THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

ADAPTIVE CASE BASED SECURITY FRAMEWORK FOR CRITICAL INFRASTRUCTURE PROTECTION

By

ERBIL YILMAZ

A thesis submitted to the
Department of Computer
Science in partial fulfillment of
the requirements for the degree
of Master of Science

Degree Awarded:
Fall Semester, 2002

The members of the Committee approve the

thesis of Erbil Yilmaz defended on Friday, November the 22nd .

Daniel G. Schwartz
Professor Directing Thesis

Sara Stoecklin
Committee Member

Alec Yasinsac
Committee Member

To Gorkem Ozer

ACKNOWLEDGEMENTS

I give my sincerest thanks to my major professor, Dr. Daniel G. Schwartz, for supervising my thesis and for his continuous support and constructive criticism towards the completion of this thesis. I would like to thank both Dr. Sara Stoecklin and Dr. Daniel G. Schwartz for weekly research meetings, and useful suggestions in my research field. Without their great help, this work may not have been completed. Also I want to express my gratitude to my committee members, Dr. Sara Stoecklin and Dr. Alec Yasinsac for working with me in reviewing my thesis and attending my defense.

Finally, my greatest gratitude goes to Gorkem Ozer for her endless support, encouragement and for always being there to cheer me up, and my family for their continuous support throughout my education.

This work was supported by the US Army Research Office grant number DAAD19-01-1-0502.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Even though modern information systems provide fairly good service under normal conditions, in the existence of malicious activity or unforeseen failures these systems have flaws, imperfections and vulnerabilities. The systems essential to the minimum operation of government and economy, called critical infrastructures, are particularly important since attacks on these systems may result in serious consequences to the nation. This paper presents an adaptive case-based reasoning security framework that can be applied to any critical infrastructure domain with minimum extra coding effort. Also the system is specifically tailored for network intrusion detection problems as a proof of concept. Snort IDS rule knowledge is ejected into a case library so that the final system is at least as powerful as Snort IDS.

CHAPTER 1

INTRODUCTION

1.1 Motivation

In recent years the US government has drawn increasing attention to the problem of protecting the nation's critical infrastructures. Some of the actions that have been taken are as follows [10, 11]:

- July 1996: The President's Commission on Critical Infrastructure Protection (CIP) was established. This was the first national effort to address the vulnerabilities of the new information age.
- October 1997: The final report of the President's Commission on CIP was released [12]. In this report, increasing dependence on critical infrastructures was stressed and a wide variety of threats were defined.
- May 1998: President Clinton announced two new directives, PDD-62 and PDD-63, highlighting the growing range of unconventional threats and focused on protecting the nation's critical infrastructures.
- October 2001: President Bush created the Office of Homeland Security.
- June 2002: President Bush proposes to create a new Department of Homeland Security, the most significant transformation of the U.S. government in over a half-century.

In the 1997 report [12], critical infrastructures are defined as "systems whose incapacity or destruction would have a debilitating impact on the defense or the economic security of the nation." The infrastructures are classified into eight categories:

- Information and Communications
- Electrical Power Systems
- Gas and Oil Production, Storage and Transportation
- Banking and Finance
- Transportation
- Water Supply Systems
- Emergency Services
- Continuity of Government Services

Information systems were put on the top of the critical infrastructures list because of the severity of the risk and dependence of other critical infrastructures on such systems. In addition, six topical categories for research and development were identified:

- Information Assurance
- Monitoring and Threat Detection
- Vulnerability Assessment and System Analysis
- Risk Management and Decision Support
- Protection and Mitigation
- Contingency Planning, Incident Response, and Recovery

The research reported in this thesis is aimed at defining a generic, domain-independent case-based reasoning framework applicable to a variety of problems occurring in all the indicated infrastructures and research categories. Briefly, case-based reasoning is a method of recording and benefiting from past experiences, so that former problems and their solutions can be recalled and reused as appropriate for responding to new occurrences of the same, or similar, problems.

There are many potential benefits of such a generic framework. Once it has been created at this level of abstraction, the communication between specific instances of this framework will be easier. Since a case-based reasoning methodology is used, each instance of the framework will learn through its experiences. Furthermore, via communication between instances of different categories (e.g., vulnerability analysis and intrusion detection) the learning process will produce a higher level of certainty. For example, if the system is used for attack detection, the overall system will discover the attacks that no one of the instances can realize by itself.

While the resulting system is expected to have a wide range of applications, the present work is focused more specifically, for proof of concept, on the problem of monitoring and threat detection for information systems. In particular, the framework is customized to replicate the functionality of Snort [27] intrusion detection system as explained in chapter 5.

1.2 Computers and Critical Infrastructures

As mentioned, Information Systems were put at the top of the list in the 1997 President's report [12]. This is because computer and networking technologies have evolved to a point that information systems, as a product of such technologies, have become an essential part of our lives. After the 1980s, with the explosion in computer connectivity, especially in the form of the Internet, the way people communicate, government works, and people do business have begun to be shaped by these technologies. Not only do individuals rely on these systems for work, leisure, communication, and many other purposes, but also in some cases, companies, large organizations, governments, or an entire society depend on their continued operations. Air traffic systems, telecommunication systems, and power distribution control systems are counted as some examples of these crucial systems that are generally referred to as critical information systems.

From the government's perspective, critical infrastructures, supported by critical information systems, are the systems that are essential for the minimal operation of the economy and government. Even though these systems provide fairly good service under normal conditions, in the presence of malicious activity or unforeseen failures, they are seen to have flaws, imperfections, and vulnerabilities. When the degree of societal dependence on these systems is considered, the impact of their failure on society can be enormous. As an example, even a few hours of effective denial of service attack to certain parts of a banking system may cause very serious monetary damage. The problem becomes more serious if one takes into consideration the growing risk of cyber attacks on these systems by potential adversaries, such as criminal organizations, terrorist groups, or insiders. Such systems become increasingly attractive as targets for attack due to society's increasing dependence on them.

In the interests of improving the efficiency of the various systems by taking advantage of the ongoing advances in information technology, critical infrastructures have become increasingly automated and interconnected. Further, most of the infrastructures are composed of many interconnected networks within themselves. The complexity and connectivity of these networks carrying critical information introduce vulnerabilities and threats to the entire infrastructure. For example, one system may be used as a base to attack other systems, or weakness in one system may be used as an entrance point to other systems.

Thus, critical infrastructure protection is an open problem for researchers working on information security. In the past five years, research activities in the area of computer security for military and intelligence systems have been greatly expanded due to the infusion of governmental sponsorship support. Some of these research activities, for example [38, 39, 40], apply also to the critical infrastructure problem, but much still remains to be done.

1.3 Definition of the Problem

In the early stages of the development of operating systems and computer networks the primary emphasis was on functionality and performance, and little thought was given to matters of security. As a result, these systems have evolved to such levels of complexity and interconnectivity that it is now perhaps impossible to retrofit them with modifications that will assure adequate security. Moreover, this problem has become much deeper due to the proliferation of insecure application programs produced by many different software vendors. Therefore, in the short-term, the only reasonable solution to computer security problems is to plug the holes in the systems as we discover them. A short-term solution is needed because of the immediacy and potential severity of the threats. This has prompted the exploration of the popular security solutions such as intrusion detection and vulnerability assessment, as well as many others.

In the long-term, however, we may expect to see the complete redesign and rebuilding of all system from the bottom up, including secure computer chips, secure operating systems, secure communication protocols, secure programming languages, and secure applications. In the end, one may expect the new systems to be inherently much more secure than those of today, but at the same time the need for intrusion detection and vulnerability analysis may also be expected to continue. No matter how well we design our systems, dedicated attackers may be expected to find new ways to break in.

Although not all the critical information systems are totally computer-based systems, they are heavily and increasingly dependent on computers. Furthermore, most of the information providing the current state of the system and history of errors is available in digital format even though the systems are different. Therefore, in this research we concentrate on processing of data and representation of information rather than gathering of evidence. We believe that a bottleneck in the CIP problem is the difficulty of processing the enormous amount of data available to the security community. When the amount of information that it is necessary to process and the complexity of the systems are considered, the need for an automated intelligent system becomes obvious.

1.4 Organization of Chapters

The rest of the chapters are organized as follows. The methodologies and technologies, used in this research, and design decisions for our security framework are presented in chapter 2. Chapter 3 contains the related work. In chapter 4, the proposed security framework is explained. As a proof of concept, the implementation of an instance of the framework, that replicates Snort's functionality, is given in chapter 5. Finally, the thesis is closed with the concluding remarks and issues for future work.

CHAPTER 2

BACKGROUND AND DESIGN DECISIONS

The framework created in this thesis can be easily customized into a domain specific security application according to needs of a particular problem domain. To achieve the required level of abstraction for this flexible framework, the knowledge representation scheme, learning methodology, programming language, and, more importantly, the design architecture for the software must be chosen carefully. In this decision process, works such as [6, 7, 8, 9, 13, 14, 16, 19] have been very influential. In this chapter, we briefly explain the methodologies and technologies used in the proposed framework and why their appropriateness for a generic adaptive software system. The first research area includes the artificial intelligence technique of case-based reasoning [4, 5]. The second research area includes the use of adaptive architectures and meta-data [16, 17]. The last area includes XML [28] and the Java programming language [14, 15].

2.1 Case Based Reasoning

2.1.1 Overview

In the early ages of artificial intelligence (AI), reasoning was generally modeled as a process of extracting conclusions from the facts about the problem domain by manipulating the domain specific knowledge. Expert, or rule-based, systems were the first successful AI applications that followed this paradigm. Rule-based systems model the knowledge about the domain in the form of if-then rules and perform reasoning by applying the principles of first-order logic to those rules.

In spite of the early successes, rule-based systems had many disadvantages. Among these, two are important. First, the explicit information needed to model the domain in the form of rules is difficult to obtain. To build a consistent and effective rule set requires considerable amount of investigation about the domain. This is known as the knowledge elicitation bottleneck. Second, even if the rule-based system is implemented, it is difficult to maintain as the domain evolves since rule extraction from data needs expert participation. A detailed discussion on these problems can be found in [4].

Case-based reasoning (CBR) was developed as a means to remedy these problems [4, 5]. CBR differs from rule-based systems in its primary source of knowledge. As opposed to the domain specific rules, the knowledge in CBR systems is stored in cases that represent past concrete experiences. Case-based reasoning systems do not require a model of the problem domain. Cases are problem-solution pairs that describe complete episodes, or full experiences. Thus, in CBR, both knowledge elicitation and knowledge maintenance amount simply to identifying new cases and adding these to the library. This can be handled by the typical user. Expert participation in extraction of rules or other information from generalizations of domain knowledge is not required. For example, in network intrusion detection problem, cases might represent full network packet information for the packets that are used in previous attacks. Therefore, such a case will have a feature for each field in the packet header and a feature for the packet payload. However, in order to create a rule for a rule-based system for the same problem, one has to extract relevant information from the network packet.

