieved and not in terms of *how* to achieve it. Constraints are relations between variables or unknowns. Each variable or unknown has a corresponding range of values. We can illustrate this with an example:
Example: A student is driving late to school on the highway. There is a maximal speed limit of 60 miles per hour on the highway. The current time is 7:30 a.m. and the class starts at 8:00 a.m. The person is 15 miles away from school, and he/she must arrive on time to class.

Every single sentence is either a fact or a constraint:

$$\begin{cases} time = 7:30 \\ distance = 15 \text{ miles} \\ speed \leq 60 \text{ miles an hour} \\ arrival\_time \leq 8:00 \quad // \text{ depends on the speed.} \end{cases}$$

These constraints define restrictions, relations on the only variable of this problem: the speed. This variable may have a range of possible values of $[0, 80]$, but we can see that it is restricted to 60 as a maximum value, due to the speed limit on the highway. In order to arrive at 8:00 a.m. and not violate the speed limit constraint, the *speed* must be in the range of $[40, 60]$. Therefore, any instanciation of *speed* in $[40, 60]$ is a solution to our problem.

Definition 1 [Constraint]. Let $X = \{x_1, \ldots, x_k\}$ be a finite sequence of variables, with their respective associated domains $D = \{d_1, \ldots, d_k\}$ (so each variable x_i ranges over the domain D_i).

A **constraint** c on X is the subset of the cartesian product $D_1 \times \ldots \times D_k$. When $k = 1$, it is said to be a unary constraint, and when $k = 2$ it is said to be a binary constraint.

In CP, problems are modeled in terms of their constraints (or requirements), the vari-

ables the constraints bind, and domains for each variable: this formulation is called a *Constraint Satisfaction Problem* (CSP).

Definition 2 [**Constraint Satisfaction Problem (CSP)**]. A CSP is a triple $\varphi = \langle X, D, C \rangle$, where:

- $X = \{x_1, \ldots, x_n\}$ is a finite set of variables;
- $D = \{d_1, \ldots, d_n\}$ is a finite set of domains, where each domain is a set of values for the corresponding variable;
- $C = \{c_1, \ldots, c_n\}$ is a finite set of constraints restricting the values that the variables $\{x_1, \ldots, x_n\}$ can simultaneously take.

Definition 3 [**Solutions**]. A solution of a CSP is a complete instantiation of the variables of the CSP, such that all constraints are satisfied. An instantiation of the set of variables $\{x_1, \ldots, x_n\}$ of a CSP is a tuple of values for each variable $(a_1, \ldots, a_n)$, where each pair (x_i, a_i) represents the assignment of value a_i to variable x_i.

Most of the constraint solvers return all solutions: such solvers are called complete solvers. In this thesis, we only consider complete solvers. There exist different types of constraints used to represent a variety of problems. The main two types of constraints are:

- discrete constraints: they are constraints whose variables' domains are discrete. The variables can take on a finite number of values. Example 2.1.1 is an example of a constraint problem with discrete domains.
- continuous constraints: they are constraints whose variable's domains are continuous. The variables can take on an infinite number of values. Example 2.1.1 is an example of a constraint problem with continuos domains.

Now let us see an example of each such type of constraints:

Example of discrete constraints

Let us consider the graph illustrated in Figure 2.1.1. The graph is composed of four vertices (A, B, C, D), one for each vertex. Each vertex is defined on the following domain values: A = {0,1}, B = {0,1}, C = {0,1,2}, and D = {0,1}. The problem is to find the possible assignation to each vertex that satisfy the constraints stated to each edge. Given the constraints:

$$\begin{cases} c_1 := A = B; \\ c_2 := B \neq C; \\ c_3 := C > D; \\ c_4 := B = D. \end{cases}$$

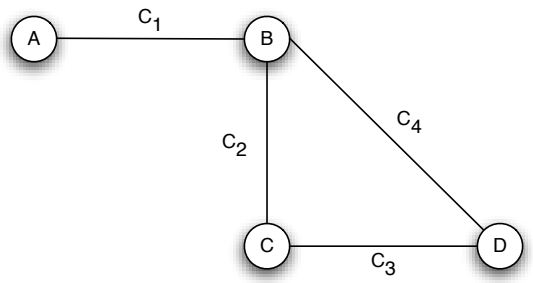


Figure 2.1: Example of a network representing a discrete CSP.

Result 1				
const/vars	A	B	C	D
c_1	0	0	_	_
c_2	_	0	1	_
c_3	_	_	1	0
c_4	_	0	_	0

Table 2.1: All constraints are satisfied, this is a solution

Result 2				
const/vars	A	B	C	D
c_1	0	0	_	_
c_2	_	0	2	_
c_3	_	_	2	1
c_4	_	**0**	_	**1**

Table 2.2: The value for B used on c_1 and c_2 conflicts with constraint $c_4 := B = D$. This is not a solution.

Result 3				
const/vars	A	B	C	D
c_1	0	0	_	_
c_2	_	0	2	_
c_3	_	_	2	0
c_4	_	0	_	0

Table 2.3: All constraints are satisfied, this is a solution

Result 4				
const/vars	A	B	C	D
c_1	1	1	_	_
c_2	_	1	0	_
c_3	_	_	**0**	**0**
c_4	_	_	_	_

Table 2.4: Constraint $c_3 := C > D$ is not satisfied, when C and $D = 0$. This is not a solution.

Result 5				
const/vars	A	B	C	D
c_1	1	1	_	_
c_2	_	1	2	_
c_3	_	_	2	0
c_4	_	**1**	_	**0**

Table 2.5: Constraint $c_4 := B = D$ is not satisfied, since $B \neq D$. This is not a solution

Result 6				
const/vars	A	B	C	D
c_1	1	1	_	_
c_2	_	1	2	_
c_3	_	_	2	1
c_4	_	1	_	1

Table 2.6: All constraints are satisfied, this is a solution.

The problem has three solutions: sol = $\{solution_1, solution_2, solution_3\}$, where:

$solution_1$ = (0, 0, 1, 0). // that comes from result 1
$solution_2$ = (0, 0, 2, 0). // that comes from result 3
$solution_3$ = (1, 1, 2, 1). // that comes from result 6

Example of continuous constraints: Intersection of two functions

Let us consider two functions:

$$f(x) = \frac{1}{x} + \frac{1}{2};$$
$$g(x) = 2 \cdot \sin(x);$$

Given the two functions illustrated in Figure 2.1.1, find the intersection of these two functions, where $x \in [0, 4]$, and $y \in [0, 3]$.

This problem can be expressed as a CSP:

$$\text{Constraints: } f(x) = \frac{1}{x} + \frac{1}{2}$$
$$g(x) = 2 \cdot \sin(x)$$
$$\text{Variables: } x \text{ and } y$$
$$\text{Domains: } \mathrm{x} \in [0, 4]$$
$$\mathrm{y} \in [0, 4]$$

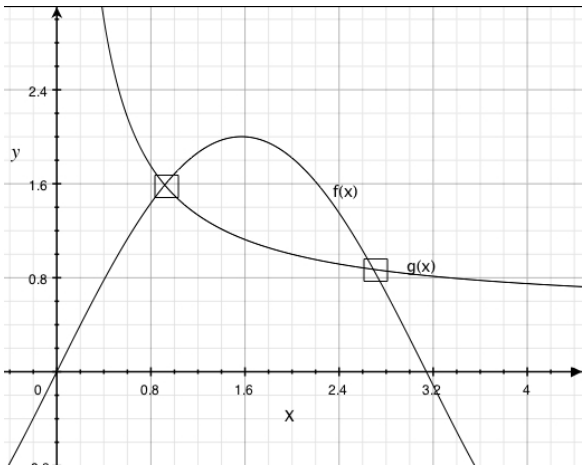


Figure 2.2: Intersection of two functions.

Here, to find the intersection of these two functions we use a constraint solver specialized in continuous domains called *Realpaver* [1] and we obtain the following result:

[1]Realpaver: http://www.sciences.univ-nantes.fr/info/perso/permanents/granvil/realpaver/

```
INITIAL BOX
  x in [0 , 4]
  y in [0 , 3]

OUTER BOX 1
  x in [2.690632487075013 , 2.690632487075014]
  y in [0.8716598252655083 , 0.8716598252655087]

  precision: 1.33e-15, elapsed time: 0 ms

OUTER BOX 2
  x in [0.9182281742617513 , 0.9182281742639904]
  y in [1.589053928019064 , 1.58905392802176]

  precision: 2.7e-12, elapsed time: 0 ms

END OF SOLVING
  Property:     reliable process (no solution is lost)
  Elapsed time: 0 ms
```

This result means that we obtain two solutions that satisfy the constraints of $f(x)$ and $g(x)$, the ones mentioned in `OUTER BOX 1` and `OUTER BOX 2`.

The traditional CSP framework has been successfully applied to solve problems in discrete and continuos domains. Nevertheless, in many real-world problems, it is desired to express preferences or priorities to the problems, hence it cannot be expressed or solved using the traditional CSP framework. The idea behind expressing these preferences/priorities over a problem is called *soft constraints* and it is described in the following section.

2.1.2 Soft Constraints

General Description

Although the framework of classical constraint satisfaction problems is very expressive and offers a natural formalism for representing problems, it has evident limitations, mainly due to the fact that it does not offer enough flexibility to represent real-life scenarios where the knowledge is neither completely available nor crisp [12]. Besides regular constraints that express strict requirements, as described above, there exists the option to relax requirements that are not crucial.

In 1987, Alan Borning et al. introduced hierarchical constraints [15, 17, 16], as an alternative scheme for extending constraint logic programming. The main motivation behind this scheme, was the need to express preferences as well as strict requirements for applications such as graphics, page layouts, and decision support systems.

The Hierarchical Constraints framework consists in eliminating all potential solutions that are *not as good as* other potential solutions. This idea, later on, led to the definition of several frameworks to express soft/flexible constraints.

From a general point of view, soft constraints are constraints that do not necessarily have to be satisfied (but will be in general if this is possible). Such constraints are very convenient when the user is not sure whether the constraints can be satisfied or not. The description of soft constraints is usually similar to that of regular constraints, but the solving process is different, since "solutions" can be found even when the original CSP did not have any.

Best compromises (between the constraints) are sought rather than solutions to all constraints. As a result, solving soft constraints comes down to solving an optimization problem.

Soft constraints (or sometimes also called flexible constraints) are particularly useful to model and solve over-constrained problems. A problem is said to be an over-constrained problem when constraints cannot be satisfied.

For example:

A student has a list of tasks to do during the day, including:

1. attending class from 8:00 a.m. to 3:30 p.m.;
2. doing his/her homework;
3. making bill payments at store before 4:00 p.m.;
4. doing his/her laundry;
5. picking up baby at daycare before 4:00 p.m.

In order to accomplish all these tasks, it is necessary to prioritize them. In this particular list, the student might want to prioritize the tasks as follows: class, pick up his/her baby from daycare, make bill payments, do homework, and finally do the laundry. However, it may be possible that h/she will not make it to pay the bill at the store (depending on the distance and the time left). In case the student would like to be able to complete all the tasks, h/she should be able to accept that either, arrive late to pick up baby or pay a late-fee to the store (e.g., pay the least expensive late fee for either the baby or the bills).

Frameworks for soft constraints

Several frameworks exist to express soft constraints; e.g., possibilities, priorities, probabilities, costs as detailed in [11, 12, 19, 35, 36]. For instance, instead of concluding that there is no solution for a problem, we can reconsider the problem by excluding the constraints that are not obligatory. Let us review some of the most classical frameworks:

- **Hierarchical CSP.** In order to make full usage of the constraint programming paradigm, this framework introduced the notion of preferences. This framework considers hard requirements and preferential requirements (soft constraints), where the preferential ones are satisfied only if the hard constraints were satisfied. In Hierarchical CSP, detailed in [15, 17, 16], different levels of strength are proposed to each constraints as a way to denote preferences at the time of solving the CSP, i.e., *required, strong, preferred, default, and weak*.

- **Max CSP (MAXCSP).** In this framework, it was proposed to search for the solutions that satisfy all the obligatory requirements and as many optional constraints as possible [23]. The objective is to find a total assignment to the variables that satisfy the maximum number of constraints. Solutions are elements of the search space that maximize the number of constraints that are satisfied.

- **Partial CSP (PCSP).** This framework is used in situations when the problem is too difficult to solve completely but it is desired to obtain a "good enough" solution for this over-constrained problem. Partial constraint satisfaction, detailed in [23], consists in weakening the CSP[2] in order to obtain alternative versions of the original problem.

- **Weighted/Valued CSP (WCSP).** This framework is used to associate a cost to each constraint in order to minimize the total cost of the solutions from the CSP. Similarly, the *Valued CSP* framework is used to represent a total order of violation degree of a problem. In this framework, the solutions are considered to be the elements of the search space for which its constraints violation value is minimal, i.e., [36].

- **C-semiring.** The c-semiring framework, detailed in [11, 12], is based on a semiring structure besides the constraints. A semiring: $\langle A, +, \times, 0, 1\rangle$, consists in:

  - a set A, and $0, 1 \in A$.

[2]i.e., a CSP C_1 is weaker than CSP C_2 if the solutions of C_2 is included in the solutions of C_1.

- $+$ is the additive operation, is a commutative and associative operator with $\mathbf{0}$ as its unit element;
- $\times$ is the multiplicative operation, is an associative operation such that $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element;
- $\times$ distributes over sum (i.e., for any $a, b, c \in A$, $a \times sum(b, c) = sum((a \times b), (a \times c))$).

The solutions for a c-semiring are the instantiations with the maximum evaluation with respect to the $\times$ operator. Also, several frameworks have been proposed that can be expressed as a c-semiring, for example:

- **Fuzzy CSP.** It extends the classical CSPs by associating a *preference level* with each tuple of values. The preference level is represented by a value in the interval $[0, 1]$, where 0 represents the worst value (e.g., the tuple that is not allowed), and 1 represents the best value (e.g., the tuple that is allowed). A solution of this formalism is defined as a set of tuples of values which have the maximal value [20, 34]. The semiring: $\langle [0, 1], max, min, 0, 1 \rangle$:
- **Probabilistic CSP.** It is used to represent uncertainty in constraints. This framework was introduced to model real-life scenarios where each constraint c has a certain probability $p(c)$ that is independent from other constraints to happen. This framework introduces new tuples to represent controllable decision variables and uncontrollable parameters, and a probability distribution over the possible values of the parameters [21]. This framework permits to model problems that are partially known and provides a probabilistic solution of the real problem. The semiring: $\langle [0, 1], max, \times, 0, 1 \rangle$

Soft constraints are particularly useful to model and solve over-constrained problems. The side effect of weakening constraints is that we might end up with too many solutions. In many situations, it is not enough to find the solutions that satisfy all given constraints,

we might as well need some kind of optimization involved in the process of finding solutions to find the best solution among the many solutions of the constraints alone. Thus, finding the best solution requires solving an optimization problem, where the "best solution" is defined by an objective function.

2.2 Optimization

In this section, a general overview about optimization and constrained optimization is presented. The term *optimization* refers to the branch of mathematics, which goal is to minimize or maximize a quantity/quality. Optimization is widely used in a variety of real-world problems, such as in engineering, science, operation research, finance, e.g., see Magoc [30]. Many algorithms, techniques, and theoretical frameworks exist to solve optimization problems in both discrete and continuous domains. Most of these techniques come from the disciplines of Operations Research (OR) and Combinatorics. In this section, we recall some of the popular ones, including *greedy approximation algorithms* that are usually applied to problems that are *NP-Hard*. NP-Hard problems are the type of problems that are considered very hard to solve, meaning that algorithms are likely to require too much computational time to find a solution even for inputs of reasonable size (e.g., see [18, 32, 38] for further details). Besides traditional optimization techniques, there is also *the constrained optimization*, i.e., in addition to optimize a function, we consider constraints as part of the formulation of the problem. These problems are called *Constrained Optimization Problem* (COP). Within the field of optimization, the special branch dedicated to special problems that on top of having an objective function to optimize, we have a set of candidates that is determined by constraints, this branch is called constrained optimization.

2.2.1 Optimization: preliminary concepts

In many practical situations, it is desired not only to obtain solutions to the constraints of a given problem, but to obtain the solution that is the best according to some criterion, or

objective function. This is optimization. Let us define what an optimization problem is:

Definition 4 [Optimization Problem]: Given a function $f : X \subset \mathbb{R}^n \to \mathbb{R}$, minimizing f over X consists in determining the set $Y \subset X$ such that: $\forall y \in Y$,

$$\boxed{\min_{x \in X} f(x) = f(y)}$$

Similarly, maximizing f over X is equivalent to minimizing $(-f)$ over X

$$\boxed{\min_{x \in X}(-f) = \max_{x \in X}(f).}$$

From now on, we will only talk about minimization.

there are two main kinds of optimization problems:

- **Global optimization** consists in finding the best solution over the whole search space.
- **Local optimization** consists in finding solutions that are minima in a neighborhood.

In this thesis, we only consider global optimization

Figure 2.2.1 illustrates an example of each.

2.2.2 Optimization subject to constraints

When we talk about solving an optimization problem under/subject to constraints, it means that is desired to minimize an objective function f subject to given constraints. The idea is to filter the solution set by adding a criterion over it in order to produce a better solution.

Definition 5 [Constrained Optimization Problem (COP)]
Given:

- $f : X \subseteq \mathbb{R}^n \to \mathbb{R}$ an objective function
- and constraints:

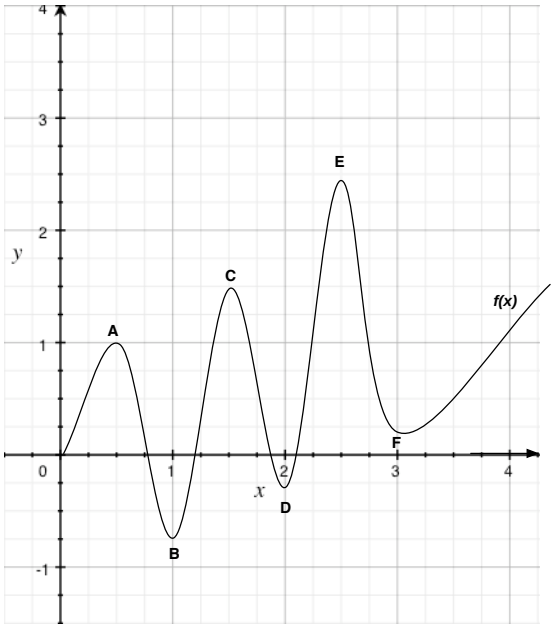


Figure 2.3: depicts a function $f(x)$ that has local minima at B, D, and F and has local maxima at A, C, and D. The global minimum is B and the global maximum is at E.

– $c_i : f_i(x) \bowtie 0$, for $i \in \{1, \ldots, p\}$ where $f_i(x)$: $X \subseteq \mathbb{R}^n \rightarrow \mathbb{R}$, and $\bowtie \in \{=, \leq\}$

determine: Y $\subseteq X$ such that: $\forall y \in Y$,

$$\boxed{f(y) = \min_{x \in C(X)} f(x)}$$

where: $c(X) \subset X$, x satisfies c_i, $\forall i \in \{1, \ldots, p\}$. and $C(X)$ is maximum for inclusion

2.2.3 Techniques for solving optimization problems

There are different approaches to solving optimization problems. Here we describe the most intuitive as well as the most used ones:

- **Enumeration**: In discrete problems, in order to find solutions to the problems, we can enumerate all possible instanciations of the variables. This particular technique, although naive, is still used as a general problem-solving technique to enumerate all possible candidates to a solution and checking whether these candidates satisfy the

problem constraints or not. The enumeration technique, sometimes called "brute-force search enumeration", will always find a solution (if it exists). However the time complexity is in most cases proportional to the number of candidate solution.

- **Branch and Bound**: Branch-and-Bound (B&B) methods belong to the category of traditional divide-and-conquer algorithms to find optimal solutions in optimization problems without having to consider all possible solutions one by one i.e., without having to enumerate. B&B is primarily used in global optimization. The B&B algorithms consist of two procedures; the *branching* (splitting) and the *bounding*.

  - Branching: is the part consisting in covering/exploring the search space. In the branching process, the domains of the variable are split in order to create two sub-domains needed to be explored.

  - Bounding: it is the process of calculating the lower bound and the upper bound of the currently explored sub-domains. The bounding part removes the solutions from the search space that violate some current bound according to the evaluation of the objective function $f(x)$.

- **Dynamic Programming**: it is a technique to solve optimization problems based on: 1) breaking the problem into small subproblems which are reused several times (overlapping subproblem), 2) finding a solution that can be constructed efficiently from optimal solutions to its subproblems (optimal solution), and by 3) recording previously-processed results to avoid repeating calculations (memoization). [2] Dynamic programming is usually applied to sub-problems that overlap with other sub-problems in the same problem.

  Dynamic programming is usually applied to sub-problems, which solution has already been computed and such solution is the optimal so far. In other words, each sub-problem keeps track of the optimal solution in order to avoid re-computation, so later on we can keep the optimal solution among the optimal ones. This property makes

the difference between these algorithms and the divide-and-conquer ones, where the divide-and-conquer solutions depend on the subproblems' solutions in order to finish.

- **Greedy Approximation Algorithm**: The approximation algorithms are the ones used to find approximate solutions to optimization problems. Usually, we apply approximation algorithms when the problem is NP-Hard. The greedy approximation algorithms select the option which is based on the current "local optimum" according to a given criterion, which consists in making the choice "the best" and then solve the subproblems that arise later; this selection is called "greedy choice property."

  Greedy algorithms guarantee an approximation to the optimal solution of the problem. As a difference of dynamic programming, greedy algorithms are not exhaustive, therefore do not cover the entire search space. These algorithms can be the fastest ones if the candidate solutions result to be the optimal, otherwise can lead to an inefficient results. For example, the *path finding* problem is a clear example that by selecting the "best solution so far" can lead to a deadened solution.

  These algorithms are widely used in graphs, e.g., *shortest path algorithm* and *vertex cover algorithm* detailed in [29, 41]. The *Minimal Weighted Set Cover* is a generalization of the vertex cover problem, and we will mention later on in this thesis.

2.3 Computer Security: Background

Computer Security has become an important concern to computer users, organizations, and the government. Computer users concern about *how secure their information is*. Of course, the term secure is vague for many of these people. Questions such as: are the users in my computer the only ones authorized to access my files? or are my friends in my *facebook* application able to break in my privacy? or what are the risks of using file-sharing programs such as *Napster* or *Kazaa*? have a specific meaning according to the context that are used.