CBR solves new problems by adapting old solutions to meet the requirements of the new problem. The CBR approach was inspired by the recognition of the importance of memory (or remembering) in human reasoning processes. It was observed that people generally use the solutions of previously solved problems as a starting point for the solution of new problems. Also it is commonly accepted that the world is regular in the sense that similar problems have similar solutions. Furthermore, most of the problems that people encounter recur over time. In view of these facts, case-based reasoning was developed as a new problem-solving paradigm.

2.1.2 Case-Based Problem Solving

The case-based problem solving process involves navigating through the solutions in a "solution space," guided by the similarity of a given problem to those represented by the cases stored in a case library. This is illustrated in Figure 1, adopted from [4].

As a new problem is encountered, the CBR system searches for those cases in the case library whose problem descriptions are similar, according to some similarity metric, to that of the given problem. The solution(s) of the most similar case(s) is (are) then used as a starting point for devising a solution to the new problem. The CBR system creates a solution to the new problem by adapting the solutions from the cases that were retrieved. This adaptation process is sometimes automatic, but typically requires human assistance.

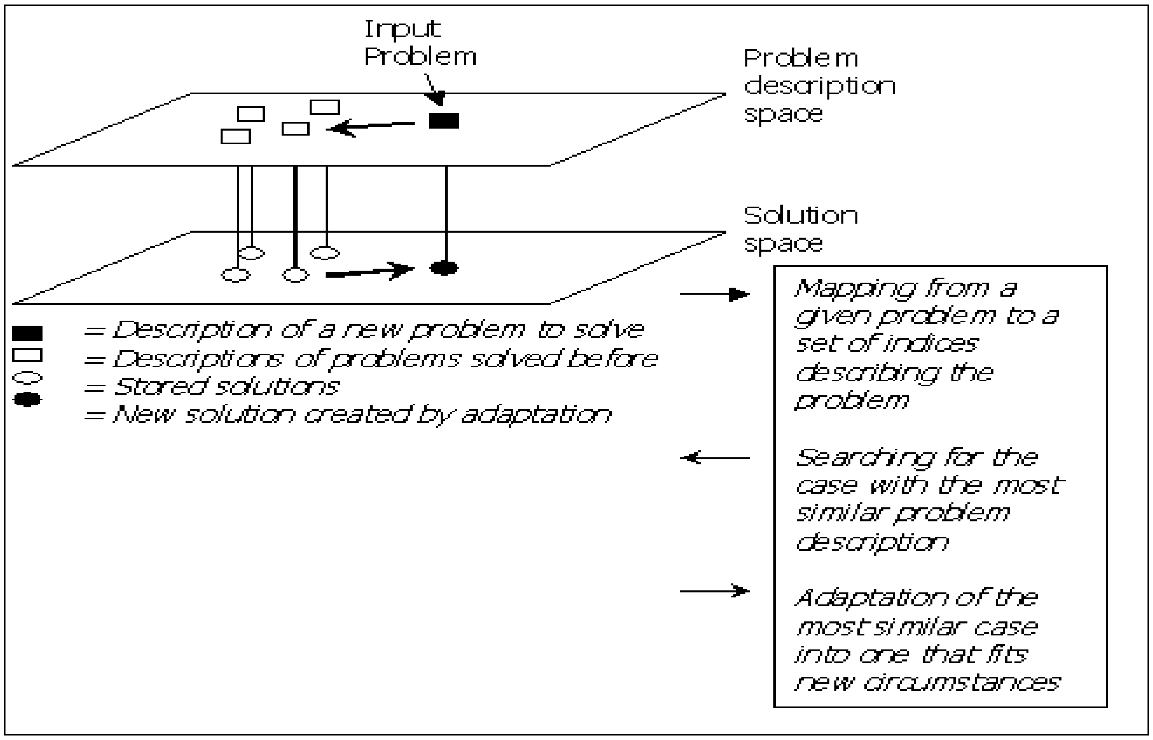

Input Problem
Problem description space
Solution space
= Description of a new problem to solve
= Descriptions of problems solved before
= Stored solutions
= New solution created by adaptation
Mapping from a given problem to a set of indices describing the problem
Searching for the case with the most similar problem description
Adaptation of the most similar case into one that fits new circumstances

Figure 1: Case Based Problem Solving

2.1.3 The CBR Cycle

Realization of the CBR methodology described above is normally accomplished via a cyclic CBR process comprising the following four "REs," as described in [23]:

1. RETRIEVE
2. REUSE
3. REVISE
4. RETAIN

The basic CBR process is depicted in Figure 2 [21, 22]. The most important part of a case-based reasoning system is its case archive, or case library, where previously encountered problems are stored as problem description/solution pairs. The problem description is a set of case features that characterize the problem. When a problem is perceived in the surrounding environment, it is transformed into a set of case features (step 1.0) and transferred to the search engine that extracts the similar cases from the case archive (step 2.0). This extraction process generally uses a similarity metric, which gives a value representing the degree of similarity between the given problem and an arbitrary case in the archive. The search engine then returns a set of top ranked cases according to their similarity to the given problem.

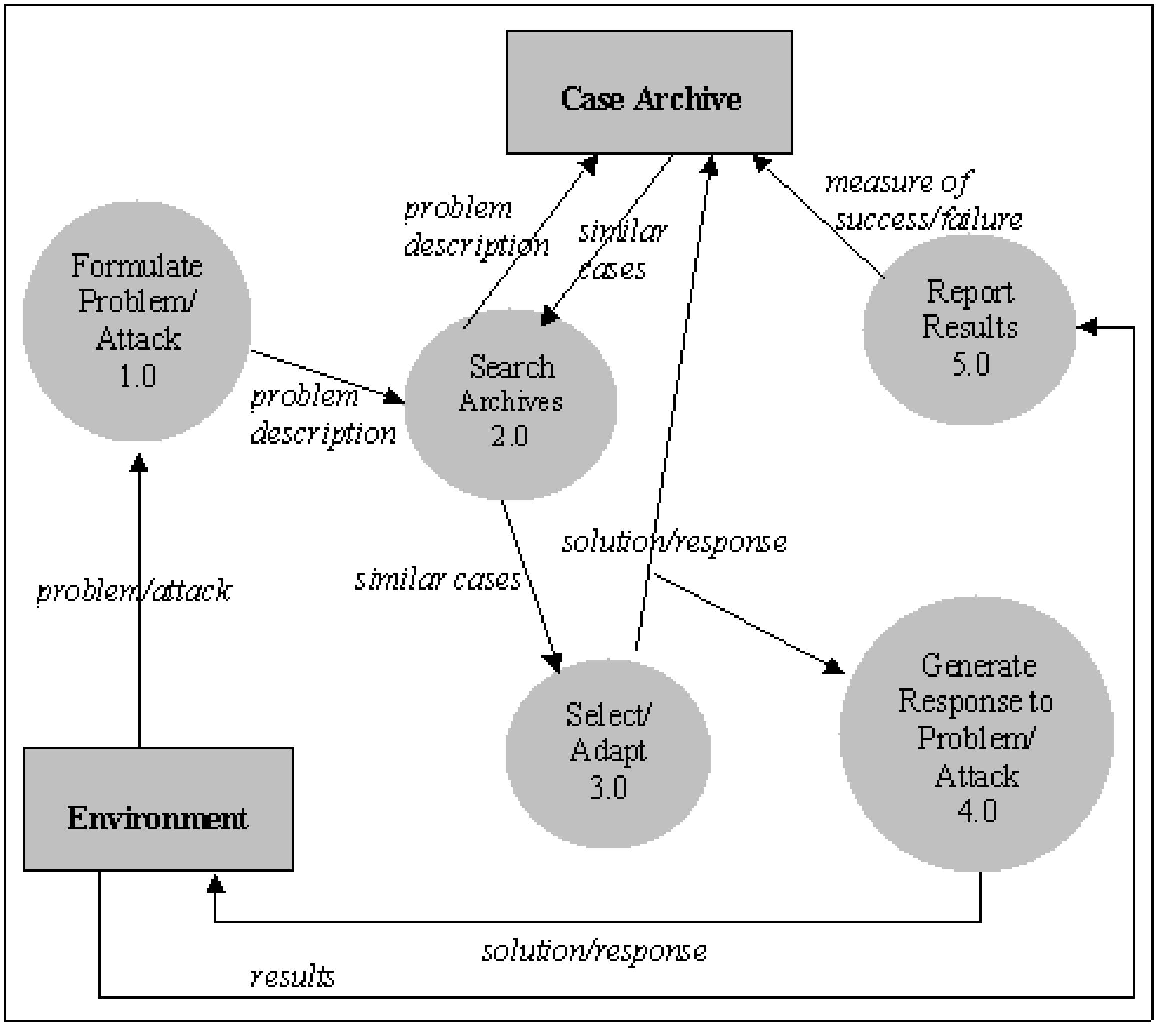


Figure 2: Case-Based Reasoning Cycle

At this stage, two different scenarios are possible: either one of the cases is selected (step 3.0), and its suggested action in its solution is taken (step 4.0), or a new solution is formulated from returned cases where their recommended solutions are used as a base point in the creation of new solution (step 3.0), and the action part of new solution is taken (step 4.0). If a new solution is created, the problem description and new solution pair is stored into case archive. In either case, the outcome of the action, as a measure of success or failure, is recorded with the case in the case archive (step 5.0). This measure is used in future case extraction processes, so that the performance of the system will improve.

2.1.4 Case Similarity and Nearest Neighbor Algorithm

The accuracy of the extraction process of similar cases to a given problem determines the success of the CBR system. If the system cannot find the best matching cases, the solution suggested by these cases will not be the most appropriate one for the given problem. The Nearest Neighbor Algorithm (NNA) [42] is extensively used in CBR systems for robust case extraction. This approach involves the assessment of similarity based on matching a weighted sum of features. The problem is described as a set of features in a case, as mentioned above. The features that are used in similarity assessment are called indexed-features, whereas the others are called non-indexed-features. The similarity between a given problem P and a case C is calculated as follows in NNA:

$$Sim(P,C) = \frac{\sum_{i=1}^{n} w_i \times sim(f_i^P, f_i^C)}{\sum_{i=1}^{n} w_i}$$

In this equation *w* is the importance weight of a feature and *sim* is the similarity function that calculates the similarity of the matching features of a case and a problem.

2.1.5 CBR for Security Framework

While the use of artificial intelligence techniques, such as knowledge-base systems and neural networks, for security applications is well-known [33, 34], the possibility of using case-based reasoning in this realm does not seem to have so far been considered. CBR can be a natural fit for this domain, however, because it adheres to the two tenets underlying case-based systems mentioned in the foregoing, namely, the regularity of the world and the recurrence of similar problems.

In order to show the recurrence rate of the security problems, we will use data from CERT [43]. CERT is a well-established repository for incident reports, supported by the US department of defense. In addition to its incidents database, CERT also maintains a list of currently known vulnerabilities. In Figure 3 and Figure 4, the number of incidents and the number of vulnerabilities reported to CERT per year are given respectively.

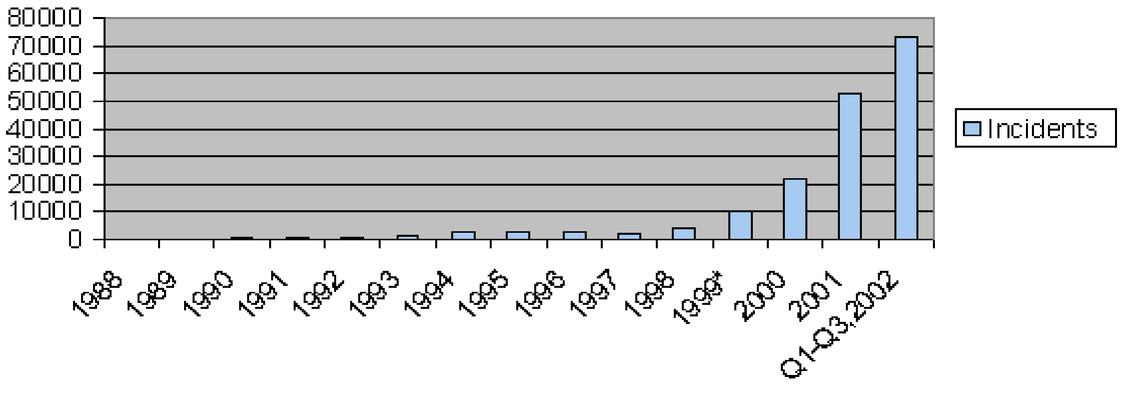


Figure 3: Number of incidents reported to CERT

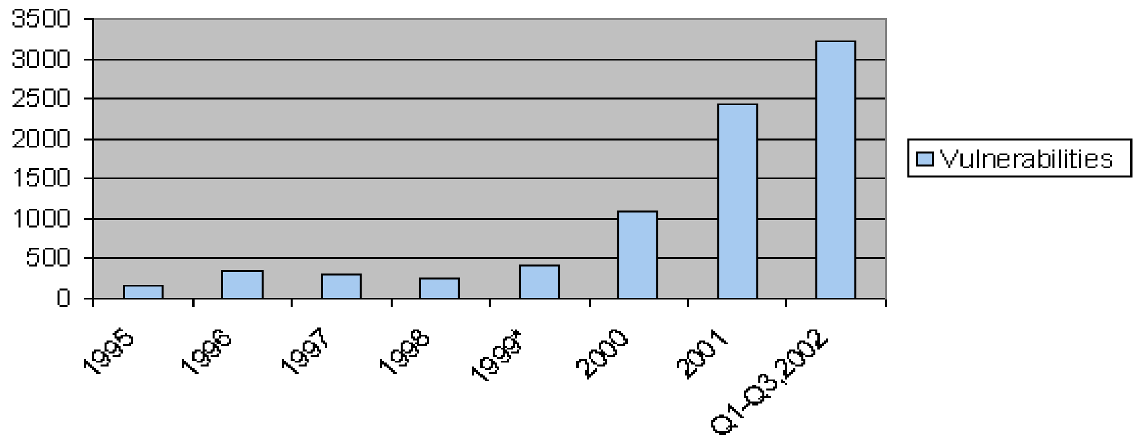


Figure 4: Number of vulnerabilities reported to CERT

From Figures 3 and 4, the average number of recurrences per vulnerability can be calculated by dividing total numbers of incidents reported for a given year to the number of different vulnerabilities reported for the same year. For example for the year 2001, on average, each vulnerability is exploited twenty times.

As noted by Ian Watson in [5] case-based reasoning is a general methodology, as distinguished from, say, rule-based reasoning and neural nets, which are more correctly viewed as technologies. The distinction is based on the observation that case-bases systems can utilize many different such AI technologies in both the case retrieval and case adaptation phases of the CBR process. For example, the retrieval phase might use fuzzy logic as a basis for the similarity metric, and the adaptation phase might use a rule-based expert system. Specific examples of CBR systems that integrate other AI techniques are noted by [24, 25]. This distinction becomes crucial in the design of our adaptive security framework, because it describes the cased-based approach at the appropriate level of abstraction and generalization and allows for the integration of other AI techniques.

2.2 Adaptive Architectures and Meta-Data

2.2.1 Overview of Adaptive Software Architectures

Due to the complexity of today's problem environments and the increasing expectations place on software systems, it is impossible to create a software system design that satisfies all such expectations at all dimensions of the given problem. The time versus space trade off for programmers in the early ages of software engineering has become a multidimensional dilemma including security, portability, performance, quality of service, reusability, and much more. These factors cause the requirements of the software systems to change rapidly in accordance with changes in the circumstances under which the software is used.

There is an observable trend in software design methodologies towards adaptive software systems. In the 1970's, structured programming was popular [20]. This technique enables one to build large applications where the specifications are given before the design phase and they are then implemented as specified. Then, with 1980's, the Object-Oriented Programming concept was introduced. This technique makes it easier for the programmer to handle specification changes. Since all the functionality of the software is separated among different classes, and the interactions between classes are organized in such a way that they minimize coupling and maximize cohesion, the change management of specification becomes easier. However, any change still requires programmer involvement. Even a simple change in requirements still leads to a costly, in terms of both time and money, software engineering cycle that includes all of redesign, re-implementation, and retesting.

Adaptive programming is a new software design methodology that enables the user to change the behavior of the program without changing its code [16, 18]. This requires a different software architecture. Adaptive or reflexive architectures are defined as the architectures that can dynamically adapt at run-time to new user requirements. In this architecture, the environment is modeled in the form of structural descriptions, constraints, and rules, rather than classes as in object-oriented programming [16]. This flexibility requires different levels of abstraction. Even though it is generally difficult to build, and more difficult to understand, these software architectures, the amount of actual code required for the program decreases dramatically.

Adaptive architectures heavily rely on dynamic programming languages. Static languages, like C, require the programmer to establish the structure of the program in terms of data structures and data manipulation in the early stages of programming. However, in dynamic programming languages, a running program can add a new method to one of its classes without recompilation. Currently, Java is one of the most popular programming language providing dynamic programming features.

Figure 5 depicts how an adaptive program differs from a conventional program [16]. In adaptive programs, the class structures are defined partially by setting a number of constraints over the class structure that must be satisfied. Examples of such constraints are method names and the types of the data structures that the methods will act upon.

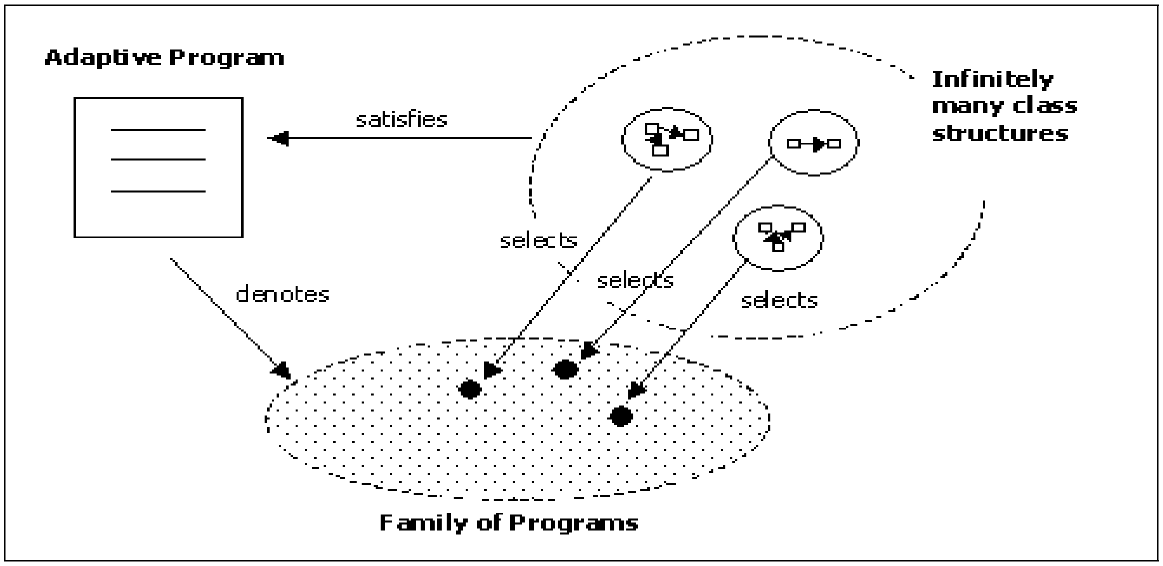


Figure 5: Infinite family of programs denoted by an adaptive program

As a result, there are infinitely many class structures satisfying the constraints defined by a given adaptive program. Further, behaviors of the methods are not implemented thoroughly in the adaptive program. Methods are implemented or bound only whenever they are needed. Since each class structure represents a conventional object-oriented program, an adaptive program denotes a family of programs where their class structures satisfy the constraints of the adaptive program.

2.2.2 Meta-Data

Meta-Data is one of the most common tools used by adaptive architectures to provide easy change management. Data describing the properties of the data and behavior used in an application domain is called meta-data. Once the programmer abstracts out predictable changes or the type of changes in the program and expresses them in terms of meta-data, the maintenance and the change management of the program becomes a task of changing the meta-data not the program code. Thus, it is an easier task. The basic idea lying behind the meta-data concept is that the designer cannot predict all future potential changes and design the system accordingly, but by shifting the information about the domain from the actual source code into data one can make the source code easy to manage and change [17, 18].

2.2.3 Adaptive Software Architecture for Security Framework

In this research, it was aimed to define a common software architecture and methodology that can be used to implement a family of programs in such a way as to enable collaboration between these programs in their decision making processes. In addition, it was aimed to implement the behavior of the well-known Snort [27] intrusion detection system as a proof of concept. In the future, collaborations among different instances of the same security framework will be explored.

Adaptive architectures have played an important role in the design of the security framework proposed in this thesis. The use of this software engineering methodology has made the implementation of the Snort [27] functionality much easier than would have been otherwise possible.