2.3.1 What is Security?

Pfleeger [33], illustrates the notion about what is considered to be "secure", see, e.g., Figure 2.3.1 as a relationship between Confidentiality, Integrity, and Availability (CIA) with respect to computer assets (files, folders, programs, information). When we are referring to a "secure computer system" (architecture, firmware, hardware, software, management system) we are addressing computer-related issues that are evaluated in CIA terms.

- *Confidentiality* makes sure that assets are accessed by authorized users/parties/ systems.
- *Integrity* regards about assets that can be modified only by authorized users or only in authorized ways such as writing, changing status, deleting.
- *Availability* regards about the accessibility of assets to authorized users/parties/systems at appropriate times.

Undoubtedly, to pursue a "secure computer system" it is important to find the intersection of these three aspects, and then, we can have a notion of *what is secure*. Nowadays, there is a need for secure computer systems that detect and prevent unauthorized users from accessing unauthorized information in an efficient and reliable manner. For the sake of this work, we will concentrate in the computer security policies and secure models.

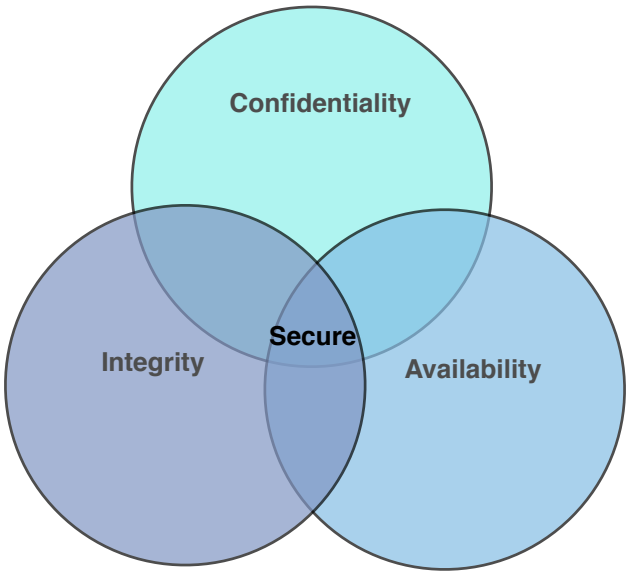


Figure 2.4: Secure: An intersection between confidentiality, integrity, and availability

Security models

Several models have been proposed through the time to establish computer security policies. There are three main lattice-based models:

- Bell-Lapadula: concentrates in protecting the confidentiality of a system.

  Bell-Lapadula model [7] characterizes the flow of information by to two mandatory access control (MAC) rules:

  1. The Simple Security Property states that a subject at a given security level may not read an object at a higher security level (no read-up), e.g., a subject with *classified* security level cannot read an object with *secret* security level.
  2. The *-property (read star-property) states that a subject at a given security level must not write to any object at a lower security level (no write-down), e.g., a subject with *secret* level cannot write to an object with a *classified* security level.

- Biba: is a model designed to maintain the integrity of a system. In contrast to the Bell-Lapadula that only address data confidentiality, the goal of this model is to prevent inappropriate modification of data. Biba model [8] defines a set of properties to maintain the integrity of a system.

    1. The Simple Integrity property states that a subject at a given level of integrity can have write access to an object at a lower integrity level (no read down), e.g., a subject with *classified* security level cannot write on an object with a *secret* integrity level.
    2. The *-property states that a subject at a given level of integrity must not write to any object at a higher level of integrity (no write up).

- MLS: A *Multi-level Security* (MLS) mechanism enables users with different levels of security clearance to simultaneously access computer systems. As a result, these security clearances prevent users from obtaining access to information for which they do not own authorization. MLS permits users from a higher clearance level to access lower clearance information. For instance, a user with a clearance level *top secret* can access a *secret* level whereas in the same manner, a user with a *secret* level can access information that owns *classified* clearance level, but not vice-versa. An illustration of this mechanism is shown in Figure 2.5.

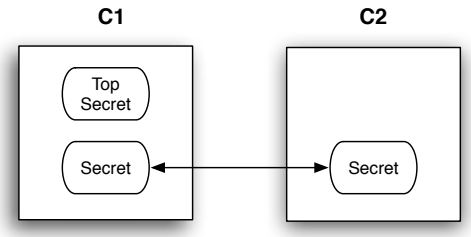


Figure 2.5: A simple MLS network composed of two computers. $C1$ shares information with $C2$ through the secret level compartment.

This MLS mechanism enforces a lattice-based security policy ℓ of security levels,

which has ordering relation $\leq$. Given $x, y \in \ell$, $x \leq y$ means that information may flow from level x to y; e.g., C $\leq$ S $\leq$ T, where C is classified, S is secret, and T is top secret information.

In this last mechanism, a problem arises when different computers share information with the same security level and sensitive information from a higher level "leaks" from one computer to another using this connection; this problem is called the Cascade Vulnerability Problem (CVP).

2.3.2 Cascade Vulnerability Problem (CVP)

The Red book of the National Computer Security Center [40], defines the Cascade Vulnerability Problem as the problem that arises in approved-trusted networks. An approved network is a network such that every computer that belongs to it agrees to own a security assurance level, e.g., MLS. A CVP arises when an intruder takes advantage of the network connectivity to compromise information across a range of sensitivity levels, and the span of accessed levels exceeds the accreditation range of any computer. Let us illustrate a potential CVP in a MLS network in Figure 2.6.

For example, let us consider a simple scenario of a university composed of three main departments: payroll, financial aid, and academic services. We know the payroll department stores records containing high-sensitive information such as social security numbers, dates of birth, amounts of wages, etc. The financial aid department may use information stored on the computer managed by the payroll department, e.g., to check student's bank information. Similarly, the financial aid department may disclose information to the academic department for registration purposes. All these computers are network-connected, which means that there exists a potential cascading vulnerability problem from payroll (a high-sensitivity information computer) to academic services department (a low-sensitivity information computer).

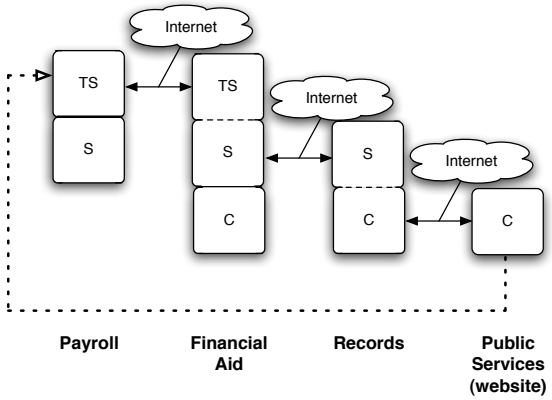


Figure 2.6: A potential CVP. Information from a higher clearance level (*top secret*) leaks to a lower clearance level (*classified*) through this MLS network.

2.3.3 Assurance Levels

The security criteria define a lattice, A, of assurance levels with ordering $\leq$. Given $x, y \in A$, then $x \leq y$ means that a system evaluated at y is no less secure than a system evaluated at x, or alternatively, that an intruder that can compromise a system evaluated at y can compromise a system evaluated at x. Let S define a set of all possible systems. We define $accred : S \rightarrow A$ where *accred(s)* gives the assurance level of system $s \in S$, and is taken to represent the minimum effort required by an intruder to compromise system s.

2.4 Current Approaches

Several approaches have been proposed to solve the cascading vulnerability problem. These approaches attack the problem in different manners, for instance: there have been several approaches proposed to decide whether a network contains a CVP. For example, in [22, 27] detect a single case of a cascade vulnerability path in polynomial time. The approach suggested in [22], was developed to help answer the security question *if two secure systems are connected together, is the resulting system secure?*.

In [25], define an algorithm based in a simulated annealing algorithm to detect and correct the cascading problem. The proposed solution provides a more efficient search

strategy, however, finding the optimal CVP elimination, is in general, NP-complete [28].

In [28], the authors proposed an algorithm to detect and correct the cascading paths in a time complexity of $O(an^3)$, where a is the number of paths that generate a CVP, and n is the number of computers in the computer network. In [31], also proposed an algorithm that requires $O(n^3 log_2 n)$ steps to correct all paths in a network. Once more, the principal shortcoming in these approaches resides that only works for a small number of computers in a network.

In [13], Bistarelli et al. proposed a paradigm shift in the modeling and detecting of the CVP. The authors proposed a soft-constraint-based framework, more specifically in the c-semiring framework, to solve the problem in a MLS network. The results from this approach demonstrated the usefulness of constraint solving as a general purpose modeling technique for security problems.

Subsequent, in [9], the authors carried on the soft-constraint work, and proposed an algorithm to correct this problem. This algorithm consists in determine the minimal number of link cuts necessary to solve the CVP.

This approach, although efficient, overlooks the operational value of providing connectivity. For example, some connections among computers are more valuable than others based on a criterion, e.g., critical impact in a network, electricity, bandwidth, money, besides others. The operational value of providing connectivity is not uniform. For instance, applying [9] provides a solution in eliminating the minimum connections that generate the problem in the network including the more valuable ones. Therefore the solution provided by Bistarelli et al. may lead to choose to remove a link that is essential to providing services.

Chapter 3

Proposed Approaches: Bistarelli's and ours

In this chapter, we present the approach proposed by Bistarelli et al. to model a MLS network in terms of constraints [9] and solve the Cascade Vulnerability problem using soft constraints [13] in a *Multi-Level Security* (MLS) environment. We present an extension to the algorithm of Bistarelli et al. that addresses the shortcomings that we described in the previous chapter.

This chapter is structured as follows: first, an overview about the problem we are solving is presented in Section 3.1. Next, the modeling stage is presented in Section 3.2; e.g., the flows of information within the computer systems and the links among other computers. The modeling part is based on the existing approach suggested by Bistarelli et al. This approach proposes a paradigm shift to solve the CVP using soft constraint programming, where a computer network is modeled with constraints. Then, we explain how to detect the computers that provoke the CVP in a network in Section 3.3. An example explaining the approach of Bistarelli et al. is shown in Section 3.4, and in Section 3.5 we describe the shortcomings of this approach. The key observation is that some connections within a computer network are more valuable than others and Bistarelli's approach overlooks the operational value in the connections at the time to eliminate the CVP. Up to Section 3.4, we present the work proposed by Bistarelli et al. . However, up to this section still containing our contribution for the implementation.

Finally, the rest of Chapter 3 is devoted to describing our proposed approach. This approach consists in extending the current approach for solving the CVP using soft con-

straints, proposed by Bistarelli et al. by assigning values to the computer's connections. By assigning values to the connections within the network, we constrain the elimination process proposed by Bistarelli et al. by considering the operational value of the connections that are generating the CVP. Considering these values, we are interested in optimizing the elimination process by finding the least expensive connections responsible for generating the CVP.

3.1 Definition of the Problem

A Multi-Level Security (MLS) network is a network N of computers $cp_1, \ldots, cp_n$, where each computer may have different *security levels* ℓ. For instance let us consider a small network composed of three computers $cp1$, $cp2$, and $cp3$ as shown in Figure 3.1. Computer cp1 owns a *Top Secret* level, cp2 owns a *Top Secret* and a *Secret* levels, and cp3 a *Secret* and a *Classified* levels.

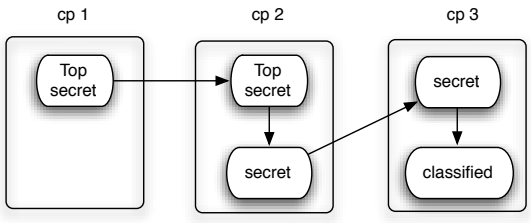


Figure 3.1: A simple MLS network composed of three computers. Each computer has one or more security levels.

Problem

Note that the given connectivity between these three computers, top secret information from cp1 can inappropriately be copied/downgraded from the classified level at cp3. This flow of information is due to the inter-connectivity that cp2 owns between its *top secret and secret* levels.

In order to solve this problem, it is necessary to remove the computer links that are generating this problem. In the example of Figure3.1, it can be either the link that connects cp1 with cp2 or the one connecting cp2 with cp3. The goal is to determine which link has the minimal impact on the network. For this, we propose to assign different values to the links that represent the operational value in the network. By extending the approach suggested by Bistarelli et al. we obtain the minimal number of link removals, and by assigning values (i.e., costs) to the links, we are able to remove the minimal and the least expensive links in a CVP network.

3.2 Modeling links

In this section, we describe what a link is, the different types of links that we can have in a network, and how these links can be represented with constraints.

3.2.1 What is a link?

The connections among computers are established through security levels in each computer. We call these connections *links*. A *link* is a connection from level k in a computer cp_i (l_{i_k}) to level m in computer cp_j (l_{j_m}). Therefore a link is fully described by a 4-tuple: a *source* computer, a *destination* computer, the source's level, and the destination's level.

$$link = (\texttt{source},\ \texttt{destination},\ \texttt{src level},\ \texttt{dest level})$$

For instance in Figure 3.1, we can see the two computers cp1 and cp2 sharing information through a link at the same security level –*top secret level*; cp2 and cp3 share information through the *secret level*. In the same way, cp2 has an inner connection with two security levels – *top secret* and *secret*. In this computer network we count three links:

1. (`cp1`, `cp2`, `top secret`, `top secret`)

2. (`cp2`, `cp2`, `top secret`, `secret`)

3. (`cp2`, `cp3`, `secret`, `secret`)

4. (`cp3`, `cp3`, `secret`, `classified`)

In the MLS environment we deal with connections within the same computer's levels and between different computer's levels.

As we know, a MLS network can have multiple security levels, where information at different levels can be stored in the same computer: e.g., a computer can have multiple users with different levels of administration, i.e., superuser, administrator, user 1, user 2, etc. Therefore it is also important to consider the links that connect these inner security levels. For instance, Figure 3.2 depicts a single computer with three security levels (top secret, secret, and classified), an inner computer link connects these two security levels.

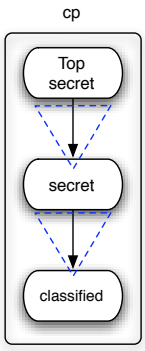


Figure 3.2: A MLS computer. This computer contains two inner security levels.

These links as well as the links shown in Figure 3.1 are classified into three types of links, *valid, invalid,* and *risk*. These links are described in the following subsection.

3.2.2 Types of links

A computer can have inner connections that establish a communication between security levels within the same computer. Also, computers can communicate with other computers

that have the same security level. For instance, a computer with a *top secret* level can transfer information to another computer whose security level is also *top secret*; this is a simple example of a *permitted link*.

Among all the possible links in the network (and within the computer), we have three types of links:

- the links that are permitted and constitute no risk, e.g., a link connecting the same secure levels – e.g., from *top secret* to *top secret*;
- the links that are permitted but also have a degree of risk, i.e., a link connected from a high sensitivity level to a lower sensitivity one – from *top secret* to *secret* level; and
- the links that are not permitted, i.e., the remaining links in the network – e.g., from *top secret* to *classified*.

Now that we know what types of links we have in a network, we need to make sure to keep the links that do not represent a problem to our network. Our goal is to have only *permitted* links in our network, so removing all *invalid links* is crucial. However, among the *permitted* links, we can have the *risk* links, which represent a potential problem in our network. Therefore, it is important to identify what links represent a degree of risk, which is described in the following section.

3.2.3 Constraints on links

Based on the matrix shown in Table 3.1, we can determine the link's classification based on the security level's source and the destination. This table is based on the computer security requirements established for the Department of Defense (DoD) detailed in [39]. In this matrix we can see four different security levels: Top Secret (TS), Secret (S), Classified (C), and Unclassified (U). Based on the description from Table 3.1, let us describe our constraints.

Src\Dest	TS	S	C	U
TS	permitted	risk	invalid	invalid
S		permitted	risk	invalid
C			permitted	risk
U				permitted

Table 3.1: This table shows the relation between the source (src) and destination (dest) levels.

Informal description of the constraints

We are interested in determine the valid, invalid, and the risk links from a network. Therefore, we define two kinds of constraints, c_1 and c_2, that are based on the information provided from Table 3.1.

- Constraint c_1 enforces that the links from the network be are valid. The corresponding values are **0** and **1** if the link is not valid or invalid respectively.

- Constraint c_2 enforces that the links from the network have not risk. The corresponding values are **0** and **1** if the link does not represent a risk or if the link represents a risk respectively.

To determine if the links from the network are valid, we rely on the assurance matrix, where the relation between secure levels is expressed.

For example, let us consider our network depicted in Figure 3.1. The first link (`cp1`, `cp2`, `top secret`, `top secret`). has a source at `top secret` level, and a destination at `top secret` level. Constraint c_1 encodes the values from the matrix, e.g., `matrix(source, destination)`: in this case it will be `matrix(top secret, top secret) = permitted` ($\equiv$ **1**).

Constraint c_2 also encodes the values from the matrix, in the same way, if we take link 2: (`cp2`, `cp2`, `top secret`, `secret`), where source is `top secret` and destination is `secret`, we obtain: `matrix( top secret, secret) = risk` ($\equiv$ **1**).

Formal description

Our goal is to have a network that contains only links that are valid and without risk.

Given a set of links $X = \{x_1, \ldots, x_n\}$, we define two constraints c_1 and c_2, that encode the function `matrix(src, dest)`, which given a link x_i determines if the link is valid, invalid, or represents a degree of risk.

- **Variables and their respective domains:**
  - $x_i \in X, \forall i$, i.e., $x_i = (\mathtt{cp}_{\mathtt{i}_1}, \mathtt{cp}_{\mathtt{i}_2}, \ell_{\mathtt{i}_1}, \ell_{\mathtt{i}_2})$, where: $x_i \in \{0, 1\}$, **1** meaning that x_i exists in the network, **0** that it does not.
- **Constraints**
  - $c_1 : isValid(x_i) \quad \Leftrightarrow \mathtt{matrix}(l_{i_1}, l_{i_2}) \neq \mathtt{invalid}$
  - $c_2 : isnotRisky(x_i) \Leftrightarrow \mathtt{matrix}(l_{i_1}, l_{i_2}) \neq risk$

Example

The solution that we aim to obtain from the above CSP is to have only links that are valid and without risk in the network. Let us illustrate these constraints with an example. For example, consider the network depicted in Figure 3.3. Let us see the transformations of the network when we apply constraints c_1 and c_2.

By applying constraints c_1 and c_2, we are able to remove the invalid and the risky links from the network. However, the resulting network configuration leads to isolate most of the computers from each other, as shown in Figure 3.5.

Observation

It is important to notice that there are two possible scenarios that we can deal with. The first one is to take a network without any connections and build the permitted and risky connections between secure levels. The second scenario is to take an already existing network (with the connections between secure levels), detect and correct the CVP.

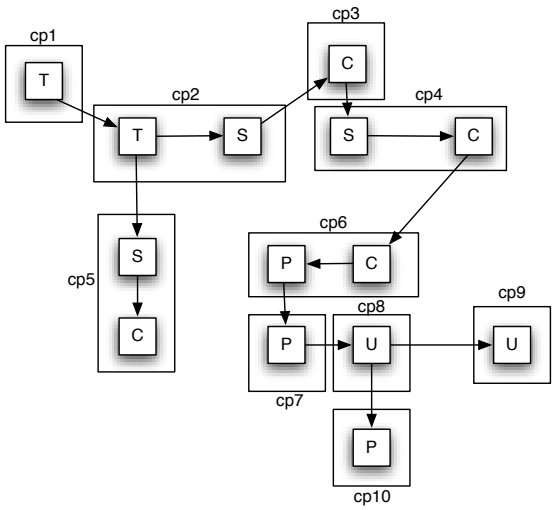


Figure 3.3: Ten computers ($cp_1, \ldots, cp_{10}$). Each computer may contain several security levels (e.g., T, S, C, P, and U).

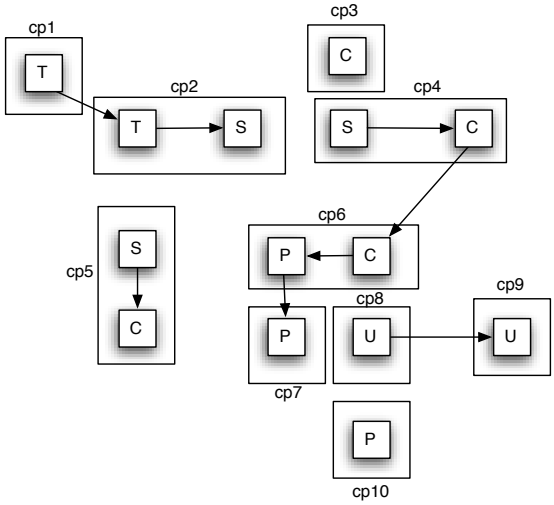


Figure 3.4: After applying c_1 : *isValid()* to all original connections in the network, we eliminate several invalid links.

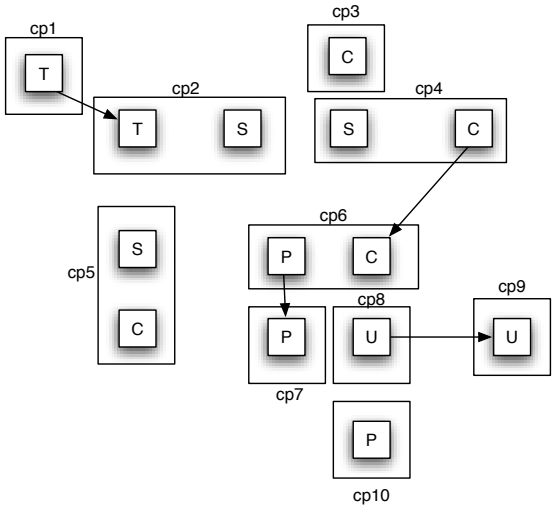


Figure 3.5: After applying c_1 and c_2 : *isnotRisky()* to all the connections, we eliminate all the risk links in the network.

Limitations

If we decide to apply these two constraints, very often we can expect isolated computers (or even perhaps no computers connected at all) in the network, which is unacceptable for a computer network. This means that these constraints are too strong and that we should reconsider the constraints.

However, we are not willing to give up on c_1 since isolating it would make the network totally unsecured. The only constraint that we can reconsider is $c_2 : isnotRisky$, since we can determine and put up with different "degrees of risk" in the links. If we reconsider c_2, we can redefine solutions based on the degree of risk that they involve. To do this, we can rely on several formalisms to solve soft-constraint problems or relaxed constraints.