2.3 XML and JAVA

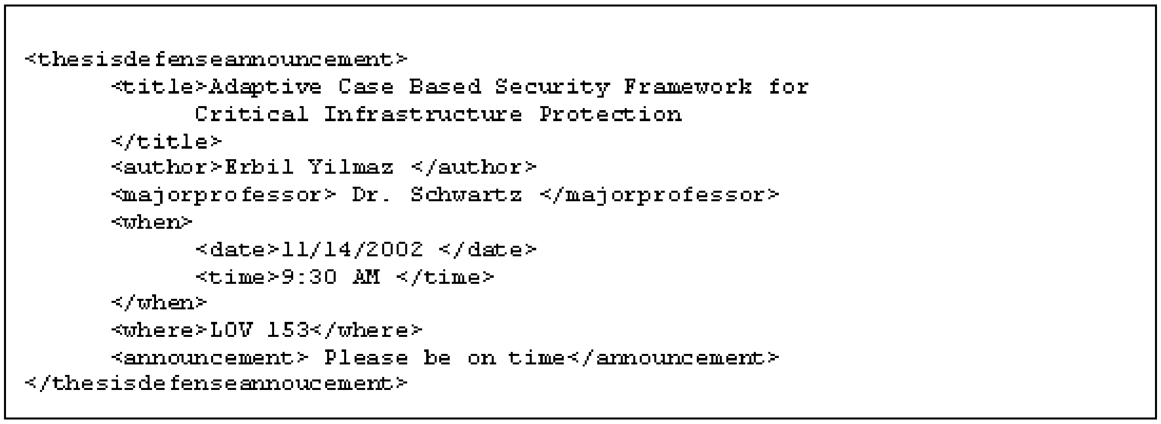
Case representation is one of the most important issues in case-based reasoning systems. In this thesis, we proposed an adaptive case-based framework such that the instances of this framework will solve different problems. Therefore, selecting a generic case representation that can be customized and used for different problem domains becomes critical. Furthermore, the aim of supporting communication among instances of the framework requires a portable and adaptive case representation. With its unique properties described below, XML provides such a portable, customizable data representation.

Java is one of the most widely used object-oriented dynamic programming languages, providing features such as portable, platform-independent code and reflection. Lately, with Java APIs for XML, it has become easy to use XML data representation from Java programs [14, 15]. These two characteristics guided the decision to use Java.

2.3.1 XML

XML (eXtensible Markup Language) is a system-independent data representation language developed by W3C (World Wide Web Consortium) and has become an industry-standard [26, 28]. As with all markup languages, XML uses special notation to differentiate sections of a document. Although XML uses the same notation, angle brackets with tags (<TAG> … </TAG>) to mark different sections, as does HTML, the purpose of these tags in XML is completely different. HTML tags tell the application how to display the text occurring between the start and end tags, whereas XML tags specify the meaning of the text enclosed by the tags. The other main difference between HTML and XML is that HTML tags are predefined. However XML allows you to define your own tags, and, further, the structure of the document, by means of different type definition languages, such as DTD (Data Type Definition) and XML Schema.

The following example, Figure 6, is the XML data representation of a thesis defense announcement structure. The <thesisdefenseannouncement> and <\ thesisdefenseannouncement> tags tells the application that the information between them is a thesis defense announcement. However, the same thesis announcement can be structured differently. In order to avoid confusion a DTD, or some other type definition providing the structure of the announcement, is necessary.



```
<thesisdefenseannouncement>
      <title>Adaptive Case Based Security Framework for
            Critical Infrastructure Protection
      </title>
      <author>Erbil Yilmaz </author>
      <majorprofessor> Dr. Schwartz </majorprofessor>
      <when>
            <date>11/14/2002 </date>
            <time>9:30 AM </time>
      </when>
      <where>LOV 153</where>
      <announcement> Please be on time</announcement>
</thesisdefenseannoucement>
```

Figure 6: Thesis defense announcement XML representation

2.3.2 DTD

A document type definition (DTD) provides the definition of the structure and the content of XML documents [26, 28]. Each declaration in a DTD defines the structure of a building block in the associated XML documents. There are two types of basic declarations: element declarations and attribute declarations. Element declarations specify the set of tags an XML document can contain. Attribute declarations provide additional information about the element defined by a given element declaration. Attribute declarations are not mandatory for each element, as can be seen in Figure 7. This figure defines the structure of the thesis defense announcement XML data given above.

```
<!ELEMENT  thesisdefenseannouncement (title, author,
       majorproffesor, when, where, announcement) >
<!ELEMENT title (#PCDATA) >
<!ELEMENT author (#PCDATA) >
<!ELEMENT majorproffesor (#PCDATA) >
<!ELEMENT when (date, time)>
<!ELEMENT where (#PCDATA)>
<!ELEMENT announcement (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT time (#PCDATA)>
```

Figure 7: Thesis defense anouncement DTD

The first line in the example DTD defines the highest-level element, thesisdefenseannouncement, which must contain all the elements between the parentheses that follow it. The DTD provides more features than the ones covered in this section, such as a capability for selecting among a set of tags. More detailed information about XML and DTDs can be found in [26, 28].

2.3.3 Overview of JAVA APIs for XML

Java provides a set of APIs for XML related applications. These APIs are listed below. They fall into one of two broad categories: document oriented and procedure oriented. [14, 15]

Document-oriented:

- Java API for XML Processing (JAXP): process XML documents using different parsers
- Java Architecture for XML Binding (JAXB): maps XML elements to Java classes

Procedure-oriented:

- Java API for XML Messaging (JAXM): sends SOAP messages over the Internet
- Java API for XML Registries (JAXR): provides a standard way to access business registries

- Java API for XML-based RPC (JAX-RPC): sends SOAP method calls to remote machines and receives results

2.3.4 JAXB

Java Architecture for XML Binding provides a way to create mappings between XML documents and Java objects [14]. The JAXB compiler, takes a given DTD for XML documents and a binding schema, and generates Java classes that contain all the necessary source code to parse and manipulate XML documents satisfying that DTD. Figure 8 explains the binding cycle.

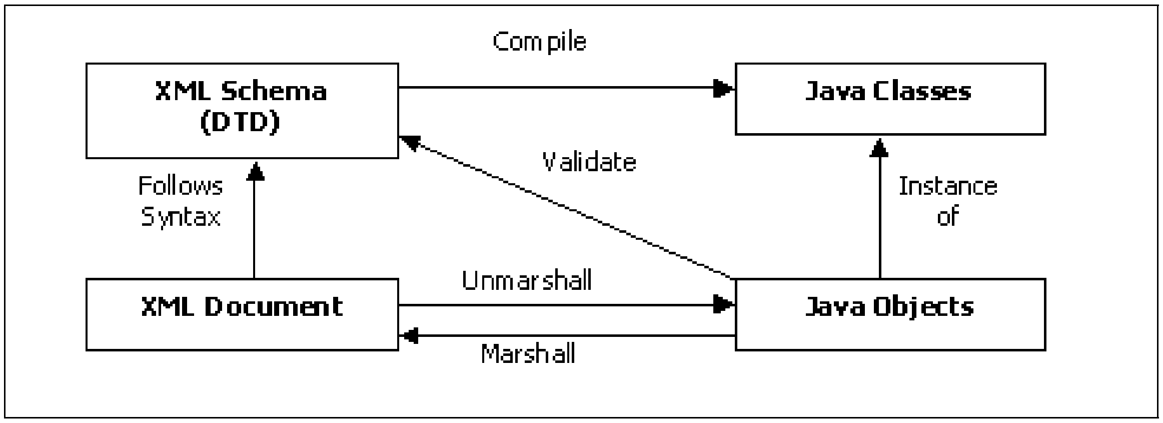


Figure 8: Binding framework cycle

The primary operations of the binding framework as shown in Figure 8 are unmarshalling, marshalling, and validation.

- Unmarshalling: the process that creates a content tree instance from a given XML document.
- Marshalling: the process that creates an XML document from a content tree instance.

- Validation: the process that checks whether a content tree satisfies all the constraints expressed in DTD.

The generation cycle of Java classes from a DTD and a binding schema is given in Figure 9. The first step is to write a binding schema. This contains instructions on how to bind the DTD components to Java classes. For example, for the DTD shown in Figure 7, the binding schema might specify that a class be created for the element `thesisdefenseannouncement` with variables representing `title`, `author`, etc., and that `#PCDATA` indicates the data type `String`. Then one runs the schema compiler (xjc) with the DTD and binding schema using the command:

```
xjc datatypedefinition.dtd bindingschema.xjs
```

Execution of the this command generates Java class source code that has all the functionality needed to handle the primary operations of the binding framework (marshalling, unmarshalling, and validation) over XML documents whose structure are defined by given DTD.

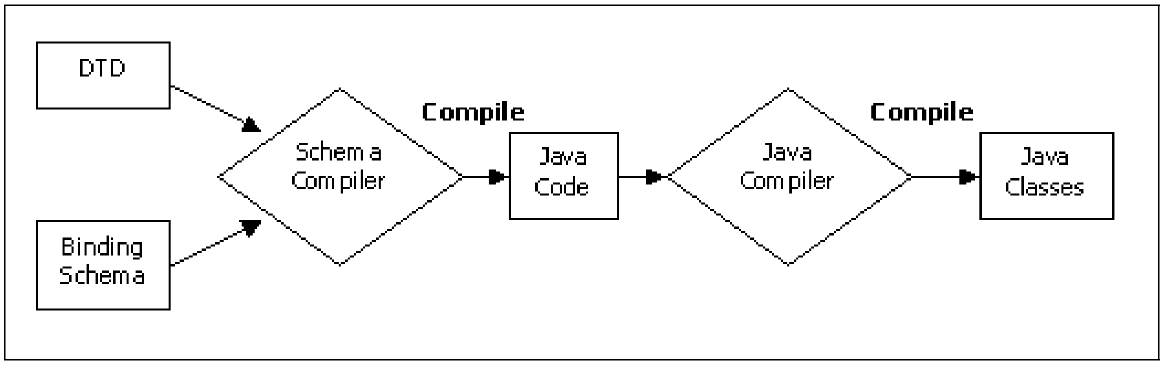


Figure 9: Generation of Java classes from DTD and binding schema

2.4 Intrusion Detection

Intrusion detection [1, 2] is the process of monitoring computers on hosts or computer networks for violations of security policy. The first model for an intrusion detection system (IDS) was proposed in 1986 by Denning [3]. Although many different intrusion detection systems have emerged since Denning's paper, the basic structure for the intrusion detection systems is still very similar to Denning's model. An IDS consists of three components:

- Information source providing a stream of event records
- Analysis engine that processes the evidence to find intrusions
- Response component that generates actions based on the analysis

Intrusion detection systems fall into three categories: network-base, host-based, and, recently, kernel-based intrusion detection systems [41].