3.2.4 General description of the soft problem

If we reconsider c_2, we are able to obtain more solutions (at least more satisfiable solutions). However, by considering only links with a degree of risk is limited, since we cannot quantify the risk of the information we have from the links by themselves. Hence we expand the

notion of risk through a set of links (paths). Having a path we can determine whether there is a CVP. This can be determined by comparing the *risk* and the *effort*.

If we reconsider c_2, we can redefine solutions based on the degree of risk that they involve. To do this, we can rely on several formalisms to solve soft-constraint problems or relaxed constraints. One option that seems reasonable is by using the c-semriring framework like it was done in [9, 13].

Informal description

Effort

Informally, the effort of a computer is the amount of time that an intruder takes to break the computer's security. For instance the amount required to defeat a computer protection mechanisms, e.g., firewalls, encryption protocols, intrusion detection systems. An effort of a computer can be evaluated in different terms. One example is the number of dollars or number of hours an intruder must spent in order to obtain unauthorized access to a secure level. For instance, the effort of a work-study to access CS Faculty confidential information from the CS office is smaller than access confidential information from the office of the VP affairs.

In the point of view of the intruder, the cost/reward that the intruder shall receive must be greater than the effort that will spent defeating the security system. In a MLS environment, a possible way to calculate the effort is by gaining secure levels from one computer to another. For instance, The TCSEC, Computer security requirements [39], define a *minimum criteria class*, which represents the minimum effort required in order to obtain clearance to access a higher secure level. For instance, a computer has two security levels, top secret and secret, and has some users who are cleared to only read level secret but not level top secret. In order that users access top secret information, the user must gain an accredited class of B2 (see Table 3.2 for further details).

		Maximum Data Sensitivity						
		U	N	C	S	TS	1C	1M
Min. Clearance of System Users	U	C1	B1	B2	B3	*	*	*
	N	C1	C2	B2	B2	A1	*	*
	C	C1	C2	C2	B1	B3	A1	*
	S	C1	C2	C2	C2	B2	B3	A1
	TS(BI)	C1	C2	C2	C2	C2	B2	B3
	TS(SBI)	C1	C2	C2	C2	C2	B1	B2
	1C	C1	C2	C2	C2	C2	C22	B13
	MC	C1	C2	C2	C2	C2	C22	C22

Table 3.2: Security index matrix for open security environments

Risk

The Security evaluation criteria defined a function *risk*, which defines the minimum acceptable risk of compromise a security level ℓ to security level ℓ'; it represents the minimum acceptable effort required to "compromise security" and copy/downgrade information from security level ℓ to security level ℓ'.

This matrix encodes the assurance matrix as shown in Table 3.5: So for instance the risk obtained from a link between a *classified* to a *top secret* level is **0**; the risk obtained from a *secret* to a *classified* level is **1**, and so forth. Further details about the assurance matrix can me found in Appendix A.

The assurance matrix

As we mentioned in Sectionthe security evaluation criteria matrix defines the minimum acceptable risk of compromising security levels; meaning that it represents the minimum acceptable effort required to "compromise security" and copy/downgrade information from one level to another. This matrix encodes the assurance matrix as shown in Table 3.5. So, for instance, the risk generated by a link between a *classified* level to a *top secret* level is **3**; the risk generated by a *secret* to a *classified* level is **1**, and so forth. Further details about the assurance matrix can me found in Appendix A.

Minimum User Clearance	Rating R_{min}
Unclassified	0
Not Classified	1
Confidential	2
Secret	3
Top Secret	5

Table 3.3: Rating scale for minimum user clearance (R_{min})

Max data sensitivity with categories	Rating R_{max}
Unclassified	0
Not Classified	1
Confidential	2
Secret	3
Top Secret	5

Table 3.4: Rating scale for maximum data sensitivity (R_{max})

src\des	U	N	C	S	TS
U	0	0	0	0	0
N	1	0	0	0	0
C	2	1	0	0	0
S	3	2	1	0	0
TS	5	4	3	2	0

Table 3.5: Assurance matrix: summarize the risk indices corresponding to the various associations of clearance levels.

U = Uncleared or Unclassified
N = Not Cleared but Authorized Access to Substantive Unclassified information
C = Confidential
S = Secret
TS = Top Secret

Risk and Effort in Paths π

In a computer network, the notion of path is the idea that an intruder takes advantage of the network connectivity and "moves" from one system to another conducting attacks on protected mechanisms.

In order to find a cascade vulnerability problem in a network, we need to extend the notion of risk and effort to a set of links that are connected with each other, this is what we call a *path*. A path π is composed of a set of links $\{l_1, \ldots, l_n\}$ that are connected with each other from a computer source (which owns the highest sensitivity information, e.g., top secret) to a destination computer (which owns the minimal sensitivity information, e.g., unclassified).

The soft CSP: formal description

To determine the existence of a CVP we need to compare the risk required to compromise the network with the effort of compromising the computer system as a whole. For that, we calculate the aggregated risk from the links that form a path and we compare it with the effort from this path. We introduce two functions $\texttt{risk}(l_1, \ldots, l_n)$ and $\texttt{effort}(l_1, \ldots, l_n)$, if the risk associated with the links given as an input in function `risk` exceeds the effort in function `effort` means that a CVP exists.

Given a set of links interconnected with each other (path π) from a source computer cp_{src} to a destination computer cp_{dest}, where cp_{src} and $cp_{dest} \in \pi$, determine if the risk $\mathbf{r}$ exceeds the effort $\mathbf{e}$ from π.

- **Variables and their respective domains:**
  - $r \in \mathbb{N}$: the is the corresponding value obtained from executing $\texttt{risk}(\pi)$
  - $e \in \mathbb{N}$: the is the corresponding value obtained from executing $\texttt{effort}(\pi)$

- **Constraints**
  - $c_1 : isValid(x_i)$, for all links in path π
  - c_3: $r > e$

3.3 Detecting a CVP in a network

We have seen how to model the computers' links in a network. Also, we have seen that in occasions the constraints can be limited, that is why we need to reconsider some of them in order to obtain satisfactory results.

In last section, we also noticed that the links are meaningless when we want to calculate the risk in a given network. That is why we expand the notion of risk through a set of links that are called paths. In this section, we focus on detecting the set of links that are generating a CVP. In order to detect a CVP, we need to consider the paths that connect one specific computer to the rest of them.

3.3.1 How to determine whether a network has a CVP?

To determine whether there is a cascade vulnerability problem, we need to compare the *effort* required to compromise the network against the *risk* of compromising the computer as a whole.

The risk represents the difference between the minimum clearance or authorization of computer users and the maximum sensitivity (e.g., classification and categories) of data processed by a computer. These risk values are obtained from Table 3.4 and Table 3.3.

3.4 Example

For example, let us consider the following MLS computer network.

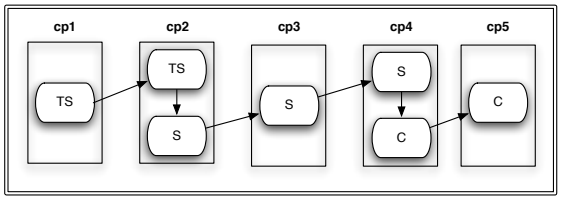


Figure 3.6: A network with a potential CVP.

Figure 3.6 depicts a computer network with a potential CVP. The first step is to model this computer network in terms of links between the secuity levels to determine only valid links. We apply the approach proposed in [13] to model the computer network. This model is depicted in Figure 3.7.

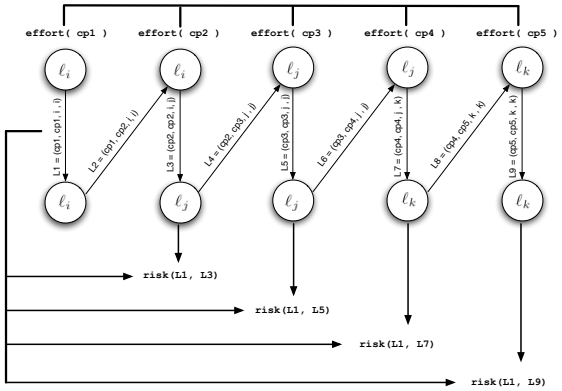


Figure 3.7: Network modeled based on [13].

In this network we have five computer: $cp1, cp2, cp3, cp4$, and $cp5$, with three different secure levels: TS, S, and C, that are modeled as ℓ_i, ℓ_j, and ℓ_k respectively. The links established between secure levels within and between computers are the following:

1. $\texttt{link}(\texttt{cp1}, \texttt{cp1}, \texttt{TS}, \texttt{TS})$ \\ inner link
2. $\texttt{link}(\texttt{cp1}, \texttt{cp2}, \texttt{TS}, \texttt{TS})$
3. $\texttt{link}(\texttt{cp2}, \texttt{cp2}, \texttt{TS}, \texttt{S})$ \\ inner link
4. $\texttt{link}(\texttt{cp2}, \texttt{cp3}, \texttt{S}, \texttt{S})$

5. `link(cp3, cp3, S, S)` \\ inner link

6. `link(cp3, cp4, S, S)`

7. `link(cp4, cp4, S, C)` \\ inner link

8. `link(cp4, cp5, C, C)`

9. `link(cp5, cp5, C, C)` \\ inner link

In this particular case, all connections are considered to be valid, since the secure levels are properly connected at the same/lower secure level, e.g., **TS** and **TS**, **TS** and **S**, and so forth.

Finding cascading paths

Although we have valid connections in the network, we also have links that constitutes risk. The second step is to find these links. Let us do so by calculating their respective risk and effort as shown in Figure 3.8.

Now, let us calculate and compare the **risk** and **effort** from the $\pi = \{l_1, l_2, l_3, l_4, l_5\}$

$risk(TS, TS) = \mathbf{0}$

$risk(TS, TS) = \mathbf{0} \quad risk(S, S) = \mathbf{0}$

$risk(TS, S) = \mathbf{2} \quad risk(S, C) = \mathbf{1}$

$risk(S, S) = \mathbf{0} \quad risk(C, C) = \mathbf{0}$

$risk(S, S) = \mathbf{0} \quad risk(C, C) = \mathbf{0}$

After comparing the risk and effort we obtain the following set of links: $\pi_1 = \{l_1, l_2, l_3\}$

$\pi_1 = \{l_3, l_4, l_5, l_6\}$

$\pi_1 = \{l_7, l_8, l_9\}$

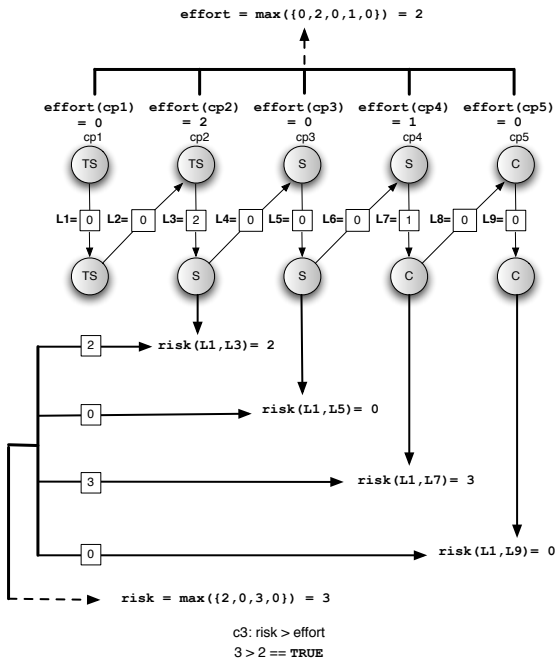


Figure 3.8: Network modeled based on [13].

Remove the minimum number of links that provoke the CVP

Once we obtain the set of links that generate the CVP, we need an algorithm to find the minimal conflict sets of links. Here we apply the Bistarelli et al. algorithm based on the *Minimal Hitting Set* (MHS) – an algorithm used to select an approximation of a minimal set of removed links. The algorithm is as follows:

1. Maintain a counter for each link involved in the set of cascading path generators that need to be removed.

2. Remove the most common link (the link with the highest counter), thus removing all cascading path generators involving that link; in case of a tie a random one is selected.

3. Update the link counters built in Step 1 to reflect the effect of reducing the set of cascading path generators that we need to consider.

4. Continue removing links and updating the link counters until all cascading paths have been removed.

Observations

After applying the approach proposed by Bistarelli et al. for modeling [13], and to detect the links that case the CVP [9], we obtain that the potential solutions, (i.e., the candidates links to remove from the network) are l_2, l_4, and l_5.

Although, the work in [13, 9] provide a transparent explanation how to soft a CSP for the CVP, it was necessary to form a better explanation for implementation purposes, since no implementation was provided before. Also, our contribution consists in extending the detection and elimination of the CVP in networks that are not limited to "linear shapes" (e.g., see Figurefig:acvp2). Up to Section 3.4, we present the work proposed by Bistarelli et al. . However, these past sections still containing our contribution for the implementation.

3.5 What if we take weights in links into account?

The results provided by Bistarelli et al. ensures the "minimum number of cuts" when eliminating the CVP. However, these results may lead to choose to remove a link that is essential to providing services as mentioned in Section 3.4.

We notice that cutting network's links we eliminate potential benefits from the network. Clearly another approach is needed to prevent information leakage.

By assigning values to the connections within the network, we consider the operational value in the connections that are generating the CVP. In this way, we constrain the elimination process proposed by Bistarelli et al.

Considering these values, we are interested in finding the least expensive connections to eliminate in the network that are causing the CVP. The process of finding the best solutions among the possible ones considering constraints is called *constrained optimization problem*. For instance, by considering different values of the links, e.g., critical, marginally valuable,

Type	Value
Critical	weight < 30
Marginal Valuable	$20 <$ weight < 30
Not Valuable	weight < 20

Table 3.6: Three different types of links and a corresponding value assigned

and not valuable, at the time of eliminating the links, we remove the ones that generates the lowest cost in the network.

3.5.1 Optimizing the links removal subject to constraints

In the process of solving the CVP using our approach, we face two situations:

1. Reduce the risk on the network: The purpose is to reduce the risk on the network. This can be done by applying constraint c_2, which removes the minimal number of links that generate risk. However, by satisfying c_2 we might end up with results that lead to an unsatisfiable solution. Hence, we are willing to accept a degree of risk in order to obtain solutions.

2. Reduce the cost of the removals: The links that have a degree of risk also have an operational value (or cost). We are also interested that when we eliminate the minimal number of links that generate the CVP, we also remove the links that generate the lowest cost in the network, e.g., the least expensive ones. Therefore, we need an objective function that not only minimizes the links removals but also the cost of the removals.

Relaxing constraints

In our earlier example mentioned in Chapter 1, Section 1, we discussed that an alternative to eliminate links with a degree of risk is to classified them with different values according to their impact in the network. For instance, if we have three different links, we can assign values as shown in Table 3.6.

Weighted links: what constitutes a weighted link?

We extend the definition of a link by incorporating a *weight* to the corresponding link, that represents the operational value between computers.

$$(\texttt{source}, \texttt{destination}, \texttt{src level}, \texttt{dest level}, \texttt{w}_\texttt{i})$$

For instance, Figure 3.6 depicts a MLS computer network, by incorporating weights on the links we deal with a network that looks like Figure 3.9.

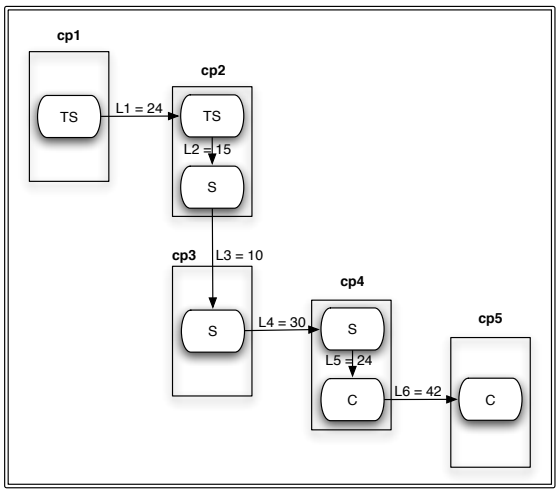


Figure 3.9: A MLS network with weights on the links. In this network the weights represent the operational value, e.g., L_6 has the highest operational value with 42 units.

3.5.2 An algorithm that optimizes the CVP cuts

In this section we present an algorithm based on the Minimal Weight Hitting Set theory, which goal is to minimize the results obtained from Bistarelli et al. approach. The goal is to identify the links that generate the CVP but have a minimal impact in the network.

The Minimum Weight Hitting Set: An optimization algorithm

Our approach to solving the Cascade Vulnerability Problem is based on the theory of *minimum weighted hitting set* (MWHS). The MWHS algorithm is applied on optimization problems that model resource-selection problems [18].

Formal definition

The MWHS takes as an input a *set system*, (i.e., a total number of links) (U, S) with $\bigcup_{s \in S} S = U$, and weights $c : S \to \mathbb{R}^+$, where U where U is a non-empty finite set, and S a family of subsets of U [29].

The objective is to find a *minimum weighted hitting set* of (U, S), i.e., a subfamily $R \subseteq S$ where R is the solution of the following constrained optimization problem:

$$\begin{cases} \min \ \sum_{r \in R} c_i \\ \text{s. t. } \bigcup_{r \in R} R = U \\ \text{and } \bigcup_{r \in R} S_r = U \end{cases}$$

- As an input, we have:

  Universe $U = \{s_1, s_2, \ldots, s_n\}$,

  Subsets $S = \{S_1, S_2, \ldots, S_k\}$,

  Weights $W = \{w_1, w_2, \ldots, w_k\}$

- The goal is:

  To find a set $I \subset \{1, 2, \ldots, n\}$ such that:

$$\begin{cases} I \text{ minimizes } \sum_{i \in I} c_i \\ \bigcup_{i \in I} S_i = U \end{cases}$$

3.5.3 Algorithms

In this section we present the main algorithm Minimum Weight Hitting Set (MWHS), and a procedure that finds the least expensive element among a set of links. Since the following algorithms are based on set theory, we consider the total number of links in the network and the minimum number of links as a set.

Algorithm 1 Find the Link with the Highest Counter

`Given:`
$counters \Leftarrow \{v_1, \ldots, v_n\}$ // number of occurrences v_i that $\mathbf{l_i}$ appears in π_k
`Return:`
The most common link (the link with the highest counter), in case of a tie a random one is selected.

$tmp \Leftarrow values[0]$
$pos \leftarrow 0$;
for all $val_i \in counters$ **do**
  if $values[i] > tmp$ **then**
    $tmp \Leftarrow values[i]$
    $pos \Leftarrow i$
  end if
end for
return pos

Algorithm 2 Minimal Weighted Hitting Set

```
INPUT:
    links ⇐ {l_1, ..., l_n}     // set of links in the network
        P ⇐ {π_1, ..., π_k}     // set of paths in the network
 counters ⇐ {v_1, ..., v_n}     // number of occurrences that l_i appears in π_k
  weights ⇐ {w_1, ..., w_n}     // w_i is the corresponding weight for l_i
OUTPUT:
weighted_hitting_set containing the least expensive (and minimal) links removals.

Process:
 counter ⇐ updateTheCounter(P) // Initialize the counter
hitting_set ⇐ ∅                // minimal removals set
while counter > 0 do
  min{} ⇐ findAllHighest( counters ) // find the set of links with the highest counter
  % Find the minimal cost from all the elements in min
  for all π_i ∈ P do
    if π_i contains (links[ pos ]) then
      π_i − links[pos]          // remove links[ pos ] from π_i
      counters ⇐ updatecounters( counters , pos )
    end if
  end for
  updateTheCounter(S)
end while
return weighted_hitting_set
```

Chapter 4

CVP Tool Simulator

In this chapter, a cascade vulnerability problem simulator (CVP Simulator) is presented. The CVP Simulator is an implementation of the algorithms proposed in Bistarelli et al. approaches [9, 13] and the algorithm that we are proposing in this thesis.

A description of the features and functionalities of the tool is presented in Section 4.1. A detailed tool's architecture design is presented in Section 4.2, and finally, a description of the tool's components is mentioned in Section 4.3. A user manual describing the main functionality of this tool can be found in Appendix D of this thesis.

4.1 CVP Tool Simulator

We present the CVP Simulator Tool, which is a Java-based application that detects if it exists a cascade vulnerability problem in a network, identifies the elements that are generating this problem, and properly eliminates the minimum elements that cause this problem in the network. The objective of building this tool is to apply the work presented in the approaches mentioned above.

The tool takes advantage of Java's operating system portability, object oriented manipulation, and graphical libraries. We mainly take advantage of Java's portability because it enables us to run this tool on any operating system environment as its slogan says *"Write once, run anywhere"*[1].

The computers, links, and paths can be represented as Java's objects, which facilitate us the manipulation of their fields and references to other objects. Finally, we use Java to

[1]Sun Microsystems: http://www.java.sun.com

render the graphical interface to enable users to interact with this tool through a network simulation that displays the computers in the network and the weighted connections between them. An advantage of using Java among other OOP languages, is that currently is a programming language very popular and with high demanded in industry[**TODO**[2]].

4.1.1 Solving constraints with SWI-Prolog

Constraint Solving is a generalization of Constraint Logic Programming (CLP), which extended Logic Programming, by embedding constraints (i.e., no longer predicates) into a logic program such as Prolog. Constraints bring important features such as logical variables and backtracking and other solving algorithms s.a. propagation. It is natural to use a constraint logic programming paradigm to describe problem's constraints and to solve our particular problem.

The constraint solver is a Prolog program that processes the constraints that are displayed on the interface, and returns a set of all possible solutions that satisfy the constraints.

4.1.2 Features/Functionalities

The main features and functionalities of the CVP simulator tool presents are:

- creates computers and networks,
- establishes links between computers,
- assigns weights to the links,
- detects the existence of a CVP in a network,
- extracts the cascading path generators (ρ) from the network in case there is a CVP,
- applies optimization algorithms,
- identifies the least expensive connections to eliminate in a network.

[2]TODO:reference needed

4.2 Architecture Design

The tool is an example of a *model-view-controller* (MVC) architectural pattern. The model is the representation of the algorithms used, e.g., that algorithms that detect and eliminate the CVP. The viewer component is composed of multiple GUIs that interact with the user, and the controller component handles the interaction between GUI and model. The MVC pattern is shown in Figure 4.1, and Figure 4.2 shows how the tool components interact with each other.