2.4.1 Network-based IDS

A network based intrusion detection system (NIDS) monitors network traffic for suspicious activity. The idea of a NIDS evolved from packet sniffers and network monitors that capture all the packets on the network and produce a view of the network traffic. NIDS are automated systems that look for particular signs in the network traffic. Once such activity is detected, they take pre-defined actions such as raising an alert message to the system administrator's screen or reconfiguring the firewall to block unwanted packets. Some examples of such systems are NIDES [44], Snort [13], network flight recorder [31], and RealSecure [32].

2.4.2 Host-based IDS

As opposed to network-based IDS systems, host-based systems monitor events on a particular computer for signs of intrusion. The information sources for host-based systems consist of operating system audit trails and system logs [1, 2]. These sources give the IDS information about login activity, root activity, file system activity, and much more. Some examples of such systems are NADIR [33], MIDAS [34] and TripWire [35].

2.4.3 Kernel-based IDS

Kernel based intrusion detection is a relatively new intrusion detection concept. Some examples of such systems are OpenWall [36] and LIDS [37]. These systems work at the kernel level of the operating system and aim at preventing buffer overflows, increasing file system protection, blocking signals among processes, and generally making intrusion more difficult.

2.4.4 Conclusions on IDS

Since new threats and security holes are being discovered every day, it is almost impossible to prevent all intrusions. However, by keeping up-to-date versions of security tools and by using more than one tool for each type of intrusion detection, one can reduce the risk of intrusion.

CHAPTER 3

RELATED WORK

3.1 Previous XML Based CBR Applications

How to best represent cases in a case archive is one of the open problems in case-based reasoning research. The following works on case representation in XML have been influential in determining the approach to case representation employed by the proposed framework.

3.1.1 CARET/XML

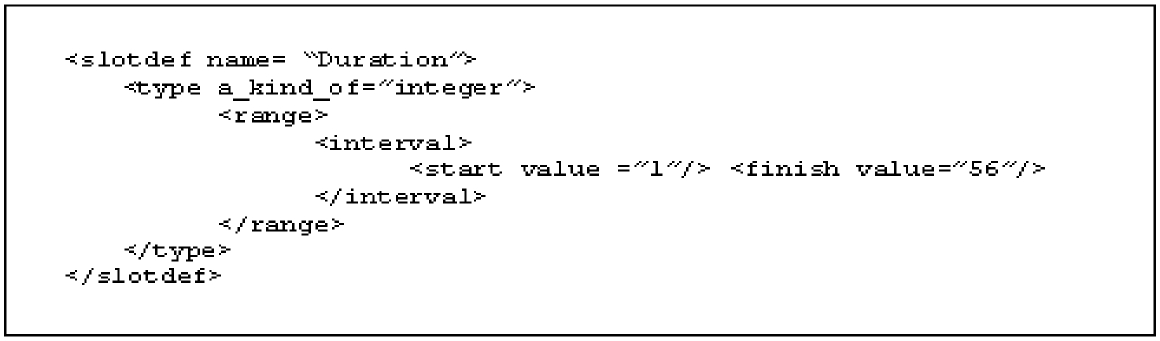
CARET/XML, designed by Shimazu [6], is textual case-based reasoning tool that uses XML as its case representation language. It is a client-server architecture based on the WWW, where a client inputs his/her queries via a web browser, and the CARET/XML server responds with the URL of the similar case profiles. Then a java applet on the client side, with case adapter, retrieves case profiles with HTTP calls to the server. For measuring the similarity between a given case and those in the archive, this system uses what is called a similarity data definition (SDD) file, which, for each possible case feature, contains a tree structure from which one can calculate the degree of match between any two given values for that feature. This tree structured similarity value assignment is adopted from an earlier system, SQUAD, by the same author.

This system is of interest in that it employs a web-based client-server architecture and an XML-based case representation. It is restrictive, however, in that one must also provide a complete SDD file constituting a predefined similarity assessment for all possible case features. Hence it does not provide the required level of flexibility for our security framework.

3.1.2 CBML

Case-Based Markup Language (CBML) has been proposed as the case representation language for a distributed CBR system [7, 8, 9]. The design of CBML was based on the CASUEL case representation language [26] and it aimed to provide functionality similar to CASUEL. Although CASUEL is an object-oriented language, CBML is designed for flat feature-value representations.

CBML consists of two files, namely a DTD file and an XML file. The DTD file describes the structure of the case, whereas the XML file contains the cases. The main characteristic feature of CBML is that it has <slotdef> tags for each case feature that define its type and possible values. The following Figure 10 gives a sample <slotdef> tag definition for the Duration feature.



```
<slotdef name= "Duration">
    <type a_kind_of="integer">
        <range>
            <interval>
                <start value ="1"/> <finish value="56"/>
            </interval>
        </range>
    </type>
</slotdef>
```

Figure 10: Sample <slotdef> tag definition.

Although the proposed <slotdef> tag structure is effective for simple data type comparisons, a more general feature comparison method, that is able to do computations for different comparison algorithms, is necessary for complex data types.

3.2 Snort Intrusion Detection System

3.2.1 Overview

Snort is a lightweight intrusion detection system developed by Marty Roesch [27]. As opposed to most of the commercial network intrusion detection systems, which require a dedicated platform and user training, Snort has a footprint of only a few kilobytes, and is easy to install, configure, and use. It is freely available, quite popular, and runs on Linux, BSD, Solaris/SunOS, HP-UX, AIX, IRIX, MAC OS X, and Windows.

Snort is a real-time packet analyzer and logger on IP networks that is used as a Network Intrusion Detection System (NIDS) [27]. It monitors the network traffic in a per packet manner for predefined suspicious patterns or activities. These suspicious patterns are expressed in terms of Snort Rules.

Snort's architecture is comprised of three subsystems: the packet decoder, the detection engine, and the logging and alerting system. The packet decoder subroutines are called in the order of the protocol stack for the TCP/IP reference model, i.e. from the data-link layer through the transport layer, and then the application layer. These subroutines assume that the raw network data satisfies the specific data structure imposed by the protocol stacks. Since performance is the key issue in the Snort packet decoder subsystem, instead of making copies of all the packets, the packer decoder only sets pointers to the packet data fields that are used by detection engine.

The detection engine is the heart of the Snort intrusion detection system. This searches through its detection rules for a match with the decoded packet data in a per-packet manner. The creation of the data structure representation of the Snort rules in memory from the rule files takes place before the packet sniffing begins. Therefore, rule parsing does not produce a bottleneck in performance. The first matching rule with decoded packet triggers the action part in the rule definition. It is possible that the packets may move through the network faster than the Snort engine can handle them, in which case some packets will not be processed. In most current systems, however, Snort is still fast enough to keep up. This is the reason it has retained its popularity.

3.2.2 Snort Rules

Snort uses a simple rule description language that is flexible and quite powerful [13]. The rule structure is divided into two sections, the rule header and the rule options. The rule header consists of rule action, protocol name, source IP address, source port number, destination IP address, and destination port number respectively. These parts are defined for each Snort rule. However, a Snort rule may or may not contain rule options. Some rule options are content, by which one specifies search pattern in packet's payload, and dsize, by which one specifies a limit on the packet's payload data size. In order for a Snort rule to be fired, all the header features and option features must match their corresponding fields in the network packet.

CHAPTER 4

THE ADAPTIVE CASE BASED SECURITY FRAMEWORK

In previous chapters, we introduced technologies and methodologies used in the adaptive case-based security framework. Namely, these were (i) the adaptive architectures used both for the software design and the general framework implementation, (ii) the case-based reasoning methodology for analysis of data, (iii) the use of XML and DTDs for the knowledge representation structure, and (iv) the use of Java with its APIs supporting XML for actual implementation of the framework. In this chapter, we will explain how all these parts fit together to build the backbone of the security framework.

The idea behind the security framework is to create an adaptive software for security applications that uses case-based reasoning as its primary methodology for problem solving and that can be customized into a specific software instance according to the requirements of the problem domain. It is intended that each instance of the framework will solve a different security problem using the methodology defined in the framework, as illustrated in Figure 11. Instances of the framework are created through a process known as customization, with each instance being to create a case-based reasoner for a specific problem domain. The instances differ in terms of both the types of the cases and the similarity metrics used in the case retrieval process. The details of this customization process will be explained in the sections that follow.

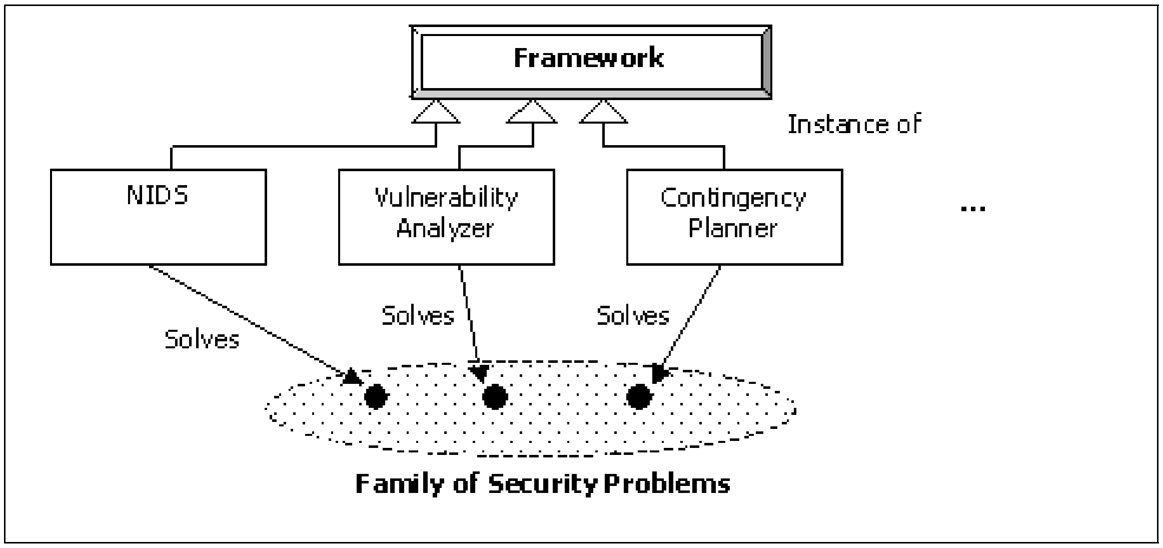


Figure 11: Instances of the security framework

4.1 Systems Overview

A schematic overview of the security framework is shown in Figure 12. This framework contains only the generic notion of case-based reasoning process and the mechanisms to adapt some given domain specific knowledge so as to create an instance of the framework. However, extracting the generic notion of CBR requires separating the characteristics of CBR that are common to all case-based reasoners from the characteristics of CBR that are specific to the given problem domain. Two major common characteristics of CBR are the case representation scheme and the mechanisms to assess similarity among cases. The details of the generic case-based reasoning system are explained in the next section.

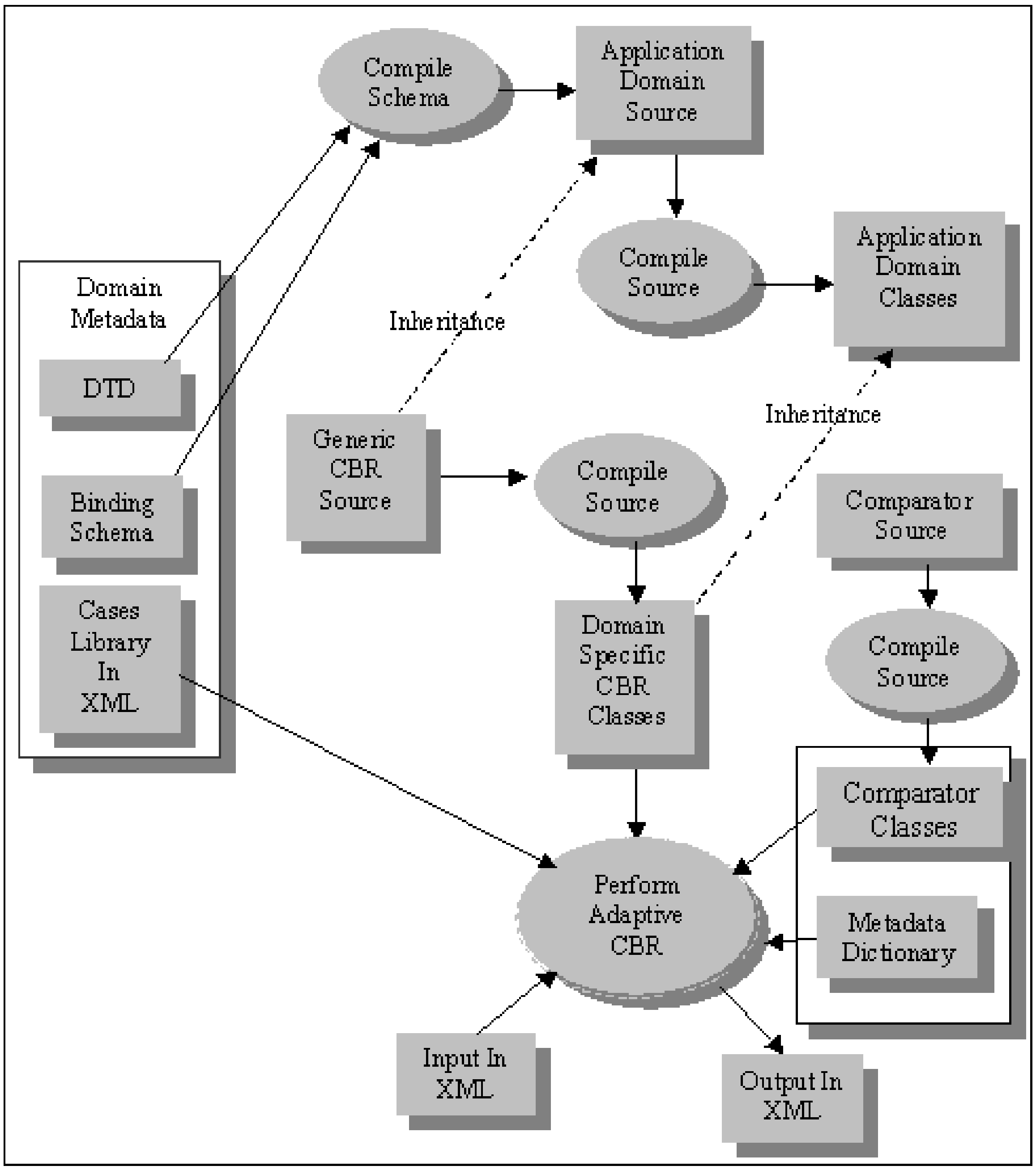


Figure 12: Adaptive Case Based Security Framework Overview

Assuming a generic case representation structure, customization of the framework for a given problem domain begins with defining a domain-specific case representation DTD that satisfies the general constraints required by the generic structure. With this defined DTD and a proper JAXB binding schema for this DTD, the adaptation mechanism in the framework creates classes to instantiate and to process XML data that conforms to the DTD. The adaptation mechanism then uses JAXB to create the desired classes to support the domain specific features.

This completes the first step of the customization process. The framework, at this point, has evolved to have capabilities to process domain specific XML data, but it is still not equipped with enough knowledge to analyze the data. Without a case library, a CBR system cannot solve any given problem. Therefore a case library has to be provided for given problem domain. The case library is easily loaded using the customized DTD and JAXB created classes.

With only a generic notion of CBR, the current state of the framework does not know how to assess the similarity of cases. Moreover, in order to perform the CBR process, the framework needs to know how to compare cases. This is accomplished by applying comparator methods to the individual case features. The determination of which comparator to apply to which feature is represented in the case meta-data dictionary. The comparator to use on a feature is determined dynamically during the CBR process by looking this information up in a metadata dictionary. If the current version of the framework does not contain all the needed comparators for a new problem domain, then new comparators are written and included in the comparator collection, and appropriate new entries for these need to be added to the metadata dictionary. Details of these components appear in the following sections.

After completing the customization process described above, the evolved system is ready to serve in given problem domain. For a given problem instance in this domain, the new system applies the CBR process to find the best solution for that problem among those that are currently known.

4.2 Generic Case Based Reasoning

For the process of separating the generic aspects of CBR from domain specific ones as well as for the representation of cases in XML format, works such as CARET [6] and CBML [8] have been influential. In CBML, a case-based markup language defined that is applicable to this research. However, since it is not an accepted standard for case representation, we avoided restraining the framework to a specific language.

As mentioned, there are two major characteristics of CBR systems that differentiate case-based reasoners from each other, namely case representation and similarity assessment. Therefore a generic CBR must have the minimal knowledge about these characteristics so that it can adapt to new problem domains through their customization.

4.2.1 Generic Case Representation

Figure 13 gives the case representation that the generic CBR assumes. Specifically, it is assumed that a case is composed of a list of case features defining the problem and a solution, and a case feature is composed of a list of feature values. Case features are categorized into required features and optional features to emphasize the fact that some features may not appear in all cases. These requirements comprise the general constraints required for all applications of the generic CBR framework. They are quite flexible and are observed as common among a wide variety of CBR systems.

```
<!ELEMENT casevector (case+) >
<!ELEMENT case (caseid, requiredfeature+, optionalfeature*,
solution)>

      <!ELEMENT caseid (#PCDATA)>

      <!ELEMENT requiredfeature (featurevalue+)>
            <!ATTLIST requiredfeature
                  featurename  #REQUIRED
                  negation (NOT) #IMPLIED>

      <!ELEMENT optionalfeature (featurevalue+)>
            <!ATTLIST optionalfeature
                  featurename  #REQUIRED
                  negation (NOT) #IMPLIED>

            <!ELEMENT featurevalue (#PCDATA)>
            <!ATTLIST featurevalue
                  negation (NOT) #IMPLIED>

      <!ELEMENT solution (actiontype, message?)>

            <!ELEMENT actiontype (#PCDATA)>
            <!ELEMENT message (#PCDATA)>
```

Figure 13: Generic Case Representation Structure assumed by the framework

4.2.2 Similarity Assessment

As explained in section 2.2, the first step in the cased-based reasoning process is to retrieve cases similar to the given one from the case library. This step requires a similarity assessment mechanism to select the most similar cases. A nearest neighbor algorithm is widely in used for this purpose. This algorithm selects the case with the highest similarity value, where this is computed as weighted sum of the similarity values for the individual features. The feature comparison requires comparators that can assign a similarity value for the features. In this thesis, any method that takes two case feature values and produces a similarity value for them is called a comparator instance, inasmuch as it is an instance of a generic comparator class (in Java, and object of type Comparator). Such an instance could be a human fingerprint comparator that evaluates the similarity between two given fingerprints, where the measure of similarity is determined by a neural network, an AI technique commonly used in fingerprint matching and face recognition. Thus, the present framework is quite flexible, and allows for the use of virtually any kind of comparator one may desire.

4.3 Domain Meta Data

Domain meta-data, shown in the box on the left of Figure 12, is the knowledge about the domain used in customizing the framework. It is composed of three components, a DTD, a binding schema, and a case library. In the DTD, the structure of the case representation and the features types that identify a case are defined. The DTD for a given domain must satisfy the general constraints specified in the generic case-based reasoner, as described in the foregoing section 4.2.

The binding schema is automatically created by the framework from the given DTD. This schema explains to the Java Schema Compiler (part of the JAXB package) how to create the Java classes needed to parse and represent XML documents that conform to the given DTD. In effect, the binding schema contains instructions regarding how to bind DTDs to a Java class.

The last component of a domain meta-data is a case library. This is a collection of previously solved problems with their definitions and solutions encoded in XML according to the DTD. Once all of the above domain meta-data is given, the framework is ready to create Java classes to represent and manipulate XML data for the given domain.

4.4 Generic and Domain Specific CBR Modules

The adaptive case-based security framework, or generic CBR, module assumes no knowledge about the problem domain. It has to go through a customization process to evolve into an instance of the framework comprising a domain-specific CBR module. It is crucial to understand the difference between generic and domain-specific modules. One can think of generic module as an adaptive program and a domain-specific module as a conventional program.

As explained in section 2.3.1, an adaptive program does not contain fully defined behaviors for the methods [16]. The methods are implemented, or bound, only when they are needed, i.e., dynamically during the execution of the program. Thus, in our generic CBR module, the methods necessary for a CBR process are defined but not totally implemented for any particular domain. The customization process makes the generic module a domain-specific module.

In summary, the customization process involves three steps: generation of Java classes to handle domain specific knowledge, definition of the necessary comparators to compare case features for a given domain, and specification of which case feature uses which comparator in terms of a meta-data dictionary. Throughout this process, the source code for the generic module is not changed. The only programming required for customization is to create new comparators as might be necessary for the new domain. It should happen that accumulation of comparators over time would make writing code for comparators less frequent or even unnecessary.

4.5 Comparators

For each new type of case feature, the framework needs a comparator that evaluates whether, or to what degree, a given problem definition feature matches with the corresponding case feature in the case library. As an example, a comparator for IP addresses can give a degree of similarity for two given IP addresses according to whether they are the same, or on the same machine, or on the same network, or some other measure as may be desired.

The comparators comprise a collection of classes that are kept and compiled separately. Whenever a new comparator is defined, the code is compiled and the comparator class is stored so that it is ready to use if a domain-specific module needs it for a case feature comparison. When a comparator is needed, it is created dynamically during run time as an instance of the associated comparator class.