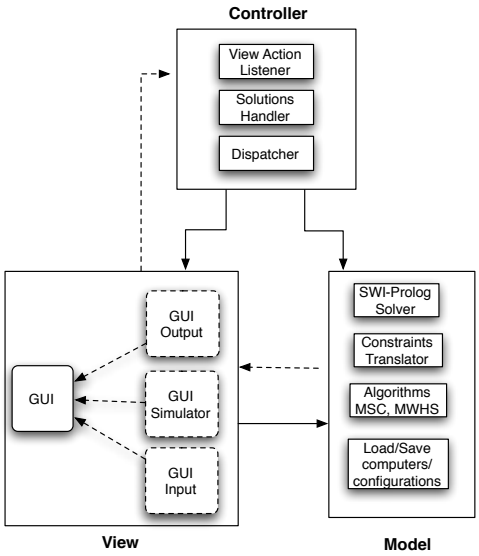


Figure 4.1: The model-view-controller pattern for the CVP simulator tool

4.3 The CVP Simulator Tool and its Components

This section presents the components of the CVP Simulator Tool.

- *Customize panel:* enables the user to add computers into the network and specify the type of secure information it owns. A text field to input a description of the computer is available. In a future version of this tool, it is desired to assign a value to the computer, i.e., the user shall know how valuable is this computer among the

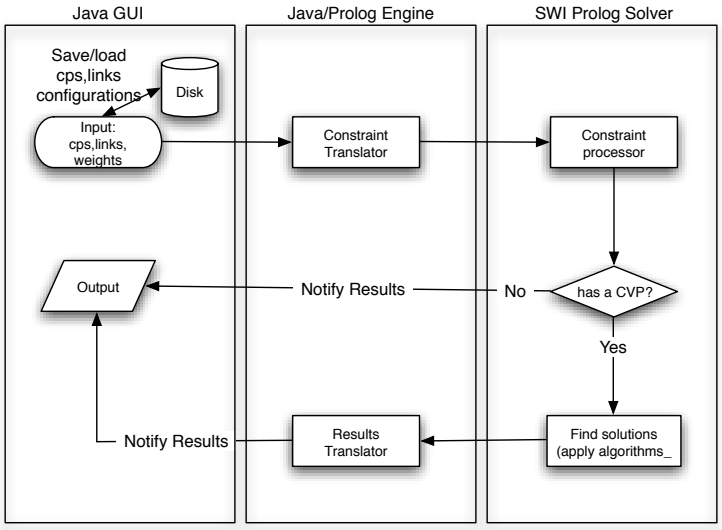


Figure 4.2: The CVP simulator tool architecture design

computers in the network, for that, a value bar is available in this panel as well. In the customize panel, the user can add the computer manually (by adding values to the fields mentioned above) or by loading an existing network file by pressing the **load** button. An example of the Customize Panel is shown in Figure 4.3.

- *Configuration panel:* enables the creation of links between computers and assign a value (weight) to that connection. The user shall add a link manually i.e., by specifying the source and destination secure level and adding a weight to that link, or by pressing the **load configuration** button. Also, in this panel we count with the feature of detecting which links are permitted, have a degree of risk, and are not permitted. This is done by satisfying the constraints stated in Chapter 3 in Section???. The output of satisfying the constraints is displayed in the text field named "Constraint Checker." This can be done by pressing the **Check Constraints** button. The resulting "Paths" displayed in the *paths field text*, are the resulting cascading paths after satisfying the constraints mentioned in Chapter 3 in Section???.Finally, the user is allowed to remove an existing computer or an existing link by pressing the

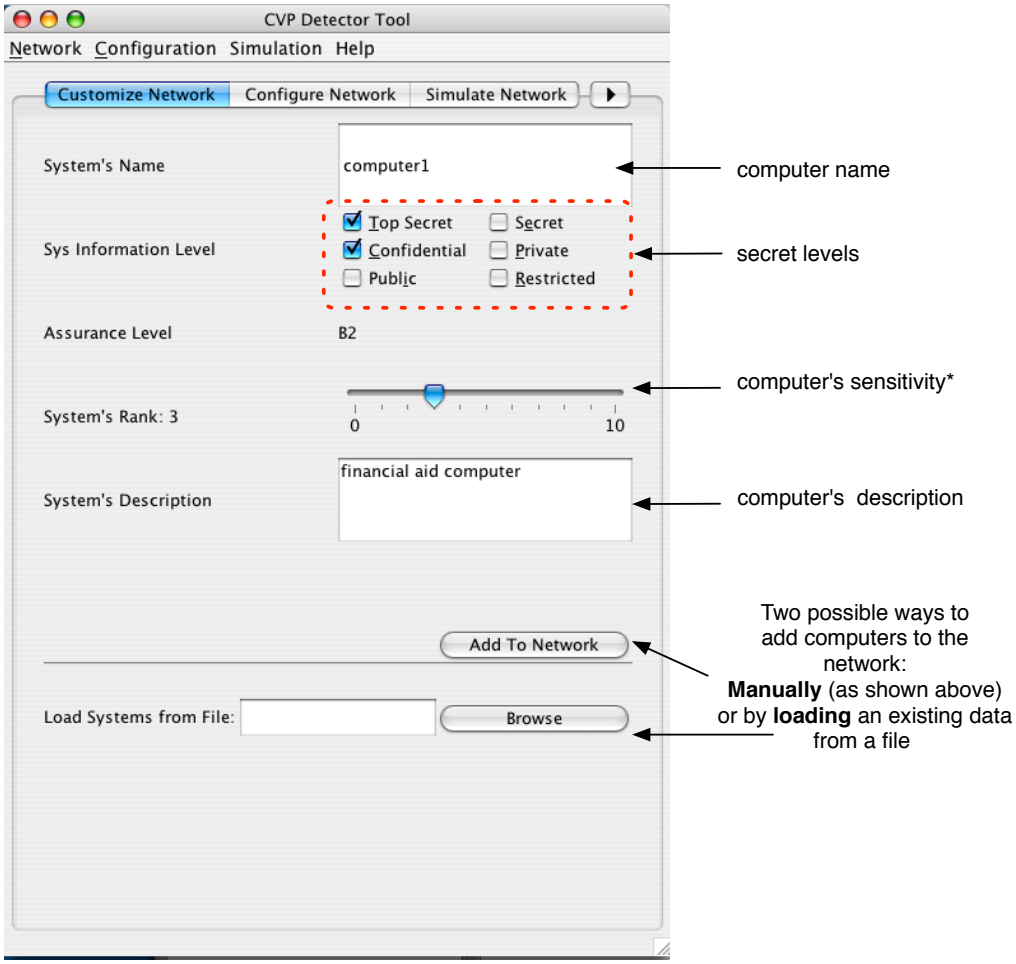


Figure 4.3: Customize panel: enables adding a new computer to the network. *Feature not available in this version.

Remove System and/or the **Remove Connection** buttons. An example of the Configure Panel is shown in Figure 4.4.

- *Simulation panel:* shows the tracing of the algorithm in the network, and provides the possible solutions. The algorithm is selected by the user (only three algorithms have been implemented in this first version). An example of the Simulation Panel is shown in Figure 4.5.

- *Constraint translator:* extracts the computer information that is added from the customize panel and the connections established in the configuration panel, and generates a SWI-Prolog file with the information extracted from these panels.

- *SWI-Prolog solver:* invokes SWI-Prolog and consults the file generated by the constraint translation component. This solver then processes the constraints to detect a CVP in the network. In the case that the detection constraints are satisfied (that indeed there exists a CVP in the network), the solver finds all the paths that are causing this cascading problem.

- *Solution handler:* opens a listener stream to the SWI-Prolog solver, and it stores all the output from the SWI-Prolog solver until it finishes to process all the constraints, then gives the results to the simulation panel.

- *Algorithm selection:* This component enables users to select the algorithm that will be applied in the elimination process of elements that generate the CVP. The chosen algorithm is used in the simulator and the results are displayed in the simulator panel. The algorithms in this first version of the application are implemented in Java, and their APIs are provided in case the user would like to incorporate another algorithm into the tool.

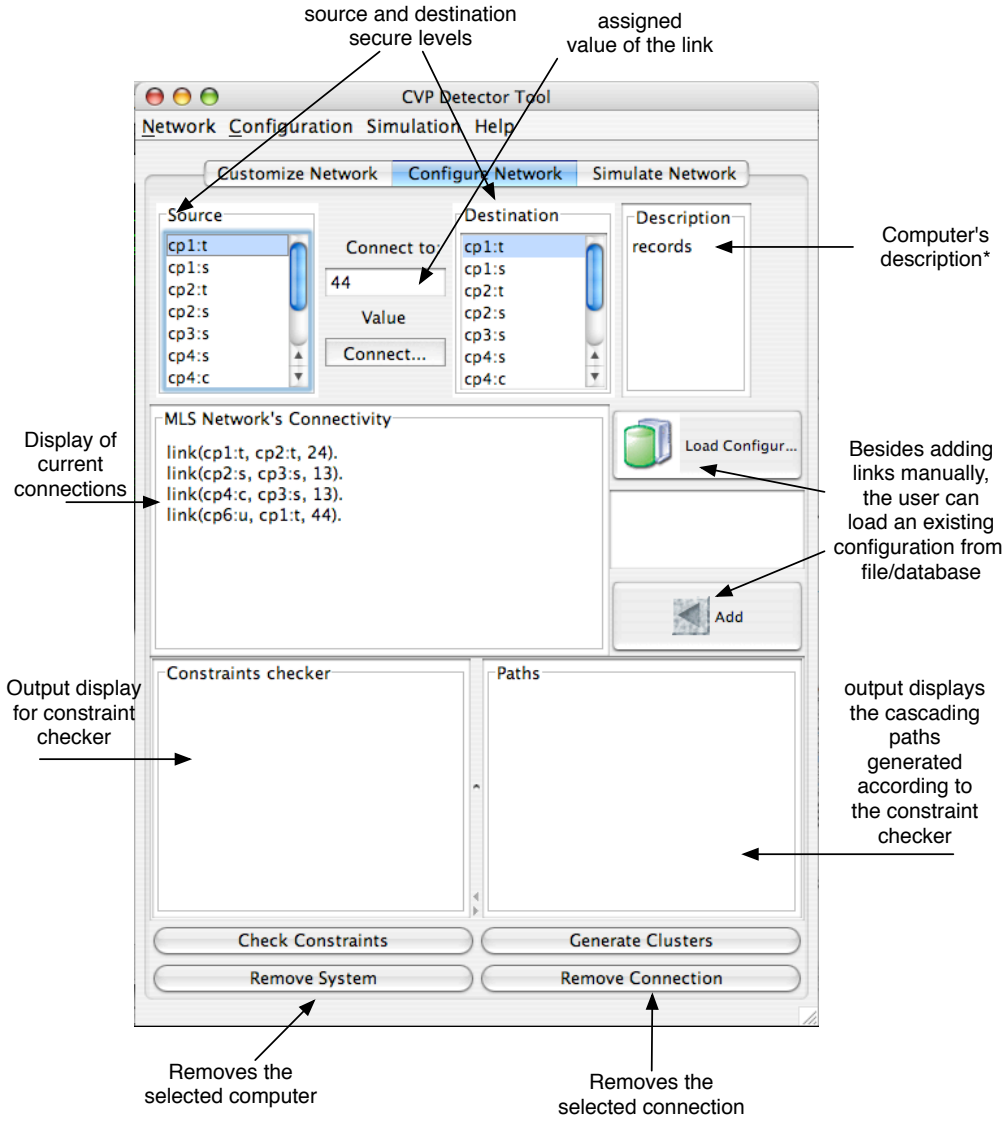


Figure 4.4: Configure panel: Allows to establish connections between computers. The functionalities in this panel includes the *constraints checker* to ensure that link's constraints are satisfied. *Feature not available in this version.

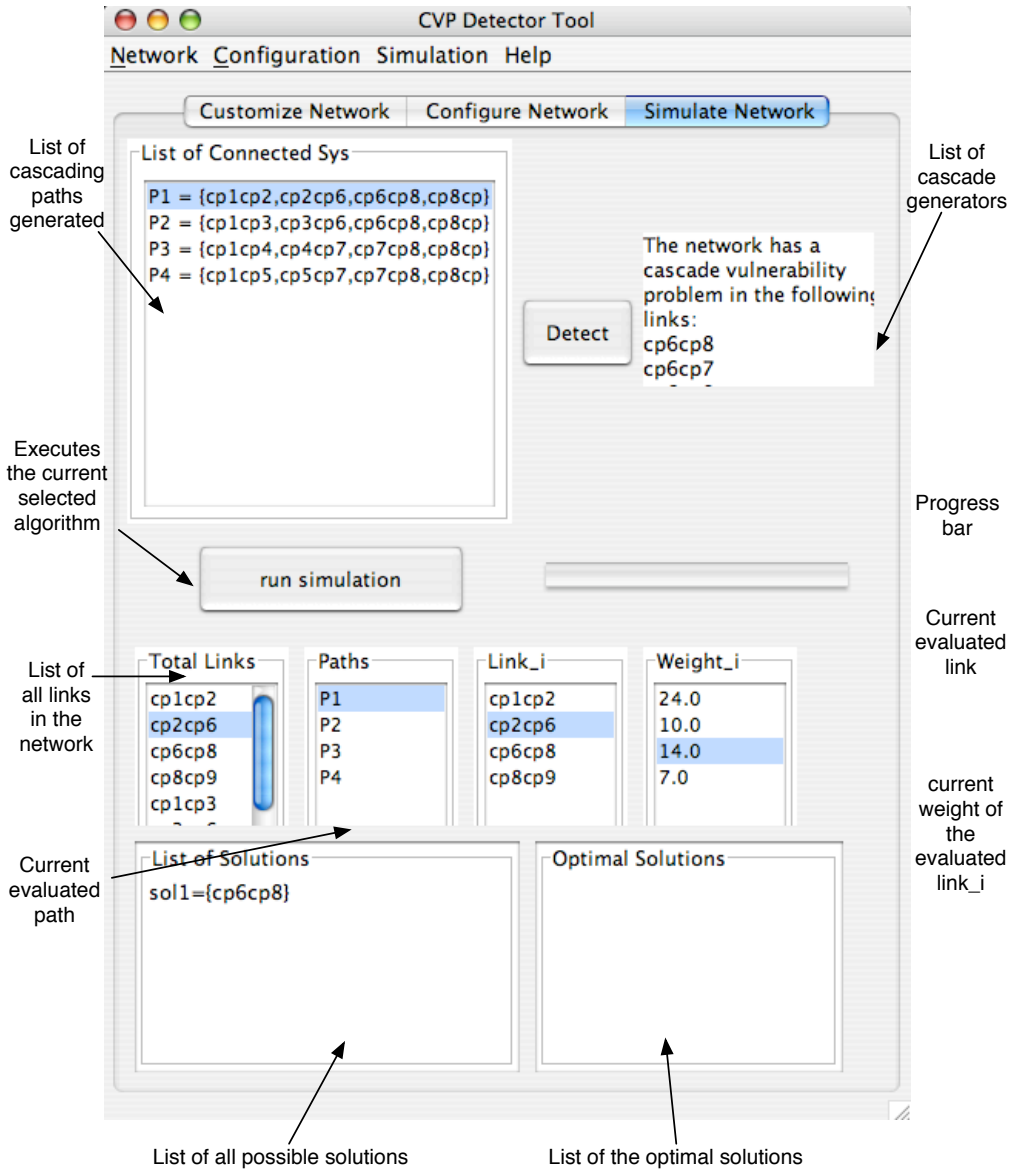


Figure 4.5: Simulate panel: simulates the specified algorithm to eliminate the CVP.

4.3.1 Constraint translator

This component is in charge of extracting the *facts* and *constraints* from a given network. Consider the network from Figure ??.The network is composed of 5 computer with different security:

cp_1, owns a *top-secret* level;

cp_2, owns a *top-secret, secret*, and *classified* levels;

cp_3, owns a *secret* and *classified* levels;

cp_4, owns a *classified* level;

cp_5, owns a *secret* component.

these facts are translated and represented in Prolog facts called **computer** as follows:

`computer(`cp_1`, [ t ] ).`

`computer(`cp_2`, [ t, s, c ] ).`

`computer(`cp_3`, [ t, s ] ).`

`computer(`cp_4`, [ c ] ).`

In the same manner, the facts that represent the links between computers:

cp_1 is connected to cp_2 through the top-secret level, and this connection has a value of 42 Gb/s,

cp_2 is connected to cp_3 through the secret level, and this connection has a value of 13 Gb/s, and to C5 with a 2 Gb/s value,

cp_3 is connected to cp_4 through the classified level, and this connection has a value of 24 Gb/s,

are also represented as follows:

```
link(cp1 , cp2, [t,t], 42).
link(cp2 , cp3, [s,s], 13).
link(cp2 , cp5, [s,s], 2).
link(cp3 , cp4, [c,c], 24).
```

In Prolog words: the rule **link** takes two computer objects, a *computer source* and a *computer destination*, and returns all the links that connect computer source with computer destination. These facts are written in a Prolog file and then passed to the *Prolog solver* to solve the constraints.

4.3.2 Prolog solver

This solver is a Prolog file executed by SWI-Prolog that takes as an input the file generated in the previous section and returns a set of paths ρ that are the least expensive links in the network to eliminate in order to avoid a CVP.

The entire procedure of the Prolog solver is demonstrated in the *Simulation Panel*. In this panel step is shown step by step how the algorithm is working, and the resulting set is displayed in the *Result text area*. Several procedures are described in detail in Figure 4.3.2, and Figure 4.3.2.

- **is valid flow:** This predicate takes as a parameters two computers: X and Y. The predicate will check if the flow between X through Y is valid or not. *is_valid_flow* will compare the *source* and *destination* secure levels from the current link.

- **is risk free:** This predicate is called from *find_all_risk-free()* predicate, which given a set of valid links, must find all the links that are risk-free. The links that are not considered risk-free, are removed from the main list. The *risk_flow* predicate, checks from the stated rules if it is a risk between computer X and Y.

4.3.3 Parser

After the *Prolog Solver* component, finds the solutions, it is necessary to retrieve this solutions in a "User's terminology". The **Parser** is a Java component that retrieves the Prolog's solutions and associates the prolog's values with the current values stored in the application, i.e., the links resulting from the prolog's engine must be mapped with the

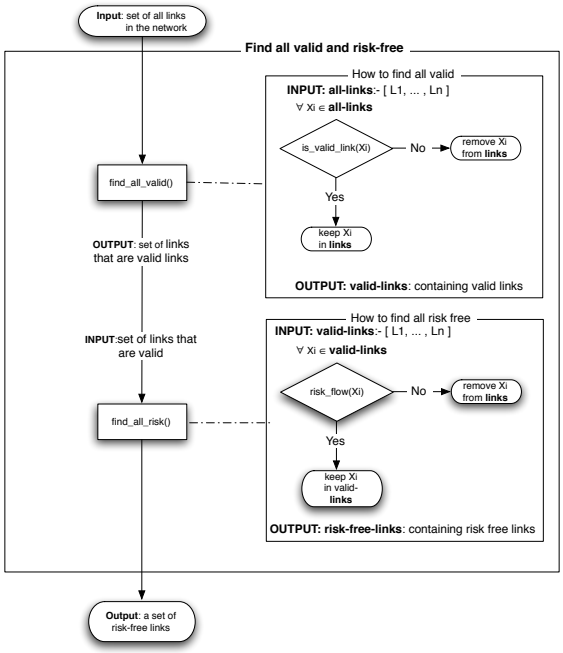


Figure 4.6: The program flow about how to find the links in the network that are valid and risk-free.

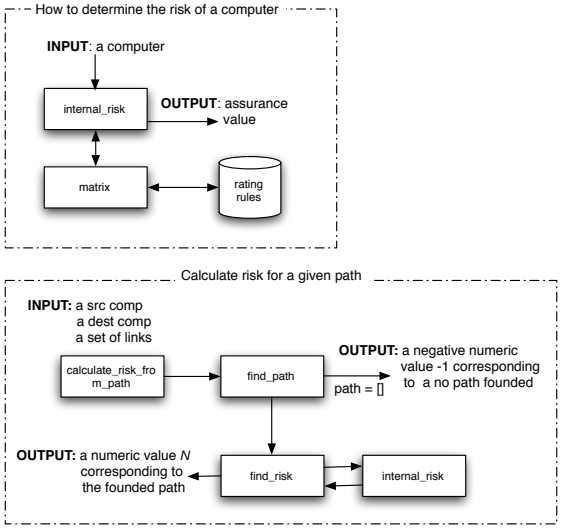


Figure 4.7: Detailed component of the program

links provided by the user. For example, after detecting the CVP in a given network, the prolog's result is the following:

is parsed as follows:

```
P1 = {cp1cp2, cp2cp7, cp7cp8, cp8cp11};
P2 = {cp1cp3, cp3cp7, cp7cp8, cp8cp11};
P3 = {cp1cp4, cp4cp7, cp7cp8, cp8cp11};
P4 = {cp1cp5, cp5cp7, cp7cp8, cp8cp11};
P5 = {cp1cp6, cp6cp7, cp7cp8, cp8cp11};
```

where the value of link 'cp1cp2' is **10**, 'cp8cp11' is **25**, and so forth.

Chapter 5

Experimental Results / Examples

In this chapter, we present three examples to illustrate the algorithm proposed in Chapter 3. We present a traditional *linear shape* example in Example 1. A more complicated network, where the maximum sensitivity information leaks from a single computer's compartment to three compartments with a low sensitivity security is presented in Example 2. Finally, a third example with 10 computers' compartments is presented in Example 3.

Example 1: A network without weights (e.g. Bistarelli [10])

Let us consider the multilevel secure computers $cp1, cp2, cp3, cp4$, and $cp5$, holding information at security levels e, f, g, h, i, j and k as depicted in Figure 5.1.