4.6 Meta Data Dictionary

As mentioned, a meta-data dictionary is used to associate features with their comparators during run time. This enables the domain-specific module to create an instance of the required comparator class as described above. This use of meta-data dictionaries is a well-known practice of adaptive, or reflective, software programming, cf. [16, 17, 18]. This requires a dynamic object-oriented programming language such as Java that supports run-time reflection. Run-time reflection provides a mechanism by which the application can dynamically determine the types of the objects and create the methods to apply based on the object's type.

To illustrate, a meta-data dictionary entry for an IP case feature is as follows:

DataElementName: SourceIP
DataElementType: int
ComparatorType: IPRangeComparator

In this example, the IP address is assumed to be stored in an integer format that corresponds to a 32-bit binary representation of an IP address. The entry also says to use an instance of the IP range comparator class to compare two IP addresses.

CHAPTER 5

IMPLEMENTATION OF SNORT WITH ADAPTIVE CASE-BASED SECURITY FRAMEWORK

A network intrusion detection system is an essential part of every organization's security framework. Among many others, Snort has recently become very popular, and is considered very successful [27]. There are two main reasons for its popularity: (i) as many other open-source systems, it can be used free of charge, and (ii) the rules written for Snort are being shared among its users, resulting in an accumulation of rules. In this chapter, as a proof of concept, customization of the proposed framework to replicate the functionality of Snort will be explained.

5.1 Domain Meta Data and Case Representation

Implementation of Snort in the adaptive case-based security framework requires transformation of Snort knowledge into a case library. In the Snort IDS, knowledge is expressed in terms of rules. Therefore, a logical approach for transformation will be to convert each Snort rule into a case. An example of a Snort rule is given in Figure 14.

As explained in section 3.2.2, Snort rules are structured by a well-defined syntax. Each rule has an action, protocol name, source IP address, source port number, destination IP

```
alert tcp any any -> 128.186.122.0/24 111 (content:"|00 01
      86 a5|"; msg: "mountd access";)
```

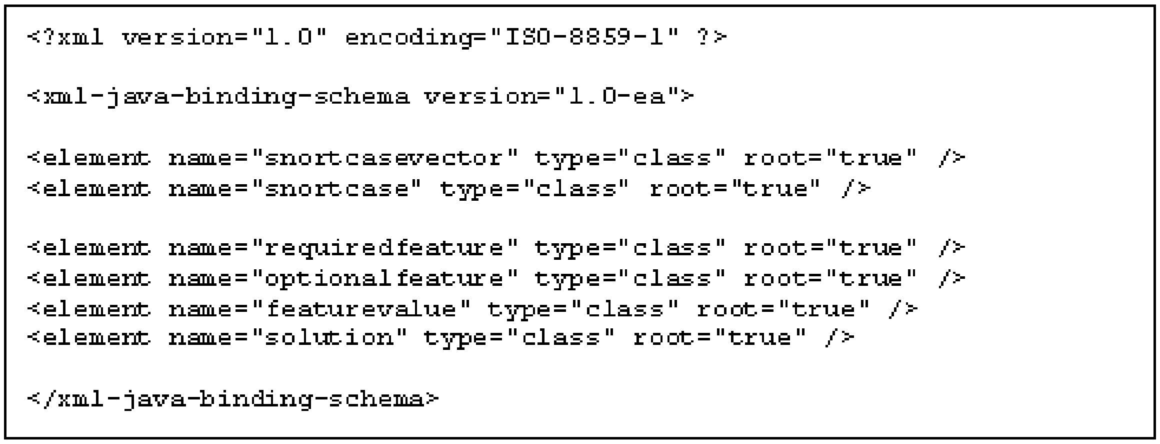
Figure 14: Sample Snort rule

address, and destination port number. These features may or may not be followed by a set of optional features in parentheses. The data type definition for case representation for the Snort IDS implementation, which satisfies the generic case-representation constraints introduced in section 4.2.1, is given in Figure 15.

```
<!ELEMENT snortcasevector (snortcase+) >
<!ELEMENT snortcase (caseid, requiredfeature+, optionalfeature*,
solution)>

      <!ELEMENT caseid (#PCDATA)>

      <!ELEMENT requiredfeature (featurevalue+)>
            <!ATTLIST requiredfeature
                  featurename ( protocol | sourceip | sourceport |
destinationip |destinationport ) #REQUIRED
                  negation (NOT) #IMPLIED>

      <!ELEMENT optionalfeature (featurevalue+)>
            <!ATTLIST optionalfeature
                  featurename ( ttl | tos | flags | id | dsize |
message | sequencenumber | acknowledgement | content | offset )
#REQUIRED
                  negation (NOT) #IMPLIED>

            <!ELEMENT featurevalue (#PCDATA)>
            <!ATTLIST featurevalue
                  negation (NOT) #IMPLIED>

      <!ELEMENT solution (actiontype, message?)>

            <!ELEMENT actiontype (#PCDATA)>
            <!ELEMENT message (#PCDATA)>
```

Figure 15: Data Type Definition for Snort case representation

Once the DTD for case representation is determined, the binding schema is generated automatically. Figure 16 shows the binding schema generated for the DTD given in Figure 15.



```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<xml-java-binding-schema version="1.0-ea">

<element name="snortcasevector" type="class" root="true" />
<element name="snortcase" type="class" root="true" />

<element name="requiredfeature" type="class" root="true" />
<element name="optionalfeature" type="class" root="true" />
<element name="featurevalue" type="class" root="true" />
<element name="solution" type="class" root="true" />

</xml-java-binding-schema>
```

Figure 16: Binding schema for Snort DTD

In the process of automated binding schema generation, an element binding declaration is defined for each non-simple element in the corresponding DTD, where “non-simple” means that there exists at least one other element contained in it. This schema is then compiled with the schema compiler, namely xjc [14]. This generates Java source code for each non-simple element to represent and manipulate the data in that element of any XML document conforming to the DTD. This source code then needs to be compiled with Java compiler, javac, to obtain Java classes. The generic CBR module uses instances of these generated Java classes in its CBR process to access the XML case library.

At this stage of the transformation, the framework is ready to process XML case library knowledge. However, the Snort knowledge is still in Snort rules format. Since the representation of a Snort rule in XML format has been determined and expressed with a DTD, the rest of the transformation is a mechanical process. A Java program was written to convert Snort rules into XML cases for the case archive. The XML representation for the sample Snort rule in Figure 14 is shown in Figure 17.

```
<snortcase>
      <caseid>1</caseid>
      <requiredfeature featurename = "protocol">
            <featurevalue>tcp</featurevalue>
      </requiredfeature>
      <requiredfeature featurename = "sourceip">
            <featurevalue>any</featurevalue>
      </requiredfeature>
      <requiredfeature featurename = "sourceport">
            <featurevalue>any</featurevalue>
      </requiredfeature>
      <requiredfeature featurename = "destinationip">
            <featurevalue>128.186.122.0/24</featurevalue>
      </requiredfeature>
      <requiredfeature featurename = "destinationport">
            <featurevalue>111</featurevalue>
      </requiredfeature>
      <optionalfeature featurename = "content">
            <featurevalue>|00 01 86 a5|</featurevalue>
      </optionalfeature>
      <solution>
            <actiontype>alert</actiontype>
            <message>"mountd access"</message>
      </solution>
</snortcase>
```

Figure 17: XML case representation for sample Snort rule

5.2 Comparators

For each new type of case feature that has not been encountered previously by the framework, a comparator has to be defined that determines to what degree a given problem definition feature is similar to the corresponding feature of a case in the case library. However, in Snort, the comparison results are either matched or unmatched, i.e. the degree of similarity is defined in a discrete manner. Snort implementation requires only four different comparators: exact-comparator, range-comparator, IP-range-comparator, and string-pattern-comparator.

5.2.1 Exact Comparator

The exact-comparator returns true if the features given are exactly the same. For example, the protocol feature requires an exact string comparison, which returns true if the protocol feature in the problem description has the exact same value as the protocol feature in compared case; otherwise it returns false. Since the type information is also stored with each feature in data dictionary that is retrieved at run-time, the exact-comparator can be also used to compare integer values by checking the type of the feature and applying proper comparison.

5.2.2 Range Comparator

In Snort, features such as port number and data size require a range comparison [27]. For example, the value "111:" for port numbers matches with any port number greater than or equal to 111. The range-comparator understands the Snort syntax and returns true if the port number of the "problem" network packet falls within the range specified by given numeric value.

5.2.3 IP Range Comparator

Because of their special string representation, IP numbers are compared with a special IP-range-comparator. IP version 4 addresses are expressed in XXX.XXX.XXX.XXX format where XXX is a value between 0 and 255. In Snort, suspicious IP values are given either as an IP address in the rule or a range of IP addresses that is in the form of IP1:IP2, where IP1 and IP2 are IP addresses. IP-range-comparator gets the value of the case feature from the case library. If it is a range value, it assigns its internal min and max variables to the IP1 and IP2 respectively. Otherwise, in case of a single value, it assigns both min and max values to the same value. Then it applies a range comparison and returns the result of the comparison.

5.2.4 String Pattern Comparator

In Snort, the optional data for packet content is either in binary format, represented as hexadecimal numbers enclosed by a pair of "|" characters, or in text format []. The comparison for the content option does not require a perfect matching. Instead a pattern match is performed. Hence a string-pattern-comparator is implemented for content feature value comparison. String-pattern-comparator differs from exact comparator in that it searches for the occurrence of the pattern given in case feature in the packet payload.

5.3 Meta Data Dictionary

As the final part of customization, in order to be able to instantiate necessary comparators, the evolving framework needs to know the comparator type for each case feature. In Table 1, the complete meta-data dictionary for our Snort implementation is given. Although fourteen case features are defined for the Snort implementation, only four different comparators have to be implemented. This is because several different features use the same comparator.

As described in the section 4.1, feature comparators are compiled and collected separately. With the accumulation of comparators over time, the likelihood that one will need to implement a new comparator for a new domain will decrease.

Table 1: Meta Data Dictionary for Snort Implementation

Data Element Name	**Data Element Type**	**Comparator Type**
Protocol	String	Exact Comparator
Source IP	Integer	IP Range Comparator
Source Port	Integer	Range Comparator
Destination IP	Integer	IP Range Comparator
Destination Port	Integer	Range Comparator
TTL	Integer	Exact Comparator
TOS	Integer	Exact Comparator
Flags	String	Exact Comparator
ID	Integer	Exact Comparator
Dsize	Integer	Range Comparator
Sequence Number	Integer	Exact Comparator
Acknowledgement	Integer	Exact Comparator
Content	String	String Pattern Comparator
Offset	Integer	Exact Comparator

5.4 How Snort Customization of the Framework Works

The customization process described above leads to an instance of the framework for the network intrusion detection problem that has the knowledge of Snort in its case library. Given these components---namely the DTD, the JAXB created Java classes to handle XML data, the case library in XML, the comparators, and the meta-data dictionary---the Snort implementation of the framework performs as shown in the lower part of Figure 18, denoted with dotted lines. Simply, when a packet is received from the network, it triggers the CBR module to search through case library for cases that are similar to the network packet.