The assurance matrix corresponding to these secure levels is shown in Table 5.

src\dest	e	f	g	h	i	j
e	0	1	1	2	2	4
f	0	0	1	1	2	4
g	0	0	0	1	1	2
h	0	0	0	0	1	1
i	0	0	0	0	0	1
j	0	0	0	0	0	0

Table 5.1: The assurance matrix for secure levels `e, f, g, h, i,` and `j`

By applying the algorithm proposed in [9], we notice that the network has a cascade vulnerability problem, and the resulting cascading paths that generate this problem are:

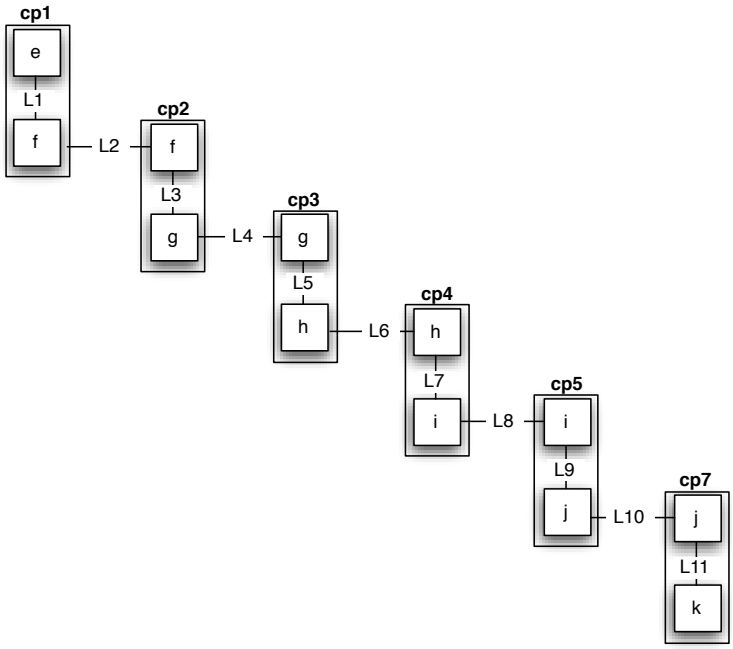


Figure 5.1: Network configuration # 1 (motivated example from [10])

- P = [L_2, L_6, L_{10}]
- P = [L_4, L_8]

We are interested in removing the minimal links that generate the CVP. By applying the MHS, the minimal links are: $\{L_4, L_8\}$ or $\{L_2, L_6, L_{10}\}$

Example 1.a: A network with weights

Now let us consider the same configuration in Figure 5.1 with weights on links. Notice that the answer of selecting the minimum link removal is different considering values in the links.

$L_1 = 50$ units;
$L_3 = 25$ units;
$L_5 = 25$ units;
$L_7 = 10$ units;

$L_9 = 20$ units;

Total: 130 units

By considering the cost in the links, we calculate that the total operational value of the network is 130 units. If we take the results obtained in previous example we obtain the following:

Links	cost	final network value
L_2, L_6, L_{10}	95 units	35 units
L_4, L_8	35 units	95 units

We notice that between the two results, the least expensive link removal is [L_4, L_8] with 35 units. The operational value (after the elimination of these links) is 95 units.

Example 2

In the next particular example, depicted in Figure 5.2, the network contains 12 computers, and 14 links, where $cp1$ owns the highest sensitivity level and the lower sensitivity levels are stored in computers $cp10, cp11$, and $cp12$.

The corresponding values for the links, and the frequencies, i.e., the number of occurrences of a specific link on a cascading path[1], are:

link	cost (units)	frequency
L_1	5	0
L_2	5	0
L_3	5	0
L_4	5	3
L_5	10	3
L_6	20	3
L_7	20	3
L_8	30	3
L_9	15	4
L_{10}	35	4
L_{11}	45	4
L_{12}	25	4
L_{13}	5	4
L_{14}	5	4
Total	240 units	

The paths and the resulting cascading paths are:

[1]The number of link's occurrences is needed for the MHS and the MWHS algorithms

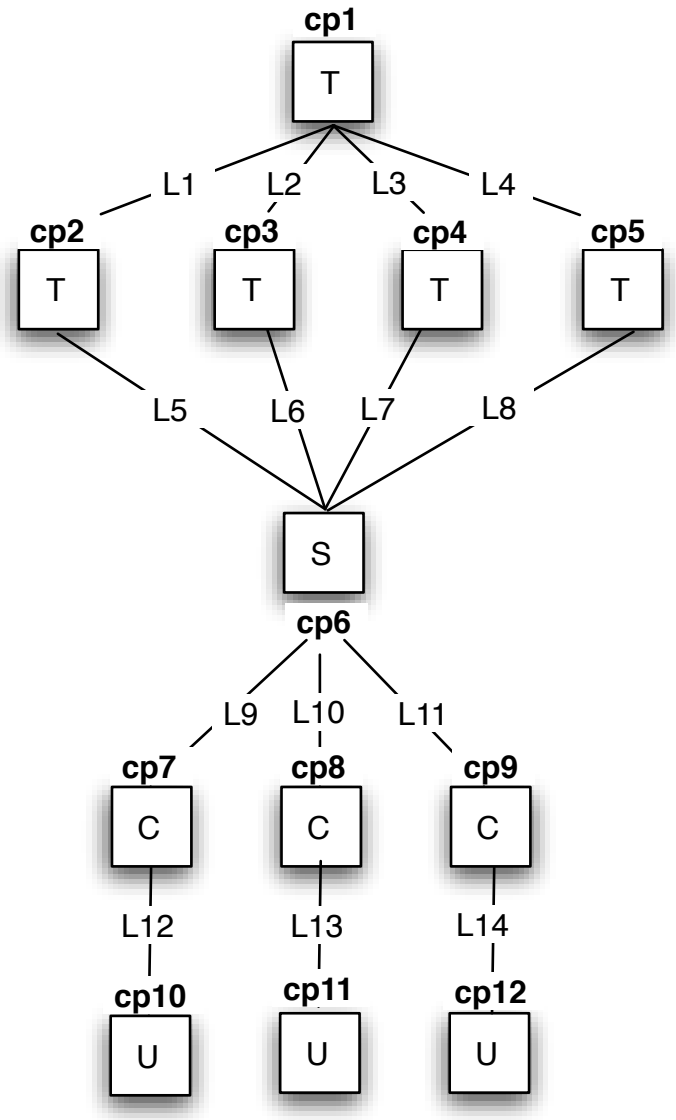


Figure 5.2: Network configuration # 2

Path#	Links	CVP Paths Generatos
P_1	$\{[L_1, 0], [L_5, 10], [L_9, 15], [L_{12}, 25]\}$	$P_{1.a} = [L_5, L_9], P_{1.b} = [L_9, L_{12}]$
P_2	$\{[L_1, 0], [L_5, 10], [L_{10}, 35], [L_{13}, 5]\}$	$P_{2.a} = [L_5, L_{10}], P_{2.b} = [L_{10}, L_{13}]$
P_3	$\{[L_1, 0], [L_5, 10], [L_{11}, 45], [L_{14}, 5]\}$	$P_{3.a} = [L_5, L_{11}], P_{3.b} = [L_{11}, L_{14}]$
P_4	$\{[L_2, 0], [L_6, 20], [L_9, 15], [L_{12}, 25]\}$	$P_{4.a} = [L_6, L_9], P_{4.b} = [L_9, L_{12}]$
P_5	$\{[L_2, 0], [L_6, 20], [L_{10}, 35], [L_{13}, 5]\}$	$P_{5.a} = [L_6, L_{10}], P_{5.b} = [L_{10}, L_{13}]$
P_6	$\{[L_2, 0], [L_6, 20], [L_{11}, 45], [L_{14}, 5]\}$	$P_{6.a} = [L_6, L_{11}], P_{6.b} = [L_{11}, L_{14}]$
P_7	$\{[L_3, 0], [L_7, 20], [L_9, 15], [L_{12}, 25]\}$	$P_{7.a} = [L_7, L_9], P_{4.b} = [L_9, L_{12}]$
P_8	$\{[L_3, 0], [L_7, 20], [L_{10}, 35], [L_{13}, 5]\}$	$P_{8.a} = [L_7, L_{10}], P_{5.b} = [L_{10}, L_{13}]$
P_9	$\{[L_3, 0], [L_7, 20], [L_{11}, 45], [L_{14}, 5]\}$	$P_{9.a} = [L_7, L_{11}], P_{6.b} = [L_{11}, L_{14}]$
P_{10}	$\{[L_4, 0], [L_8, 30], [L_9, 15], [L_{12}, 25]\}$	$P_{10.a} = [L_8, L_9], P_{4.b} = [L_9, L_{12}]$
P_{11}	$\{[L_4, 0], [L_8, 30], [L_{10}, 35],[L_{13}, 5]\}$	$P_{11.a} = [L_8, L_{10}], P_{5.b} = [L_{10}, L_{13}]$
P_{12}	$\{[L_4, 0], [L_8, 30], [L_{11}, 45],[L_{14}, 5]\}$	$P_{12.a} = [L_8, L_{11}], P_{6.b} = [L_{11}, L_{14}]$

Eliminating the least expensive links

If we extract the cascade vulnerability paths generators, we eliminate the minimal number of links that generate the CVP. Yet, before eliminate the links, we consider their cost to find the least expensive links in the network. This analysis is shown in Table 5.2.

From the analysis showed on Table 5.2, in order to obtain a CVP-free network, we have two options. The first one is to cut the links `[`L_9`, 15]`, `[`L_{10}`, 35]`, and `[`L_{11}`, 45]`, which represents a total risk of **3** according to the sensitivity matrix. And the second option is to cut the links `[`L_{12}`, 25]`, `[`L_{13}`, 5]`, and `[`L_{14}`, 5]`, which represents an accumulative risk of **6**.

From this configuration we can say that if we remove the first option of links, we will remove the links that generate the minimum risk on the network and whose operational cost is 115 units.

If we choose to remove the second option of links, we will be removing the ones that

Path#	Risk	Min. link to eliminate	CVP generators	Cost	Final Risk
P_1	5	L_9			
P_2	5	L_{10}	L_9	15	4
P_3	5	L_{11}	L_{10}	35	4
P_4	5	L_9	L_{11}	45	4
P_5	5	L_{10}			
P_6	5	L_{11}	or		
P_7	5	L_9			
P_8	5	L_{10}			
P_9	5	L_{11}	L_{12}	25	3
P_{10}	5	L_9	L_{13}	5	3
P_{11}	5	L_{10}	L_{14}	5	3
P_{12}	5	L_{11}			

Table 5.2: Network configuration # 2 analysis

generate more risk and whose total operational value of 35 units.

Given this information, we can choose the best option according to our preferences. For instance, among the links that generate more risk, we eliminate the ones that cost the less, i.e., L_{13} and L_{14}. An alternative is to choose eliminate L_9 and L_{13}, which is another way to reduce the risk and the operational cost.

Links	cost	final network value
L_{13}, L_{14}	10 units	230 units
L_9, L_{13}	20 units	220 units

Example 3

The last network configuration depicted in Figure 5.3, contains 10 computers, and 12 links, where *cp*1 owns the highest sensitivity level and the lower sensitivity levels are stored in computers *cp*9 and *cp*10. For this example, the corresponding weights for the links and the number of occurrences are:

link	cost (units)	frequency
L_1	10	1
L_2	10	1
L_3	10	1
L_4	10	1
L_5	30	3
L_6	50	3
L_7	15	3
L_8	10	3
L_9	20	4
L_{10}	40	4
L_{11}	45	4
L_{12}	55	4
Total	260 units	

The total operational value of this network is 260 units, and has a total of 8 cascading paths.

Path#	Links	CVP Paths Generatos
P_1	$\{[L_1, 10], [L_5, 30], [L_9, 20], [L_{11}, 45]\}$	$P_{1.a} = [L_1, L_5], P_{1.b} = [L_5, L_9, L_{11}]$
P_2	$\{[L_1, 10], [L_5, 30], [L_9, 20], [L_{12}, 55]\}$	$P_{2.a} = [L_1, L_5], P_{2.b} = [L_5, L_9, L_{12}]$
P_3	$\{[L_2, 10], [L_6, 50], [L_9, 20], [L_{11}, 45]\}$	$P_{3.a} = [L_2, L_6], P_{3.b} = [L_6.L_9, L_{11}]$
P_4	$\{[L_2, 10], [L_6, 50], [L_9, 20], [L_{12}, 55]\}$	$P_{4.a} = [L_2, L_6], P_{4.b} = [L_6.L_9, L_{12}]$
P_5	$\{[L_3, 10], [L_7, 15], [L_{10}, 40], [L_{11}, 45]\}$	$P_{5.a} = [L_3, L_7], P_{5.b} = [L_7, L_{10}, L_{11}]$
P_6	$\{[L_3, 10], [L_7, 15], [L_{10}, 40], [L_{12}, 55]\}$	$P_{6.a} = [L_3, L_7], P_{6.b} = [L_7, L_{10}, L_{12}]$
P_7	$\{[L_4, 10], [L_8, 10], [L_{10}, 40], [L_{11}, 45]\}$	$P_{7.a} = [L_4, L_8], P_{8.b} = [L_8, L_{10}, L_{11}]$
P_8	$\{[L_4, 10], [L_8.10], [L_{10}, 40], [L_{12}, 55]\}$	$P_{7.a} = [L_4, L_8], P_{8.b} = [L_8, L_{10}, L_{12}]$

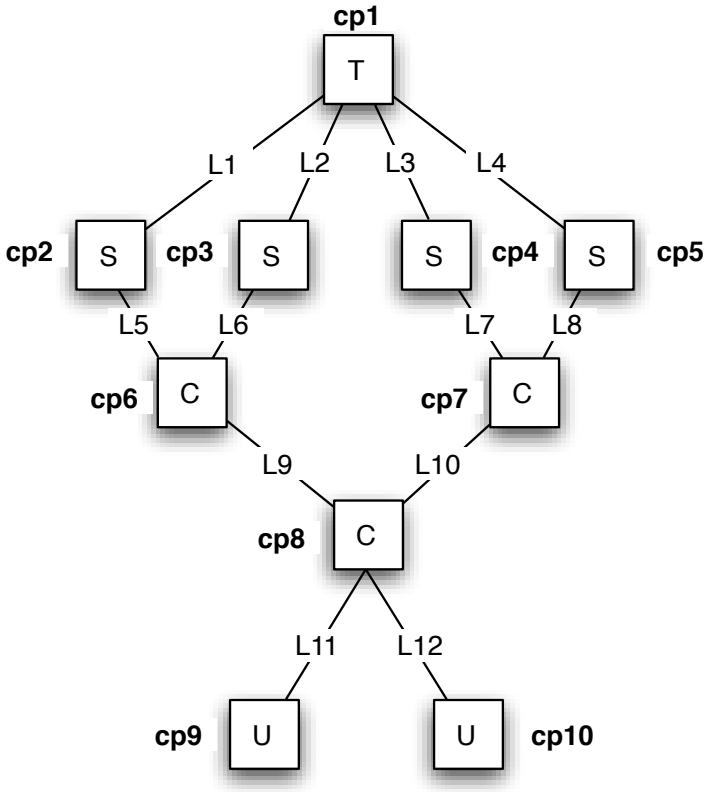


Figure 5.3: Network configuration # 3

Eliminating the Least expensive links

In Table 5.3 we can see that the 8 paths generate a risk of **5** according to the sensitivity matrix. In order to eliminate the CVP, we found two possible choices. The first one is to remove links L_5, L_6, and L_7; the link's removal will reduce the risk to **2**. The second choice is to remove links L_{11}, and L_{12}; the link's removal will reduce the risk to **2**. Notice that either option will remove the CVP. The intuitive decision is to select the second choice, since removes the minimal number of links in the network. However, notice that links L_{11}, and L_{12} are the most expensive links in the network, with 45 and 55 units respectively.

If we decide to eliminate the links suggested in the first choice, we will eliminate the operational value from the network that costs 65 units. If we decide to eliminate the second choice, the operational value removed will be of 100 units, i.e., the more expensive.

Path#	Risk	Min. link to eliminate	CVP generators	Cost	Final Risk
P_1	5	L_5	L_5	15	2
P_2	5	L_{11}	L_6	35	2
P_3	5	L_6	L_7	15	2
P_4	5	L_{12}			
P_5	5	L_7	or		
P_6	5	L_{11}			
P_7	5	L_8	L_{11}	45	2
P_8	5	L_{12}	L_{12}	55	2

Table 5.3: Network configuration # 3 analysis

Observations

We have shown three different network configurations. In Table shows the results of comparing the MHS algorithms (without considering weights in the links), and the MWHS (considering weights).

Notice that these algorithms that find the minimal links and minimal weighted links are greedy-based algorithms, which means that more than one answer can be obtained (i.e., depends of the best option on a specific configuration).
Sometimes when it is obtained more than one results, the MWHS algorithm picks as a solution the option that has the smallest impact in the network, i.e., the least expensive choice.

In the last Table from this chapter we can analyze the results from different configurations. First we mention the *Total Links*, which are the total number of links involved in the network. The *CVP links generators* are the links responsible of generating the CVP. The *MHS* are the results obtained by applying the Minimal Hitting Set algorithm (without considering weights), similarly, the *MWHS* are the results obtained by applying the Minimal Weighted Hitting Set algorithm (considering weights). The *Cost* column shows the smallest and the largest operational value from the link removal candidates. And finally, the *Total cost* column shows the final operational value removed from the original network.

Network Configurations Analysis						
Network Configs	Total Links	CVP links generators	MHS	Cost	MWHS	Total Cost
Config.#1	5	$\{L_1, L_2, L_3, L_4\}$	$\{L_1, L_3, L_4\}$ $\{L_2, L_4\}$	[35,95]	$\{L_2, L_4\}$	35 units
Config.#2	14	$\{L_5, L_6, L_7, L_8\}$ $\{L_9, L_{10}, L_{11}, L_{12}\}$ $\{L_{13}, L_{14}, L_{15}, L_{16}\}$	$\{L_9, L_{10}, L_{11}\}$ $\{L_{12}, L_{13}, L_{14}\}$	[220,230]	$\{L_9, L_{13}\}$	220 units
Config.#3	12	$\{L_1, L_2, L_3, L_4\}$ $\{L_5, L_6, L_7, L_8\}$ $\{L_{11}, L_{12}\}$	$\{L_{11}, L_{12}\}$ $\{L_5, L_6, L_7\}$	[65,100]	$\{L_5, L_6, L_7\}$	65 units

Chapter 6

Conclusions

6.1 Our Contribution

6.1.1 Significance of our contribution

In this thesis, we considered a problem of security leakage in a network of computer systems. The particular problem we addressed is well-known as the Cascade Vulnerability problem, in the *Multi-Level Security* (MLS) model.

Our approach consisted in extending the previous work of Bistarelli et al. , by taking into account values on the connections of the network: some connections are indeed more valuable than others, and we needed to represent such information.

Our contribution was the following: We kept the approach of Bistarelli et al. , using soft constraints to model, detect and solve the CVP, but we extended it by including "weights" in the links. We use the minimum weighted hitting set theory to eliminate the minimal and least expensive links in a network affected by the CVP.

The network, after the link's elimination provided by our approach, was leakage free. In addition, we implemented the approaches from [9], [13], and [37].

We implemented the constraints mentioned in Bistarelli et al. approaches. We design and implement the CVP Tool Simulator, which is an implementation of the algorithms mentioned above. A graphical user interface was implemented as well to interact with the CVP Tool.

Although, the work in [13, 9] provide a transparent explanation how to soft a CSP for the CVP, it was necessary to form a better explanation for implementation purposes, since

no implementation was provided before. Also, our contribution consists in extending the detection and elimination of the CVP in more complex networks that are not limited to "linear shapes" (e.g., see Figurefig:acvp2).

6.2 Future Work

As future work, we would like to see the impact that instead of transforming constraint $c2$ we will see if we can modify $c2$ in such a way that we can keep it 'soft' and change the optimal problem to prevent the function to reach $\mathbf{0}$.

Also, we are considering different cost aggregation, instead of considering the minimum weighted hitting set; for example, if the weights are not costs but probabilities of failure in the connections. This lead us to maximize the probability to maintain reliable paths in the network.

Finally, we realize that after eliminating the least expensive links that generate the CVP, a cluster of computers (with potentially the same security level) will remain disconnected. We have begin to develop algorithms that can connect these computers in the cluster in order to keep a small network (with a small degree of risk) functioning.

References

[1] Russ B. Altman, Bryn Weiser, and Harry Noller. Constraint satisfaction techniques for modeling large complexes: Application to the central domain of 16s ribosomal rna, 1994.

[2] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag, Berlin-Heidelberg, 1999.

[3] Bruno Backer, V. Furnon, P. Kilby, P. Prosser, and P. Shaw. Solving vehicle routing problems using constraint programming and metaheuristics. *Journal of Heuristics*, 6(4):501–523, September 2000.

[4] Rolf Backofen. Constraint techniques for solving the protein structure prediction problem. In Michael Maher and Jean-Francois Puget, editors, *Proceedings of* 4^{th} *International Conference on Principle and Practice of Constraint Programming (CP'98)*, volume 1520 of *Lecture Notes in Computer Science*, pages 72–86. Springer Verlag, 1998.

[5] Rolf Backofen and Sebastian Will. A constraint-based approach to fast and exact structure prediction in three-dimensional protein models. *Journal of Constraints*, 11(1):5–30, January 2006.

[6] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint P-based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer International Series, 2001.

[7] D. Bell. and L. La Padula. Secure computer systemsl mathematical foundations and model, November 1973.

[8] Kenneth J. Biba. Integrity considerations for secure computer systems. In *ESD-TR-76-372, ESD/AFSC*, Hanscom AFB, Bedford, Mass., April 1977.

[9] Stefano Bistarelli and Simon N. Foley. Detecting and eliminating the cascade vulnerability problem from multi-level security networks using soft constraints. In *Innovative Applications of Artificial Intelligence Conference*, pages 25–29, July 2004.

[10] Stefano Bistarelli, Simon N. Foley, and Barry O'Sullivan. A soft constraint-based approach to the cascade vulnerability problem. *J. Computer Security.*, 13(5):699–720, 2005.

[11] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Constraint solving over semirings. In Chris Mellish, editor, *Proceedings International Joint Conference on Artificial Intelligence (IJCAI'95)*, Montreal, 1995.

[12] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, March 1997.

[13] Stefano Bistarelli, Barry O'Sullivan, and Simon N. Foley. Modeling and detecting the cascade vulnerability problem using soft constraints. In *ACM Symposium on Applied Computing (SAC 2004)*, pages 14–17, March 2004.

[14] Philippe Blache. Using active constraints to parse gpsgs. In *Proceedings of the 14th conference on Computational linguistics*, pages 81–86, Morristown, NJ, USA, 1992. Association for Computational Linguistics.

[15] Alan Borning, R Duisberg, Bjorn Freeman-Benson, Gleen A. Kramer, and M. Woolf. Constraint hierarchies. In *In Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications*, pages 48–60. ACM, 1987.

[16] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. In *Lisp and Symbolic Computation*, pages 48–60. ACM, 1992.

[17] Alan Borning, Michael Maher Amy Martindale, and Molly Wilson. Constraint hierarchies and logic programming. In *In Proceedings of the Sixth International Conference on Logic Programming*, pages 149–164, Lisbon, Portugal, 1989.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, Inc., New York, NY, USA, 1990.

[19] D. Dubois, H. Fargier, and H. Prade. Fuzzy constraints in job-shop scheduling, 1995.

[20] Hélène Fargier and Jérôme Lang. Uncertainty in constraint satisfaction problems: a probabilistic approach. In *ECSQARU '93: Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 97–104, London, UK, 1993. Springer-Verlag.

[21] Hélène Fargier, Jérôme Lang, Roger Martin-Clouaire, and Thomas Schiex. A constraint satisfaction framework for decision under uncertainty. In *11th International Conference on Uncertainty in Artificial Intelligence*, Montreal, Canada, 1995.

[22] John A. Fitch and Lance J. Hoffman. On the shortest path to network security. In *Computer Security Applications Conference*, pages 149–158, December 1993.

[23] Eugene C. Freuder. Partial constraint satisfaction. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 278–283, Detroit, Michigan, USA, 1989.

[24] Eugene C. Freuder. *Constraint Programming*, volume 2. Springer Netherlands, April 1997.

[25] Stefanos Gritzalis and Diomidis Spinellis. The cascade vulnerability problem: The detection problem and a simulated annealing approach to its correction. In *Microprocessors and Microsystems*, volume 21, pages 621–628, 1998.

[26] Pascal Van Hetenryck. *Constraint Satisfaction in Logic Programming.* MIT Press series in logic programming., 1989.

[27] Joseph D. Horton. Detecting cascade vulnerability in linear time. In *IEEE Symposium on Security and Privacy*, pages 110–116, 2000.

[28] Joseph D. Horton, Robert Harland, Elton Ashby, R.H. Cooper, and W.F. Hyslop. The cascade vulnerability problem. In *IEEE Symposium on Security and Privacy*, pages 110–116, 1993.

[29] Bernhard Korte and Jens Vygen. *Combinatorial Optimization.* Springer, third edition, 2006.

[30] Tanja Magoc. *New algorithms for portfolio selection.* PhD thesis, Computer Science Department, The University of Texas at El Paso, April 2009.

[31] J.K. Millen. Algorithm for the cascading problem. In *IEEE Cipher Forum on DOCK-MASTER.NCSC.MIL*, June 1990.

[32] C. H. Papadimitriou. *Computational Complexity.* Addison Wesley, San Diego, CA, USA, 1994.

[33] Charles P. Pfleeger and Shari Lawrence Pfleeger. *Security in Computing.* Prentice Hall Professional Technical Reference, 2002.

[34] Zsofi Ruttkay. Fuzzy constraint satisfaction. In *Proceedings 1st IEEE Conference on Evolutionary Computing*, pages 542–547, Orlando, 1994.

[35] Thomas Schiex. Possibilistic constraint satisfaction problems or "how to handle soft constraints?". In *Proceedings of the Eight International Conference on Uncertainty in Artificial Intelligence*, pages 268–275, Stanford, CA, 1992.

[36] Thomas Schiex, Hélène Fargier, and Gerard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In Chris Mellish, editor, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, Montreal, 1995.

[37] Christian Servin, Martine Ceberio, Eric Freudenthal, and Stefano Bistarelli. An optimization approach for the cascade vulnerability problem using soft constraints. In Marek Reformat and Michael R. Berthold, editors, *Proceedings of the 26th International Conference of the North American Fuzzy Information Processing Society NAFIPS'2007*, pages 372–377, San Diego, California, June 2007. IEEE, Press.

[38] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 2 edition, February 2005.

[39] CSC-STD-003-85 TCSEC, Computer security requirements guidance for applying the department of defense trusted computer system evaluation criteria in specific environments. Orange book. Technical report, National Computer Security Center, 1985.

[40] TNI. Trusted computer system evaluation criteria: Trusted network interpretation (tni). Technical report, National Computer Security Center. Red Book, 1987.

[41] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2003.

Appendix A

Sensitivity Matrices

MINIMUM USER CLEARANCE	RATING
Uncleared (U)	0
Not Cleared but Authorized Access to Sensitive Unclassified Information (N)	1
Confidential (C)	2
Secret(S)	3
Top Secret (TS)/Current Background Investigation (BI)	4
Top Secret (TS)/Current Special Background Investigation (SBI)	5
One Category (1C)	6
Multiple Categories (MC)	7

Table A.1: Minimum User Clearance and the corresponding rating for each security level

src\dest	U	N	C	S	TS
U	0	0	0	0	0
N	1	0	0	0	0
C	2	1	0	0	0
S	3	2	1	0	0
TS	5	4	3	2	0

Table A.2: Assurance matrix

U = Uncleared or Unclassified
N = Not Cleared but Authorized Access to Substantive Unclassified information
C = Confidential
S = Secret
TS = Top Secret
TS (BI) = Top Secret (Background Investigation)
TS (SBI) = Top Secret (Special Background Investigation)
1C = One Category
MC = Multiple Categories

Appendix B

Prolog Source Code

Link Checker

```
:− dynamic link.
link(a1, a3, t, c, 32).
link(a2, a4, c, t, 43).
link(a3, a2, t, s, 32).
link(a1, a4, s, c, 13).
link(a1, a6, s, s, 13).
link(a5, a6, c, c, 13).
link(a1,a3,t,s,69).


% Each security level has been assigned a numeric value
% for categorizing computers according to the security protection
% they provide. These values are obtained from the
% Trusted Computer System Evaluation Criteria (TCSEC)
%
% Since a clearance implicitly encompasses lower clearance levels
% (e.g., a Secret cleared user has an implicit Confidential clearance),
% the phrase ‘‘minimum clearance of system users’’ is more accurately
% stated as ‘‘maximum clearance of the least cleared system user.’’
% The levels of sensitiviness are:
% u − unclassified
```

```
% n − non classified
% c − classfied
% s − secret
% t − top secret
%
rating(u,0).
rating(n,1).
rating(c,2).
rating(s,3).
rating(m,4).
rating(t,5).

% There are different flows that are possible within a system and/or links.

% permitted_flow :− Represents the information flows that are permitted
% by the policy in each node, e.g., may flow from level
% T in sys.A to level T in sys.B.
is_permitted(Src, Dest):−
        Dest = Src.

is_permitted(Src, Dest):−
        Dest > Src.

% risk_flow :− Represents the information flows that are not
% permitted by the policy, but for which there
% is a risk of flow if the system became
% compromised. e.g., the risk from the level T
% to level S in a system S is B2 (assurance level 2),
```

```
% corresponding to the level of assurance at which
% Sys.E has been evaluated.
is_risky(Src, Dest):−
        Src − Dest =< 2.

% The rest of the flows are considered to be invalid flows.
% invalid_flow :− Represents all the remaining flows that are not
% valid (that is, are impossible for the given system).
% For example, a flow from T to C is not possible on
% a system that has not been configured to store
% information labelled as classified.
is_invalid(Src,Dest):−
not( is_permitted(Src,Dest)),
not( is_risky(Src,Dest)).


% flow checker 'checks' if the links are permitted, risky, or invalid.
% This procedure invokes the corresponding predicates to determine the type of
% flow the link has.
flow_checker :−
                link(Src, Dest, L1, L2, W),
                rating(L1, V1),
                rating(L2, V2),
                is_permitted(V1, V2),
                write('link('),write(Src), write(', '), write(Dest),
                write(', ['), write(L1), write(','), write(L2),
                write(']). is_permitted'),nl,
                retract(link(Src, Dest, L1, L2, W)),
```

```
                flow_checker.

flow_checker :−
                link(Src, Dest, L1, L2, W),
                rating(L1, V1),
                rating(L2, V2),
                is_risky(V1, V2),
                write('link('),write(Src), write(', '), write(Dest),
                write(', ['), write(L1), write(','), write(L2),
                write(']). is risky'),nl,
                retract(link(Src, Dest, L1, L2, W)),
                flow_checker.

flow_checker :−
                link(Src, Dest, L1, L2, W),
                rating(L1, V1),
                rating(L2, V2),
                is_invalid(V1, V2),
                write('link('),write(Src), write(', '), write(Dest),
                write(', ['), write(L1), write(','), write(L2),
                write(']). is invalid'),nl,
                retract(link(Src, Dest, L1, L2, W)),
                flow_checker.

flow_checker.
```

```
write_answer(File, P) :−
        open(File, append, Stream),
        write(Stream, P),
        nl(Stream),
        close(Stream).
```

CVP Path Finder

```
% Description of the Program:
% INPUT: a network composed of computers and links.
% OUTPUT: a CVP−free network
% The network (set of links between computers, L),
% are given to is_cvp_free(L).
%


% ::::: Description of the Network :::::
% A network is composed of computers and the connections
% (links) between computers.
% Computer
% A computer is composed of a name and a list containting
% the security levels in this computer.
% computer( <name> , [list of security levels])
% consult('computers').
computer(a,[t,s]).
computer(b,[s,c]).
computer(c,[c]).
computer(cp1,[t]).
computer(cp2,[t]).
computer(cp3,[t]).
computer(cp4,[t]).
computer(cp5,[t]).
computer(cp6,[t]).
computer(cp7,[s]).
computer(cp8,[c]).
```

```
computer(cp9,[c]).
computer(cp10,[c]).
computer(cp11,[u]).
computer(cp12,[u]).
computer(cp13,[u]).
% Link
% A computer connections (link), between two computers, is
% composed of the following elements
% link(<src> , <dest> , [src_sec_level, dest_sec_level], weight)
% <src> :- the computer source where the connection
% was established
% <dest> :- the computer destination, where the
% connection was targeted
% <src_sec_level> :- the security level from the source computer
% that this connection was established.
% <dest_sec_level>:- the security level from the destination
% computer that this connection was targeted
% to be connected.
% <weight> :- a numeric value representing the weight of
% this link between computer <src> to
% computer <dest>.
% e.g.,
% A link from a top secret level from Computer A to the secret
% level in computer B has a weight 24.
%
% link(A, B, [t,s], 24 ).
%
% consult('links').
```

```
link(a,b,[t,s],24).
link(b,c,[s,c],11).
link(cp1,cp2, [t,s], 10).
link(cp1,cp3, [t,s], 15).
link(cp1,cp4, [t,s], 11).
link(cp1,cp5, [t,s], 15).
link(cp1,cp6, [t,s], 10).
link(cp2,cp7, [s,s], 10).
link(cp3,cp7, [s,s], 10).
link(cp4,cp7, [s,s], 10).
link(cp5,cp7, [s,s], 10).
link(cp6,cp7, [s,s], 10).
link(cp7,cp8, [s,c], 20).
link(cp7,cp9, [s,c], 20).
link(cp7,cp10, [s,c], 20).
link(cp8,cp11, [c,u], 13).
link(cp9,cp12, [c,u], 13).
link(cp10,cp13,[c,u], 13).
% Maximum data sensitivity and minimum data sensitivity
%
% Each security level has been assigned a numeric value
% for categorizing computers according to the security protection
% they provide. These values are obtained from the
% Trusted Computer System Evaluation Criteria (TCSEC)
%
% Since a clearance implicitly encompasses lower clearance levels
% (e.g., a Secret cleared user has an implicit Confidential clearance),
% the phrase ''minimum clearance of system users'' is more accurately
```

```
% stated as ''maximum clearance of the least cleared system user.''
% The levels of sensitiviness are:
% u - unclassified
% n - non classified
% c - classfied
% s - secret
% t - top secret
%
max_data_sensitivity(u,0).
max_data_sensitivity(n,1).
max_data_sensitivity(c,2).
max_data_sensitivity(s,3).
max_data_sensitivity(t,5).
min_data_sensitivity(u,0).
min_data_sensitivity(n,1).
min_data_sensitivity(c,2).
min_data_sensitivity(s,3).
min_data_sensitivity(t,5).



% assurance function.
% calculates the assurance value N from a given computer X
% this method invokes the information from matrix.
%
% X = a computer which assurance value is to be calculated
% N = the assurance value for computer X
%
```

```
assurance(X,N):−
        computer(X,L),
        matrix(L,N).

% matrix function
% Given a list of security levels, calculate the difference
% of sensitivity levels.
% [H|T] = List of sensitivity levels
% N = the total assurance value for the composition of the
%   sensitivity levels from the list.
matrix([_], N) :− N is 0.
matrix([H|T], N) :−
        last(Tail,T),
        max_data_sensitivity(H,Z),
        min_data_sensitivity(Tail,Y),
        N is Z − Y.


% find_path
% Find the path between a source ‘‘X’’ and a destination ”Y”
% In case there exista link between computer ”X” and
% computer ‘‘Y’’ return this link. If not, find a computer
% ‘Z’ that is connected to ‘X’ that might be connected to
% computer ”Y”. Return a path ‘P’ that contains all the links
% from computer ‘‘X’’ to computer ‘‘Y’’.
%
% E.g., Find the path from computer X to computer Y,
```

```
% return the path containing the links that connect X to Y
% also return the total cost of the path V, which is the
% sum of all the links' weight W,
% X = source computer
% Y = destination computer
% Z = a computer that is in the path
% W = the corresponding weight from link X Y
% V = total cost of the path (addition of all weights W)



find_path(X,Y,[[X,Y,W]], V) :-
                    link(X,Y,_,W),
                    V is W.

find_path(X,Y,[[X,Z,W]|P], V) :-
                    link(X,Z,_,W),
                    find_path(Z,Y,P, V2),
                    V is V2 + W.

% get paths
% This method invokes 'find_path', and writes the output
% into a file called 'plg_out.txt'. This method also invokes
% write_path, that is in charge of the writing file routine.
% X = source computer
% Y = destination computer
% P = set of paths that connects computer X with computer Y
% V = the corresponding value for path P
%
```

```
get_paths(X,Y,P,V):−
                find_path(X,Y,P,V),
                write_path('plg_out.txt', P, V).


% extract_subpaths
% This method identifies the sub paths that are generating
% the CVP. Given a path 'P' from a source computer to a destination
% computer, will return a set of subpaths that violate the
% constraint i.e., Risk > Effort.
% The Effort is calculated as fallowing:
% For each link (composed of two computers), we calculate
% the Effort from this connection, e.g., see 'calculate_effort'.
% and it is compared with the Risk obtained from this link, e.g.,
% see 'matrix'.
% Example:
% Given a path P = [[cp1, cp2, 10], [cp2, cp7, 10], [cp7, cp8, 20]]
% extract Effort from cp1 and cp2 (using assurance)
% check Risk from [cp1, cp2, 10] (using matrix)
% compare
% IF Risk < Effort THEN keep the Effort and Risk values and compare
% the next link, i.e., [cp2, cp7, 10]
% ELSE
% extract the path so far that violates the constraint, i.e.,
% from the last source computer until this point.
% call again this method but now the source computer will be the
% last link processed.
% [ [X, Y, W1] | P] : list of links in the form [X1, Y1, W1], ...,
```

```
% [Xi, Yj, Wj], where X is src computer, Y is dest
% computer, and W is the weight
% Efforts : the maximum accumulative effort from link to link
% Risk : the accumulative risk from link to link
%, Partial :-the partial set of links, which holds the links
% that violated the constraint risk > effort.
%

extract_subpaths( [[X, Y, W1],[Y,Z,W2],|P], Partial ):-
                    calculate_effort(X,Y,Effort),
                    write('MAX: '),write(Effort),nl,
                    link(X,Y,Levels,_),
                    matrix(Levels, Risk),
                    Risk > Effort,
                    Tmp = [Partial|[X, Y, W1]],
                    extract_subpaths([Y,P], ,Tmp).

% calculate_effort
% This method calculates the effort of two given computers
% (computer X and computer Y). This method maps the corresponding
% effort of X and Y and finds the maximum value of these two and
% returns 'Effort' as the maximum value of these two computers.
% This method uses the helper method 'max'
% X = a computer which effort is desired to be calculated
% Y = a computer which effort is desired to be calculated
% Effort = the maximum effort between X's effort and Y's effort
%
```

```
calculate_effort(X,Y,Effort):-
                    assurance(X,Effort1),
                    assurance(Y,Effort2),
                    Effort is max(Effort1,Effort2).




get :-
                    find_path('cp1','cp8',P,_),
                    write('PATH:␣'),write(P),nl,
                    extract_subpaths(P,0,0,[]).




% Helpers
% The following methods are helpers for other methods:
% - max : returns the maximum from two numbers X and Y
% - maxlist : returns the maximum value from a list [X]
% - write_path: performs the writing routine. Writes the paths 'P'
% into a file 'File'.
% - last  : returns the last element X from a list [X].
%
%


max(X,Y,X) :- X>=Y.
max(X,Y,Y) :- X< Y.
maxlist([X],X).
maxlist([X,Y|Rest],Max) :-
```

```
maxlist([Y|Rest],MaxRest),
max(X,MaxRest,Max).



write_path(File, P,V) :−
                open(File, append, Stream),
                write(Stream, P),
                write(Stream, V),
                nl(Stream),
                close(Stream).
                % Helper


last(X,[X]).
last(X,[_|Xs]):−last(X,Xs).
```

Appendix C

Java Source Code

Algorithms

```
import java.util.*;
import javax.swing.SwingUtilities;
import java.lang.Exception;
import java.lang.reflect.InvocationTargetException;


/**
 * <center>
 *  The University of Texas at El Paso<br>
 *    UTEP<br>
 * Computer Science Department<br>
 * </center>
 *
 * This class contains different algorithms used in the
 * CVPTool class.
 *


 * @author Christian Servin
 *
 * <b>Thesis Name</b>: An Optimization Approach to Solve the
 * Cascade Vulnerability Problem using Soft Constraints.
```

```
 *
 * Advisor:
 * Dr. Martine Ceberio, Dr. Eric Freudenthal
 *
 * Date:
 * Wed Jul 11 17:16:02 MDT 2007
 * Note:
 * @modified today
 */



public class Test{
static int CONTADOR = 0;
/**
 * <b>findHighest</b><br>
 * Method that extract the position from an array of integers,
 * which position has the highest value.
 * @param values array of integers containing the respective counters
 * for the elements in the correponding position, e.g.,<br>
 * L<sub>1</sub> = values[ 0 ]<br>
 * L<sub>2</sub> = values[ 1 ]<br>
 * ...<br>
 * L<sub>n</sub> = values[ n-1 ]<br>
 * @return the position of the most common link (the link with the
 * highest counter), in case of a tie a random one is selected.
 **/
protected static int findHighest(int[] values){
        int tmp = values[0];
```

```
        int pos = 0;
        LinkedList<Integer> duplicates = new LinkedList<Integer>();
        duplicates.add( new Integer(pos) );

        for (int i = 1; i < values.length; i++) {
                if(values[i]== tmp){
                        duplicates.add( new Integer(i));
                }
                if(values[i]>tmp){
                        duplicates.clear();
                        tmp = values[i];
                        pos = i;
                        duplicates.add( new Integer(pos) );
                }
        }
        return pos = duplicates.size() == 1 ? pos : duplicates.get( ((int)
        (Math.random()*duplicates.size())) ).intValue();
}

/**
 * <b>findChepestLinks</b><br>
 * Method that extract the position from an array of integers, which
 * position has the least expensive element.
 * @param values array of integers containing the respective counters
 * for the elements in the correponding position, e.g.,<br>
 * L<sub>1</sub> = values[ 0 ]<br>
 * L<sub>2</sub> = values[ 1 ]<br>
 * ...<br>
```

```
* L<sub>n</sub> = values[ n-1 ]<br>
* @param weights array of doubles containing the weights from the
* respective elements in the corresponding position, e.g., <br>
* L<sub>1</sub> = weignts[ 0 ]<br>
* L<sub>2</sub> = weignts[ 1 ]<br>
* ...<br>
* L<sub>n</sub> = weignts[ n-1 ]<br>
* @return the position of the least expensive link in case of a
* tie a random one is selected.
**/

protected static int findChepestLinks(int[] values, double[] weights){
        int tmp = values[0];
        int pos = 0;
        LinkedList<Integer> duplicates = new LinkedList<Integer>();
        duplicates.add( new Integer(pos) );
        for (int i = 1; i < values.length; i++) {
                if(values[i]== tmp)
                        duplicates.add( new Integer(i));
                if(values[i]>tmp){
                        duplicates.clear();
                        tmp = values[i];
                        pos = i;
                        duplicates.add( new Integer(pos) );
                }
        }
        // find the least expensive among the duplicates available
        double minimum = weights[ duplicates.get(0) ];
```

```
        LinkedList<Integer> cheapest_links = new LinkedList<Integer>();
        cheapest_links.add( duplicates.get(0) );

        for (int i = 1; i < duplicates.size(); i++) {
                if(weights[duplicates.get(i)] == minimum)
                        cheapest_links.add( duplicates.get(i));
                if(weights[duplicates.get(i)] < minimum){
                        cheapest_links.clear();
                        minimum = weights[ duplicates.get(i) ];
                        cheapest_links.add( duplicates.get(i) );
                }
        }
        return pos = cheapest_links.size() == 1 ? pos :
        cheapest_links.get( ((int)(Math.random()*cheapest_links.size())) ).intValue();
}
/**
 * <b>updatecounter</b><br>
 * Method that decrements the contents in <i>counters</i> at position
 * <i>pos</i> by <b>1</b>.
 * @param counters array of integers containing the respective counters
 * @param pos position in <i>counters</i> that will be used to update
 * the contents in <i>counters</i>
 * @return the array <i>counters</i> with the updated position stated
 * by <i>pos</i>
 **/
protected static int[] updatecounters(int[] counters, int pos){
        counters[pos] = (counters[pos])-1;
        return counters;
```

```
}

protected static void updateTheCounter(LinkedList<Set> S){
        CONTADOR = 0;
        for(Set s : S){
                CONTADOR+= s.size();
        }
}

/**
* <b>apply_MWHS</b><br>
* apply_MWHS (Minimal Weighted Hitting Set) algorithm.
* solves the MWHS problem<br>
* <b>INSTANCE:</b><br>
* Finite set U, with |U| = m;
* C = {S<sub>1</sub>, &hellip; ,S<sub>n</sub>} s.t. S<sub>i</sub>
* &sube; U &forall; i = {1, &hellip; ,<i>n</i>}
* A weight function <i>w</i>: U &rarr; &real;+.<p>
* <b>SOLUTION:</b><br>
* A hitting set for <tt>C</tt>, thatis to say H &sube; U s.t.
* H<sup>S</sup><sub>i</sub> &ne;,
* &forall; <i>i</i> = 1, &hellip; ,n.<p>
* @param  links: containing all the links in the network
* @param  S a set of subsets {s<sub>1</sub>, ..., s<sub>k</sub>}
* such that Union of s<sub>i</sub> &isin; S.
* @param  weights the correspoinding costs for links, i.e.,
* w<sub>i</sub> &rarr; l<sub>i</sub>.
* @param counters the corresponding times that that links
```

```
* <i>l<sub>i</sub></i> is in the subsets s<sub>i</sub>.
* @return a minimal weighted hitting set for set <i>S</i>
* @see  MySet.java
* @since V 1.0 March / 25 / 2009
* @author Christian Servin
*
**/

public static String appy_MWHS(String[] links, LinkedList<Set> S,
                                    double[] weights, int[] counters, int posT){
        System.out.println("Paths: "+S);
        String tmp_links = "{";
        Set<String> minimal = new LinkedHashSet<String>();
        boolean flag;  // this flag is needed to notify that links
                       // were removed from S
        updateTheCounter(S);

        for (int i = 0; CONTADOR > 0 ; i++) {
                int pos = posT == -1 ? pos = findChepestLinks(counters, weights) : posT;
                flag = false;
                for (int j = 0; j < S.size() ; j++) {
                        if(S.get(j).contains(links[ pos ])){
                                flag = true;
                                minimal.add(links[ pos ]);
                                S.remove(j);
                                counters = updatecounters( counters , pos );
                                j--;
                        }
```

```
                }
                counters = updatecounters( counters , pos );
                String paths = "";
                for(Set s : S)
                        paths+=s;
                // make sure the element was removed from the set
                if(flag == true){
                        //------------ cosmetics ------------
                        tmp_links+= tmp_links.length() == 1? links[pos] : ", "+links[pos];
                        //--------------------------------
                }
                System.out.println((i+1)+"\t "+tmp_links+"}"+"\t\t"+paths+
                                "\t\t"+minimal);
                updateTheCounter(S);
                posT = -1;  // in case we initiate with posT, we reset ''pos''
                                // for next iteration
        }
        return tmp_links+"}";
}
/**
 * <b>trace</b><br>
 * @param  links: containing all the links in the network
 * @param  S a set of subsets {s<sub>1</sub>, ..., s<sub>k</sub>}
 * such that Union of s<sub>i</sub> &isin; S.
 * @param counters the corresponding times that that links
 * <i>l<sub>i</sub></i> is in the subsets s<sub>i</sub>.
 * @param posT: optional parameter, sets the position as the link
 * with the highest counter
```

```
 * @return void
 * @see  MySet.java
 * @since V 1.0
 * @author Christian Servin
 **/
public static void trace(String[] links, LinkedList<Set> S,
                int[] counters, int posT){
        String tmp_links = "{";
        Set<String> minimal = new LinkedHashSet<String>();
        boolean flag;

        System.out.println("Paths: "+S);
        System.out.println("step\t links\t\t Paths remaining \t\t partial min");
        System.out.println("----------------------------------------------");

        updateTheCounter(S);
        Stack stack = new Stack();

        for (int i = 0; CONTADOR > 0 ; i++) {
                int pos = posT == -1 ? pos = findHighest( counters ) : posT;
                flag = false;
                for (int j = 0; j < S.size() ; j++) {
                        if(S.get(j).contains(links[ pos ])){
                                flag = true;
                                minimal.add(links[ pos ]);
                                S.remove(j);
                                counters = updatecounters( counters , pos );
                        }
```

```
                }
                counters = updatecounters( counters , pos );



                String paths = "";
                for(Set s : S)
                        paths+=s;

                // make sure the element was removed from the set
                if(flag == true){
                        //------- cosmetics -------
                        tmp_links+= tmp_links.length() == 1? links[pos] : ", "+links[pos];
                        //------------------------------
                }
                System.out.println((i+1)+"\t "+tmp_links+"}"+
                                        "\t\t"+paths+"\t\t"+minimal);
                updateTheCounter(S);
                posT = -1;   // in case we initiate with posT, we reset '
                                // 'pos'' for next iteration
        }
}



/**
 * <b>appy_MHS</b><br>
 *
 * <b>Instance: </b>A set system <i>(U,S)</i> with &cup;
 * <sub>s &isin; S</sub>S = U weights c: s &rarr; R.<br>
```

```
* <b>Task: </b> Find a minimum weight set cover for ...
* @param  links: containing all the links in the network
* @param  S a set of subsets {s<sub>1</sub>, ..., s<sub>k</sub>}
* such that Union of s<sub>i</sub> &isin; S.
* @param counters the corresponding times that that links
* <i>l<sub>i</sub></i>
* is in the subsets s<sub>i</sub>.
* @param posT: optional parameter. posT represents the position
* desired to begin the MHS algorithm.
* @see  MySet.java
* @since V 1.0 March / 25 / 2009
* @author Christian Servin
**/

public static String appy_MHS(String[] links, LinkedList<Set> S,
                                        int[] counters, int posT){
        System.out.println("Paths: "+S);
        String tmp_links = "{";
        Set<String> minimal = new LinkedHashSet<String>();
        boolean flag;  // this flag is needed to notify that links
                       // were removed from S
        updateTheCounter(S);

        for (int i = 0; CONTADOR > 0 ; i++) {
                int pos = posT==-1 ? pos = findHighest(counters ) : posT;
                flag = false;
                for (int j = 0; j < S.size() ; j++) {
                        if(S.get(j).contains(links[ pos ])){
```

```
                                flag = true;
                                minimal.add(links[ pos ]);
                                S.remove(j);
                                counters = updatecounters( counters , pos );
                                j--;
                        }
                }
                counters = updatecounters( counters , pos );
                String paths = "";
                for(Set s : S)
                        paths+=s;
                // make sure the element was removed from the set
                if(flag == true){
                        //------- cosmetics ----------
                        tmp_links+= tmp_links.length() == 1? links[pos] : ", "+links[pos];
                        //------------------------
                }
                System.out.println((i+1)+"\t "+tmp_links+"}"+"\t\t"+
                                        paths+"\t\t"+minimal);
                updateTheCounter(S);
                posT = -1;  // in case we initiate with posT,
                                // we reset ''pos'' for next iteration
        }
        return tmp_links+"}";
}


public static void main (String [] args){
```

```
String[] links = new String[]{"L1","L2","L3","L4","L5"};
Set<String> total_links = new LinkedHashSet<String>();
MySet.addElements(total_links, links);
Set s1 = new LinkedHashSet();
Set s2 = new LinkedHashSet();
Set s3 = new LinkedHashSet();
Set s4 = new LinkedHashSet();
int[] counters = new int[]{1,2,2,2,1};
double[] weights = new double[]{1,5,5,5,1};



System.out.println("step\t links\t\t Paths remaining \t\t partial_min");
System.out.println("-----------------------------------------");
Set<String> total_min_set = new LinkedHashSet<String>();
for(int i = 0 ; i < ITERATIONS ; i++){
        MySet.addElements(s1, new String[]{"L1","L2"});
        MySet.addElements(s2, new String[]{"L2","L3"});
        MySet.addElements(s3, new String[]{"L3","L4"});
        MySet.addElements(s4, new String[]{"L4","L5"});
        LinkedList<Set> total = new LinkedList<Set>();
        total.add(s1);total.add(s2);total.add(s3);total.add(s4);
        counters = new int[]{1,2,2,2,1};
        weights = new double[]{1,5,1,5,1};
//  total_min_set.add(appy_MWHS(links, total,weights, counters, -1));
        total_min_set.add(appy_MHS(links, total, counters, -1));
        System.out.println("-----------------------------------");
        System.out.println("--------------------");
```

```
        }
        System.out.println("==============================");
        System.out.println("Minimal Weighted Set Cover");
        for(String s : total_min_set){
                System.out.println(s);
        }
        System.out.println("==============================");
        System.out.println("Total Cost: ");


}
```

Parser and other utilities

```
import java.util.*;
import java.io.*;
import java.util.regex.Pattern;

public final class Parser {


/**
 * <b>readLinks parser</b><br>
 * Parser reads <tt>output</tt> file (where Prolog's results are stored)
 * and transforms the result into the user's format.
 *
 *<blockquote><pre>
 * <tt>[[cp1, cp2, 24], [cp2, cp6, 24], [cp6, cp8, 24], [cp8, cp9, 24]] ;</tt>
 * ...
 * to
 * P1 = {L<sub>cp1cp2</sub>, ..., L<sub>cp8cp9</sub>}
 * ...
 * Pk = { ... }
 *</pre></blockquote>
 *<p>
 * @param void
 * @return a LinkedList of Paths, containing the corresponding CVP paths
 * @see Path.java, Link.java
 * @throws IOException in case the file <tt>out.txt</tt> is not found
 **/
public static LinkedList<Path> readLinks(){
```

```
String file = "out.txt";
LinkedList<Path> paths = new LinkedList<Path>();
Path path;
Link link = null;
String src,dest,src_level,dest_level,weight,pathname;

try{
        BufferedReader inFile = new BufferedReader( new FileReader(file));
        String line = inFile.readLine();
        int pathnumber = 1;

        StringTokenizer st;
        while (line != null){
                pathname = "P"+pathnumber+" = ";
                path = new Path(pathname);
                st = new StringTokenizer(line,"[,] ;",false);
                while(st.hasMoreTokens()){
                        src = st.nextToken();
                        dest = st.nextToken();
                        weight = st.nextToken();
                        src_level = "";
                        dest_level = "";
                        link = new Link(src,dest,src_level,dest_level,weight);
                        path.add(link);
                }
                line = inFile.readLine();
                pathnumber++;
                paths.add(path);
```

```
                }
                inFile.close();
                paths.removeLast(); // I dont know why i still add the last part!!
        }
        catch(Exception e){ e.printStackTrace(); }

        return paths;
}

/**
 * union
 * A unnion of two sets, takes two linkedlist containing Strings
 * and returns an ordered list containing the union of L1 and L2
 * @param L1 a LinkedList containing elements in one set
 * @param L2 a LinkedList containing elements in one set
 * @return L3 a LinkedList set containing the union of L1 and L2
 **/
public static LinkedList<String> union(LinkedList<String> L1,
                                       LinkedList<String> L2){
        int ind1 = 0;
        int ind2 = 0;
        int cont = 0;
        LinkedList<String> result = new LinkedList<String>();
        while ((ind1 < L1.size()) || (ind2 < L2.size())){
                cont ++;
                if (ind1 == L1.size()){
                        result.add (L2.get(ind2));
                        ind2++;
```

```
                }
                else if (ind2 == L2.size()){
                        result.add (L1.get(ind1));
                        ind1 ++;
                }
                else if (L1.get(ind1).compareTo(L2.get(ind2)) < 0 ){
                        result.add(L1.get(ind1));
                        ind1++;
                }
                else if (L1.get(ind1).compareTo(L2.get(ind2))== 0){
                        result.add(L1.get(ind1));
                        ind1 ++;
                        ind2 ++;
                }
                else{
                        result.add(L2.get(ind2));
                        ind2 ++;
                }
        }
        return result;
}
/**
 * intersection
 * The intesection of two sets, takes two linkedlist containing Strings
 * and returns an ordered list containing the intersection of L1 and L2
 * @param L1 a LinkedList containing elements in one set
 * @param L2 a LinkedList containing elements in one set
 * @return L3 a LinkedList set containing the intersection of L1 and L2
```

```
 **/

public static LinkedList<String> intersection(LinkedList<String> L1,
                                                LinkedList<String> L2){
        int ind1 = 0;
        int ind2 = 0;
        LinkedList<String> result = new LinkedList<String>();

        while ((ind1 < L1.size()) || (ind2 < L2.size())){
                if (ind1 == L1.size() || ind2 == L2.size()){
                        return result;
                }
                else if (L1.get(ind1).compareTo(L2.get(ind2)) == 0 ){
                        result.add(L1.get(ind1));
                        ind1 ++;
                        ind2 ++;
                }
                else if ( L1.get(ind1).compareTo(L2.get(ind2)) < 0 )
                                ind1 ++;
                else
                        ind2 ++;
        }
        return result;
}
/**
 * difference
 * The difference of two sets, takes two linkedlist containing Strings
 * and returns an ordered list containing the difference of L1 and L2
```

```
 * @param L1 a LinkedList containing elements in one set
 * @param L2 a LinkedList containing elements in one set
 * @return L3 a LinkedList set containing the difference of L1 and L2
 **/

public static LinkedList<String> difference(LinkedList<String> L1, LinkedList<String> L2){
        int ind1 = 0;
        int ind2 = 0;
        LinkedList<String> result = new LinkedList<String>();
        while ((ind1 < L1.size()) || (ind2 < L2.size())){
                if (ind1 == L1.size()) {
                        return result;
                }
                else if (ind2 == L2.size()){
                        result.add (L1.get(ind1));
                        ind1 ++;
                }
                else if (L1.get(ind1).compareTo(L2.get(ind2)) == 0){
                        ind1 ++;
                        ind2 ++;
                }
                else if (L1.get(ind1).compareTo(L2.get(ind2)) < 0){
                        result.add(L1.get(ind1));
                        ind1 ++;
                }
                else
                        ind2 ++;
        }
```

```
        return result;
}
}

/**
 * Class Link
 * This class stores Link objects.
 **/

class Link{

String src, dest, src_level, dest_level, name;
double weight;

public Link(String src, String dest, String src_level, String dest_level, double weight){
        this.name  = src+dest+src_level;
        this.src   = src;
        this.dest   = dest;
        this.src_level = src_level;
        this.dest_level = dest_level;
        this.weight  = weight;
}
public Link(String src, String dest, String src_level, String dest_level, String weight){
        this.name  = src+dest+src_level;
        this.src   = src;
        this.dest   = dest;
        this.src_level = src_level;
        this.dest_level = dest_level;
```

```
        this.weight  = Double.parseDouble(weight);
}

public String toString(){
        return name;
}

}

/*
 * Class Path
 * The class path holds a collection of links
 *@see Link
 */
class Path{
        LinkedList<Link> links;
        String name;

        public Path(String name){
                links = new LinkedList<Link>();
                this.name = name;
        }

        public boolean add(Link l){
                return links.add(l);
        }

        public String toString(){
```

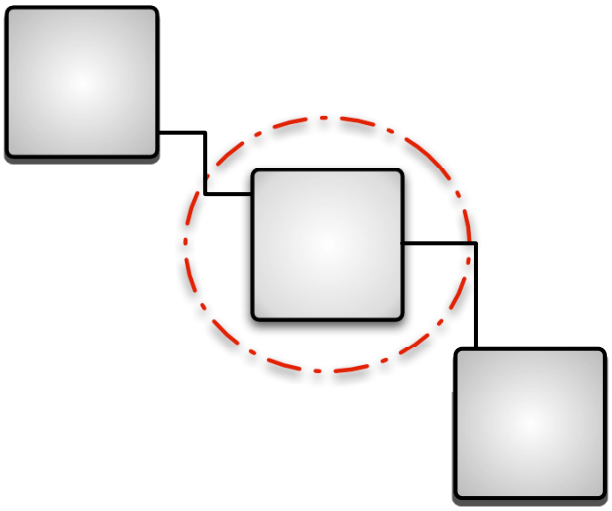
```
                String ans = name+"{";
                for(Link s : links){
                        ans+=s+",";
                }
                ans = ans.substring(0,ans.length()-2);
                return ans+"}";
        }

}
```

Appendix D

CVP User's Manual

CVP Detection Tool User's Manual



Christian Servin

Computer Science Department.
The University of Texas at El Paso (UTEP)

Computer Science Department 500 W. University Ave. El Paso TX, 79968. **email:** cservin@gmail.com **URL** http://www.cs.utep.edu/students/christians

Christian Servin
Department of Computer Science 124
University of Texas at El Paso
500 W. University Ave,
El Paso, TX 79968–0518
e-mail: cservin@gmail.com

Acknowledgements

I would like to thank Dr. Martine Ceberio and Dr. Eric Freudenthal for their support, guidance, and to provide their expertise in the disciplines of constraint programming and computer security.
I'm grateful to Dr. Stefano Bistarelli for many interesting discussions on constraint programming and the cascade vulnerability problem, for his feedback and guidance in the construction of this tool.

Contents

1 Distribution

CVP Tool 0.1
BSD License

The following terms apply to all files associated with this software that are mentionned in the file manifest.txt.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the UTEP nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

2 Installation

Software and Hardware Requirements

CVP Tool was developed and tested with the following system specifications:

- UNIX (Mac OS X) as Operating System,
- Java (TM) 2 Runtime Environment, Standard Edition (build 1.5.0_07-164)
- SWI-Prolog version 5.6.22 for i386-darwin8.8.1

Download the latest version of *J2SE* from java.sun.com site,
http://java.sun.com/j2se/1.4.2/download.html, and install it.

Download and install Prolog. It works with all prologs but authors use SWI-Prolog[1]. For UNIX users install e.g., `/opt/local/bin/swipl`

Installation

Since CVP Tool is a Java/Prolog-based application, it is necessary to install Java and Prolog in the computer. To install the CVP follow these steps:

1. Create a directory in your system.
2. Unpack the CVPTool.zip in the directory
3. Run CVPTool.jar

Use of CVP Tool

The CVP Tool has a graphical user interface (GUI) that allows the user interact with the tool. The tool has commands (e.g., save/load network) that allow the user to use the tool depending on the action allowed by the working *tab*.

[1] SWI-Prolog: http://www.swi-prolog.org/

3 Overview

Overview

The objective of this tool is to detect the existence of a cascade vulnerability problem in a network, identify the element(s) that are generating this problem, and properly eliminate the minimum elements(s) that provoke this problem in the network. The intention of building the CVP Tool, is to assess the work presented in Detecting and eliminating the cascade vulnerability problem from multi-level security networks using soft constraints [**?**], A Soft Constraint-Based Approach to the Cascade Vulnerability Problem [**?**], and An Optimization Approach using Soft Constraints for the Cascade Vulnerability Problem [**?**].

Features/Functionalities

This version of CVP Tool includes the following features and functionalities:

- Detect the existence of a CVP in a network.
- Extract the cascading path generators from the network in case there is a CVP.
- Provide the element(s) that are the least expensive to eliminate in a network.
- Represent graphically the network and its computers.

Definitions and Abbreviations

- **Assurance Ratings:**
- **Effort:** the level of work done by a intruder to consume a systemÕs resources
- **CVP:** Cascade Vulnerability Problem
- **Information Levels:** The type of information in a system, e.g., Top Secret (T), Secret(S), Classified (C).
- **MLS:** Multi level system
- **MWHS:** Minimum Weighted Hitting Set
- **Risk:** Notion taken from [1] and [2], where the risk represents .....
- **TCSEC:** Trusted Computer System Evaluation Criteria
- **TNI:** Trusted Network interpretation

Introduction

A Cascade Vulnerability Problem (CVP) arises in approved-trusted networks. An approved network is a network such that every computer that belongs to it agrees to own a security assurance level. A CVP arises when an intruder takes advantage of the network connectivity to compromise information across a range of sensitivity levels, and the span of accessed levels exceeds the accreditation range of any computer. Let us illustrate a potential CVP in a MLS network in Figure 1.

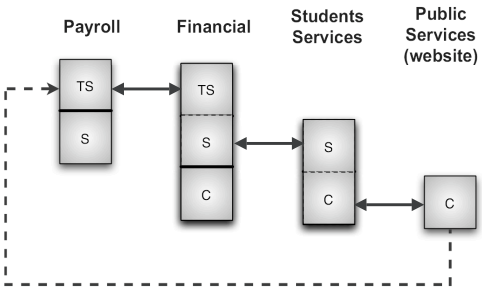


Figure 1: A potential CVP. Information in higher clearance level (*top secret*) leaks to a lower clearance level (*classified*) through this MLS network.

The CVP detector Tool is a Java based application used to detect the CVPs in a network and to provide options to eliminate this problem. On the one hand, the tool takes advantage of Java's operating system portability, object oriented manipulation, and graphical libraries; and on the other hand, the tool has the ability to solve constraints in the network using the power of Prolog.

The CVP detector tool is composed of several panels that allows the user to interact with the tool. These panels are described in detail in the following sections.

The graphical user interface allows the user to interact with the tool such as configuring the computers in a network, as well to establish valuable connections between other computers systems.

The constraint solver is a prolog program that processes the constraints stated from the graphical user interface and returns a set of all possible solutions that satisfies the constraints.

A general design of this architecture is depicted in Figure 3.

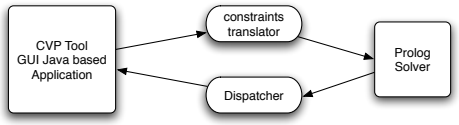


Figure 2: A CVP Tool Abstract level Design.

A Network and its representation.

A network is a set of computers. A computer owns at least one information level that can interact with others computers. For example: Let us consider a network of two computers: $C1$ and $C2$. The computer $C1$ owns information at level *Top Secret* and *Secret* and computer $C2$ only owns information at level *Secret*. Computer $C1$ and $C2$ share information at level *Secret* like is shown in Figure 3.

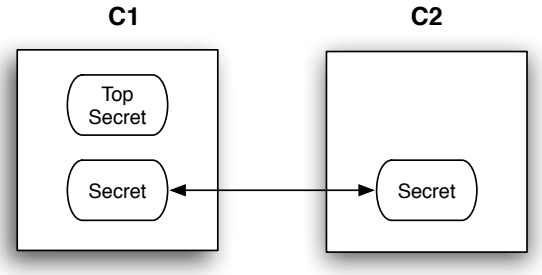


Figure 3: A Network composed of two computers. Both share information at level 'Secret'

4 Components

Tool Overview

This tool is composed of four main panels: 1. customize network, 2. configure network, 3. simulate network, and 4. the graphical representation of the network.

Customize Network

The Customize Network screen allows the user to add computers into a network. A computer may own several information levels. The user categorizes the information according to its sensibility, for instance, a computer may contain social security numbers (Top Secret) and the annual wages (Secret). The user must provide a distinguishable name for each computer added in the network. This is to identify the computer from others. In the same way, the user must specify the information levels that the computer can hold.

Each computer is accredited according to levels of assurance from the Trusted Computer System Evaluation Criteria [**?**]. Once the user specifies the information level, an assurance level will be assigned to the computer according to the sensitiveness of this information. These values encode the assurance matrix from [**?**].

In addition, there is a system's rank and a system's description that the user can optionally provide.

- *System's Name*: An alpha numeric name that represents the computer system.
- *Level Classification*: An information level (e.g., Top Secret, Secret, Classified)
- *System's Rank*: This is an optional value assigner to the computer system that can be used in case there is no Level Classification. For instance, in case a user assigns a monetary value to this computer to identify it as a more valuable computer than others.
- *System's Description*: A textual description for this system.

Example: Let us add a computer from the network shown in Figure 4. Computer 1, named as $c1$, owns *Top Secret* and *Secret* information, using the CVP Tool Customize Network Screen we can see:

Configure Network

Once the computers are added in a network using the Customize Network panel, the Configuration panel enables the establish a valued (weighted) connection between systems. A weighted connection is a value that the user gives to a connection between two computers. For example, consider the network shown in Figure 4.

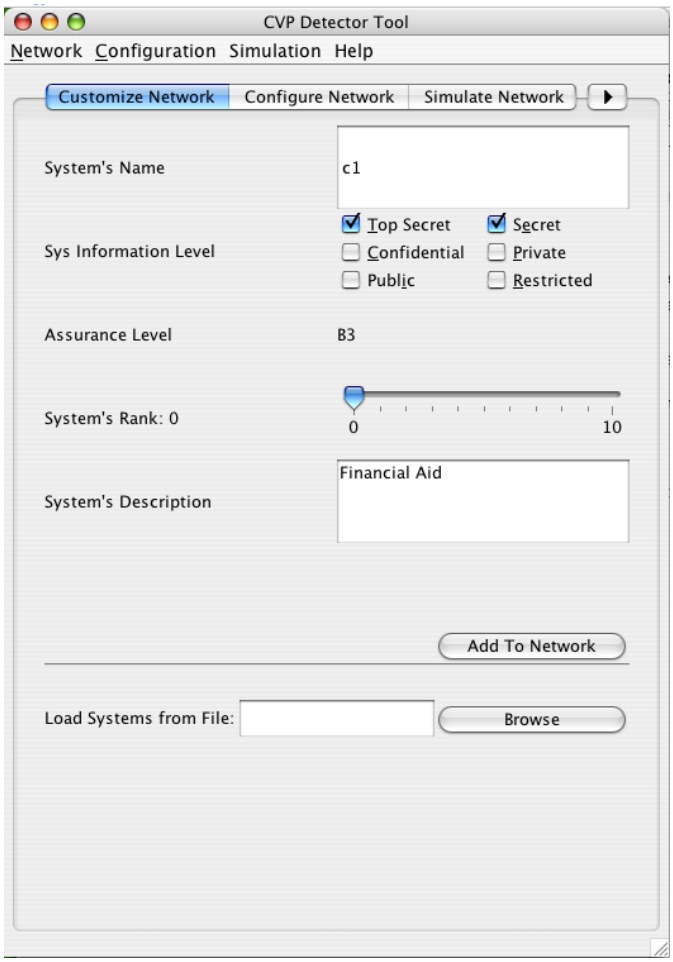


Figure 4: Customize Network Screen: Adding the Financial Department to the network called 'C1' that holds information at level Top Secret and Secret. The corresponding assurance values according to the TNI matrix is 'B3'.

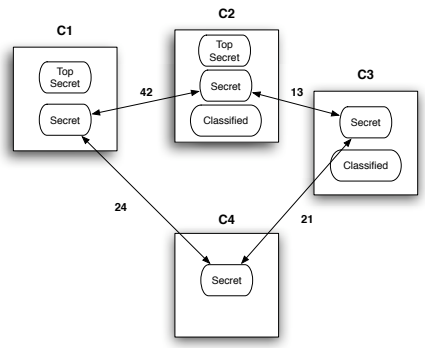


Figure 5: A weighted network composed of 4 computers

Using the Configuration panel, the user selects the computer where the information will start to flow from the Source List, and selects the destination from the Destination List as shown in the Figure 4. This panel extracts the computer information that is added from the customize panel and the connections established in the configuration panel, and generates a SWI-Prolog file with the information extracted from these panels. More information about adding and loading network see Section 6: from this manual.

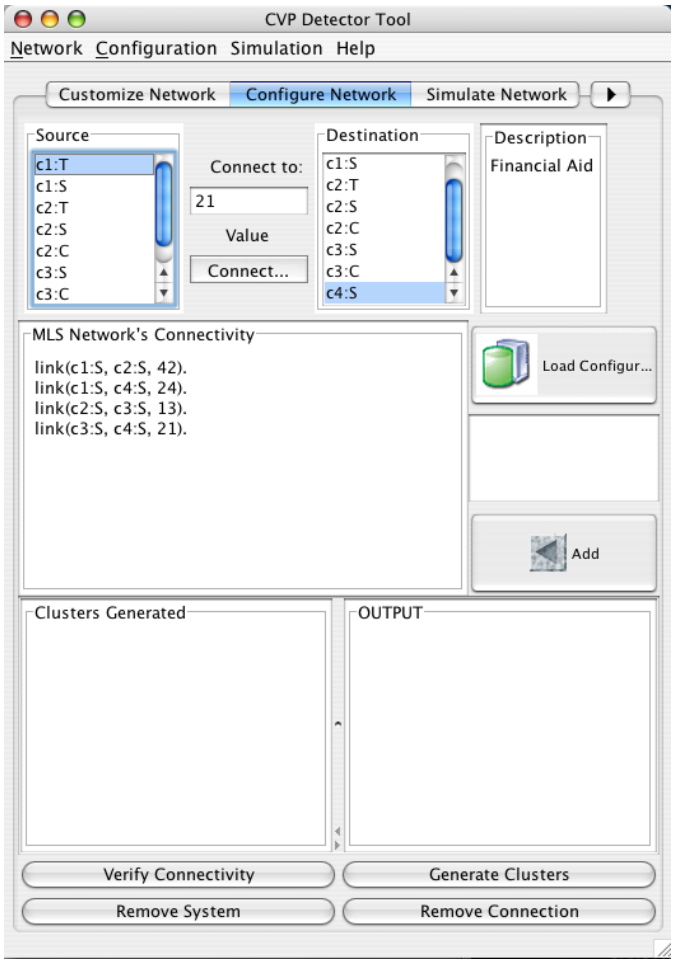


Figure 6: The Configuration Panel enables the creations of connections between systems and assign a value (weight) to that connection.

Constraint translator

This Configuration component, invokes a procedure that is in charge of extracting the *facts* and *constraints* from a given network. Consider the network from Figure 4. The network is composed of 5 computer with different sensitiveness compartments:
C1, owns a *top-secret* compartment;
C2, owns a *top-secret, secret*, and *classified* compartments;

C3, owns a *secret* and *classified* compartments;
C4, owns a *classified* compartment;
C5, owns a *secret* component.
these facts are translated and represented in Prolog facts called **computer** as follows:

```
computer(c1, [ t ] ).
computer(c2, [ t, s, c ] ).
computer(c3, [ t, s ] ).
computer(c4, [ c ] ).
```

In the same manner, the facts that represent the links between computers:
C1 is connected to C2 through the top-secret compartment, and this connection has a value of 42 Gb/s,
C2 is connected to C3 through the secret compartment, and this connection has a value of 13 Gb/s, and to C5 with a 2 Gb/s value,
C3 is connected to C4 through the classified compartment, and this connection has a value of 24 Gb/s,
are also represented as follows:

```
link(c1 , [ [c2, t, 42] ]).
link(c2 , [ [c3, s, 13], [c5, s, 2] ]).
link(c3 , [ [c4, c, 24] ]).
```

In Prolog words: the rule **link** takes two objects, a *computer source*, i.e. $c2$, and an *list composed of lists* (an adjacent list) that represents all connections to computer source, i.e.,

$$[[c3, s, 13], [c5, s, 2]]$$

These facts are written in a Prolog file and then passed to the *Prolog solver* to solve the constraints.

CVP Checker

Basically, once the constraints are extracted from the Constraint Translator procedure mentioned above, the procedure generates a `network.pl` file, that will be used as an input for the CVPChecker that will return a `'yes'` or a `'no'` depending if there is a vulnerability problem in the network.

Clusters Generator

This solver is a Prolog file executed by SWI-Prolog that takes as an input the file generated in the previous section and returns a set of paths η that are the least expensive links in the network to eliminate in order to avoid a CVP. This process is called by the `PrologSolver.pl`[2].

[2]NOTE: This is a work in progress, see Section 7. for more details

The entire procedure of the Prolog solver is demonstrated in the *Simulation Panel*. In this panel step is shown step by step how the algorithm is working, and the resulting set is displayed in the *Result text area*.

Algorithm Simulator

The main purpose of this panel is to simulate step-by-step the algorithm that is been executing. There are two algorithms implemented in this version: The *Minimal Hitting Set Cover* (MHSC) [**?**], and the *Minimal Weighted Hitting Set* (MWHS) [**?**].

In the Simulator panel, there are several fields displayed as shown in Figure 4.

- **List of clusters**: This list corresponds to the *set of the minimal elements* generated by the procedure mentioned in 4.
- **Element list**: lists the total connections (links) involved in the network.
- **Min Sets list**: lists all the sets of the minimal elements involved in current selected algorithm.
- **Links**: lists all the links that belong to the *Min Sets list* that is currently selected
- **Weight**: lists the corresponding weight from the link selected in *Links' list*.
- **List of Solutions**: component that keeps track of all the solutions generated by the algorithm,
- **Optimal Solutions**: component lists the best solution so far generated by the algorithm

- *run simulation button:* runs the algorithm selected with the list of connected systems as the input.
- *detect button:* initializes all the fields mentioned above.

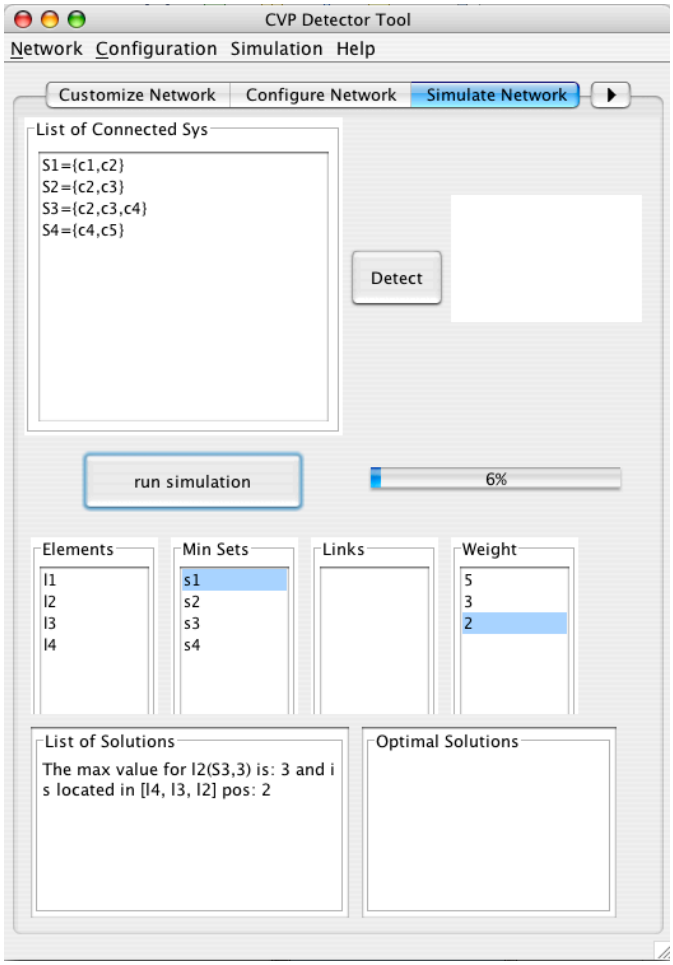


Figure 7: Simulate panel: simulates the specified algorithm to eliminate the CVP

5 Design and Components

- *CVPTool.java* JFrame is the main class that invokes all the texttt.class files. This is the main runner.
- *Customize.java:* JPanel that enables a user to add a computer into the network and specify the type of sensitivity information it owns, e.g., *Top-Secret, Secret, Classified*.
- *Configuration.java:* JPanel that enables the creations of connections between systems and assign a value (weight) to that connection.
- *Simulatior.java:* JPanel that shows the tracing of the algorithm in the network, and provides the possible solutions. The algorithm is selectable by the user (only two algorithms have been implemented in this first version).
- *PrologFileWriter.java:* Java class that extracts the computer information that is added from the customize panel and the connections established in the configuration panel, and generates a *SWI-Prolog* file with the information extracted from these panels.
- *JavaPrologDispatcher.java:* invokes SWI-Prolog and consults the file generated by the constraint translation component. This solver then processes the constraints to detect a CVP in the network. In the case that the detection constraints are satisfied (that indeed there exists a CVP in the network), the solver finds all the paths that are causing this cascading problem.
- *SolutionHandler.java:* opens a listener stream to the SWI-Prolog solver, and it stores all the output from the SWI-Prolog solver until it finishes to process all the constraints, then gives the results to the simulation panel.
- *Algorithms.java:* This component enables users to select the algorithm that will be applied in the elimination process of elements that generate the CVP. The chosen algorithm is used in the simulator and the results are displayed in the simulator panel. The algorithms in this first version of the application are implemented in Java, and their APIs are provided in case the user would like to incorporate another algorithm into the tool.

Architecture Design

The tool is an example of a *model-view-controller* (MVC) architectural pattern. The model is the representation of the algorithms used, e.g., algorithms that detect and eliminate the CVP. The viewer component are multiple GUIs that interact with the user, and the controller component handles the interaction between GUI and model. The MVC pattern is shown in Figure 8.

Figure 9 shows how the system components interact with each other.

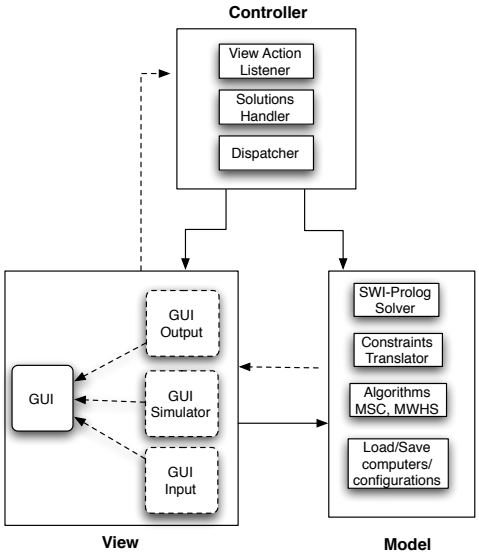


Figure 8: The model-view-controller pattern for the CVP simulator tool

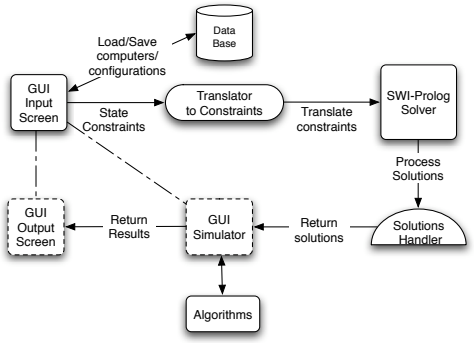


Figure 9: The CVP simulator tool architecture design

6 How to:

- **Choose a Different Algorithms** The tool allows the user to choose different algorithms. to simulate. To select an algorithm: *Main Menu > Simulation > Algorithms*. For more information about the specification on the algorithms please see `Algorithms.java`.

- **Load a network:** This tool facilitates the network input and provides a feature to load an entire network's configuration from a file. The file extension `.dat`, contains the information needed to add the entire network (e.g., computers and its properties s.a. name, description, etc). An *Add to network* button can be found in the *Customize* Panel. Alternatively, in: *Main Menu > Network > Load Entire Network*.

- **Load an existing network connectivity:** Loads an existing network's connectivity. This is the same as *loading a network*, with the addition of loading the connection between computers. The file extension `.config`. The *Load Configuration* button can be found in the *Configure* Panel. Alternatively, in: *Main Menu > Configuration > Load Connectivity*.

- **Remove System:** Removes a computer system from the network. A *Remove System* button can be found in the *Configure* Panel.

- **Remove Connection:** Removes an existing connectivity (a link) between computers. A *Remove Connection* button can be found in the *Configure* Panel. (Note: some bugs were found in this first version, avoid to use this).

- **Save a network:** The computers added in a network are saved in a `.dat` file. This file contains all the elements added from the *Customize* Panel. To save a current network: *Main Menu > Network > Save Network As*. The file will be called: *name*`.dat`

- **Save a network connectivity:** Saving a network connectivity works the same way as *Saving the Network*. If the users want to save the connectivity between computers, then shall use this mechanism. To save a current network's connectivity: *Main Menu > Configure Network > Save Connectivity*.

- **Save:** For any changed done in the network or connectivity, the *Save*, will automatically save recent changes. To save: *Main Menu > Network > Save*.

Error Messages

In this section we display the possible errors that can appear in the CVP Tool and a possible explanation how to resolve them.

- *Loading Error*: Appears when the user tries to load an existing network file with different extension. Please select a file with an extension `.dat` or `.config`.

Figure 10: Loading Error: Appears when attempting to load an incorrect file

7 Current State of the Tool

We are working in the algorithm to extract the cascading paths from the network. Here we will use the approach mentioned in [**?**]. There is a *Cluster Helper* that "simulates" the output from the "Extract cascading Paths" procedure.

Here is the diagram that depicts the current state of the tool.

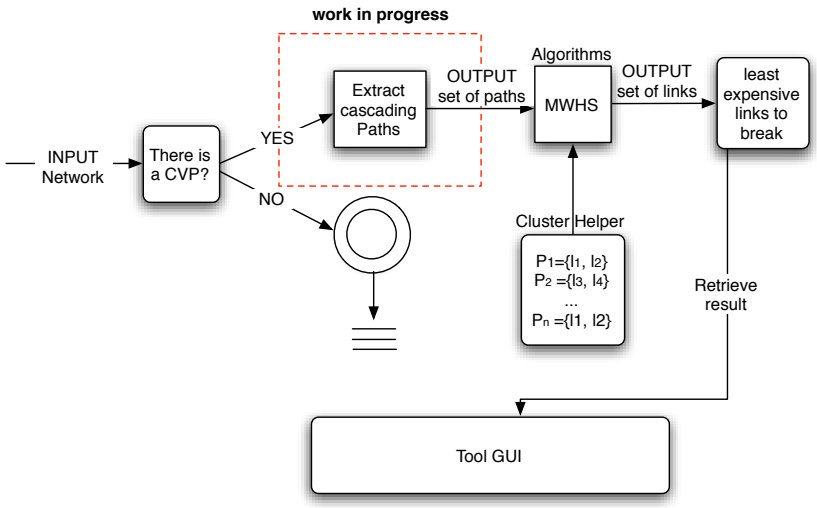


Figure 11: The current state of the CVP Tool

Work in Progress and Future work

Currently we are developing the algorithms to extract the minimal paths from a CVP network. For the next version of the CVP Tool, we anticipate to find, extract and eliminate the minimal and least expensive links in the network.

There is also a graphical representation of the network that we are planning to incorporate to the tool. We use an open source graph visualization library written in Java called **JGraph**[3]. This graphical representation shall allow to visualize the network and its current connections, in this way the user can appreciate the affected connectivity in the network.

Currently the tool's configuration allows to add a simple criterion to consider in this optimization problem. We also plan to implement a Multi-Criteria Decision Maker (MCDM) feature in the tool, to consider different weights in the problem.

[3]Java Graph Visualization and Layout: http://www.jgraph.com/

Curriculum Vitae

Christian Servín Meneses was born on February 13 of 1982. The first son of Jesús Servín and Marisela Servín. He graduated from Instituto Tecnológico y de Estudios Superiores de Monterrey (ITESM) in the city of Juárez México,

In the Fall 1999 he was accepted in the University of Texas at El Paso (UTEP), and in the fall of 2001 he joined the Computer Science program. During his undergraduate he worked as technical support in the systems department at the UTEP Library. He also worked several years in the Instructional Support Services (ISS) as a web applications and educational software developer.

He was part of several research groups in CS such as: the Interaction Systems Group (ISG), Java Modeling Language Group (JML), Theoretical Research and its Applications in Computer Science (TRACS), the Robust Systems Group, and the Constraint Research and Reading Group (CR^2G).

He played several roles and responsibilities in the Association of Computing Machinery (ACM) university student chapter during his undergraduate. He has been an active member since 2001, outreach chair from 2002-03, treasurer from 2003-04, and president from 2004-05.

During three summers on his undergraduate he received fellowships, internships and research awards to participate in research challenges that provided him the opportunity to increase his problem solving skills and intensively increase his research activities.

During these summers of 2003 thru 2005 he traveled to UT San Antonio, UT Austin and the Arctic Region Supercomputing Center (ARSC) in Fairbanks Alaska to develop research that related computer constraints and security. These internships motivated him to formalize his goals in life, that are become a Ph.D. in Computer Science in the area of Computer Security Constraints.

He officially received his bachelor's degree in Computer Science in the Fall of 2005. At

the same time, he was awarded with the NSF LSAMP Bridge to the Doctorate fellowship, that supported his graduate studies. During this time he worked under the supervision of Dr. Martine Ceberio and Dr. Eric Freudenthal in the subject of "computer security and constraint solving". In 2006 they began a collaboration with Dr. Stefano Bistarelli from Italy, leading to what now is known as "optimizing the cascade vulnerability problem." Currently, Servín is a Ph.D. student in the department of Computer Science, working for the Center of Science, Technology, Ethics and Policy (CSTEP).

Servín was selected by the college of graduate school to serve as the Graduate School Banner Bearer – a position of honor during the spring 2009 commencement.

Permanent address: 2801 North Kansas Apt 3
El Paso, Texas 79902-2537

This thesis was typed by the author.