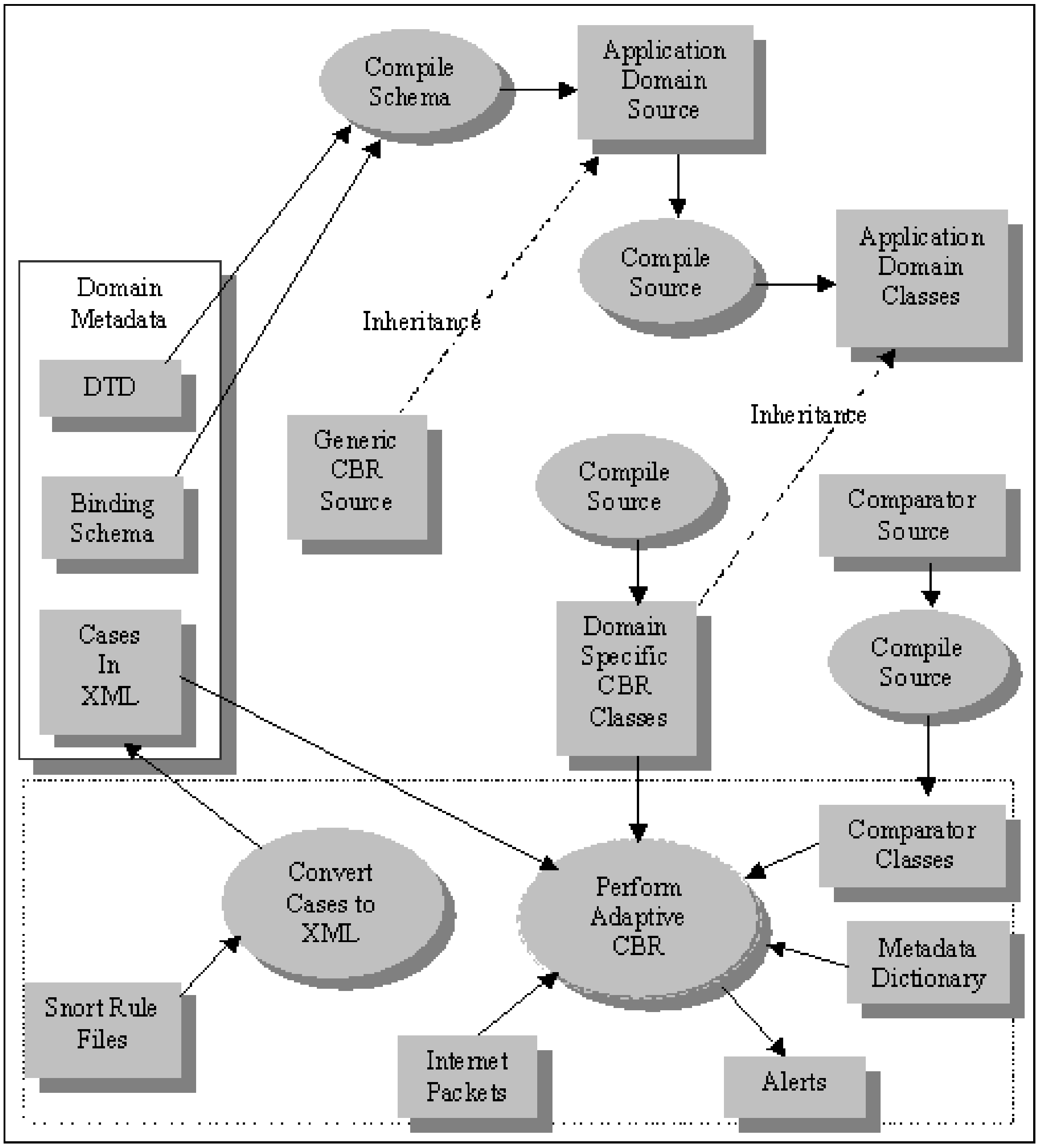


Figure 18: Customization of the security framework for Network Intrusion Detection

The search process examines each case in the case library. For each case, every case feature defined by the DTD is compared with its corresponding packet feature. In order to perform this comparison, first the record containing the name of the case feature is retrieved from the meta-data dictionary. This record contains the name of the comparator that is needed for that particular case feature. Then using reflection on the name of the comparator, an instance of that comparator is created, and both the packet feature value and the case feature value are passed to it. Then the comparator instance determines the similarity value for that feature.

In Snort, since all features in a case need to match with their corresponding packet features in order to conclude that the case and network packet are matched, only those cases that have a complete match on all features are retrieved. In this manner, the CBR module retrieves all cases that match the given network packet. The CBR module proceeds with performing the case actions of these matched cases. In Snort, action is generally reporting an alert message.

5.5 Testing of the Snort Customization

We tested the customization of the framework for Snort by creating six different test modules to test different parts of the implementation. In order to achieve a thorough test of the entire implementation, a test module for each level of comparison was completed.

5.5.1 Case Feature Level Comparator Testing

In the customized framework for Snort, four different comparators are required as explained in previous sections. Each comparator takes a pair of case features and comparison is exercised at case feature level. Therefore, four different test modules were implemented to test each comparator, namely for exact comparator, range comparator, IP range comparator, and string pattern comparator respectively. They were tested with different values of the case features to make sure they give the proper results. They produced the expected results.

5.5.2 Case Level Testing

The fifth test module compares two given cases: one is network packet as a case and the other is a case from the case library. First, the case library with seventy-three cases is created from a subset of current Snort rules. Then, fifteen different synthetic network packets were created in XML format. These network packets were compared with fifteen different cases from the case library for the case level testing and the ones that were expected to match with their tested cases were actually matched.

5.5.3 Case-Based Reasoner Level Testing

Finally, the sixth testing module was implemented to test a given packet against the case library with seventy-three cases generated for the fifth test. This module takes a network packet in XML format and a case library and gives the similar cases, in the case library, to the given packet. Each of the fifteen packets used in the previous test was passed as a parameter to this testing module iteratively along with the case library. The cases matched in the previous test were again matched. Additionally for some packets more cases were matched from the case library that were not tested previously. All the results were followed the Snort rules as expected.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In the present work, we have proposed an adaptive security framework that can be customized according to the needs of any given CBR problem domain. The adaptation techniques used in this research allowed the investigation of the implementation of case-based reasoning and security applications from a different point of view. Instead of producing software for a new problem domain, such as vulnerability analysis, an instance of a more generic framework, created in this thesis, is enough to solve this new problem. Furthermore, instantiation requires minimal coding effort as explained.

We implemented a network intrusion detection instance of the framework that obtained Snort IDS knowledge through its customization phase. The resulting system is not a real-time application that requires network packets in a specific XML format. However the same level of success with Snort is achieved in matching.

The framework is created in such a way that both its software and its knowledge in terms of XML cases are portable. This portability makes the framework very suitable for distributed systems and agent architectures. As a future work, we will build a network of a set of instances of this framework, so that these instances exchange knowledge, in terms of both data and functionality and help each other to increase their performance.

Also, this near term future effort will focus on building higher degrees of intelligence through communication among instances of the framework.

Another potential application of the framework is a distributed architecture where instances of the framework will send their results to a central unit. Then, the central unit will put all the results together and produce an output that cannot be achieved by any single instance of the framework.

In addition, we plan to explore the possibility of applying the methods to CBR problems in the other CIP domains discussed in section 1.1. As mentioned, the aim of creating a case-based reasoning applicable across multiple domains was the primary motivation for this work.

The present work represents a first, and major, step towards this goal. Further work needs to be done, however, to fill out the remainder of the CBR paradigm and produce a fully functioning tool. These tasks include, in particular, creating a module for managing the case adaptation process depicted in Figure 2. This in itself could comprise a major undertaking, possibly entailing application of expert systems technology. Last we can mention the possibility of incorporating an expert system into the case retrieval process as well. This would amount to adopting an interactive, or “conversational,” case-based reasoner along the lines of [30].

BIBLIOGRAPHY

[1] Northcutt, S., J. Novak, and D. McLachlan. *Network Intrusion Detection: An Analyst's Handbook*. New Riders Publishing, 2000.

[2] Bace, R. G. *Intrusion Detection*. Macmillan Technical Publishing, 2000.

[3] Denning, D. E. "An Intrusion-Detection Model." *IEEE Transactions on Software Engineering 13*, no. 2: 222-232, 1987.

[4] Leake, D. B. *Case-Based Reasoning: Experiences, Lessons and Future Directions*. AAAI Press/ The MIT Press, 1996.

[5] Watson, I. "CBR is a methodology not a technology." *The Knowledge Based Systems Journal 12*, no.5-6: 303-8, Elsevier, 1999.

[6] Shimazu, H. "A textual cased-based reasoning system using XML on the world-wide web." *Advances in Case-Based Reasoning,* Proceedings of 4th European Workshop, EWCBR-98, Lecture Notes in Computer Science, LNAI v 1488: 274-285, Springer Verlag, 1998.

[7] Doyle, M., M. Ferrario, C. Hayes, P. Cunningham and B. Smyth. "CBR Net: Smart technology over a network." Technical Report, TCD-CS-1998-07, Department of Computer Science, Trinity College Dublin, 1998.

[8] Hayes, C. and P. Cunningham. "Shaping a CBR view with XML." *Case-Based Reasoning Research and Development*, Proceeding of the Third International Conference on Case-Based Reasoning, ICCBR-99, Lecture Notes in Computer Science, LNAI v 1650: 468-, Springer Verlag, 1999.

[9] Hayes, C., P. Cunningham and M. Doyle. "Distributed CBR using XML." Technical Report, TCD-CS-1998-06, Department of Computer Science, Trinity College Dublin, 1998.

[10] Davis, M. T. "Homeland Security: new mission of a new century." Northrop Grumman's Analysis Center Papers, 2002.

[11] Wulf, W. A. and A. K. Jones. "Cybersecurity." *The Bridge 32*,no. 1, National Academy of Engineering, 2002.

[12] The President's Commission on Critical Infrastructure Protection. "Critical Foundations: Protecting America's Infrastructures." Government Printing Office, Washington D.C., U.S, 1997.

[13] Roesch, M. and C. Green, "Snort Users Manual Snort Release: 1.9.1", Online Documentation, available at http://www.snort.org/docs/writing_rules/, 2002.

[14] Sun Microsystems, Inc. "The Java Architecture for XML Binding User's Guide.", Early-Access Draft, Online Documentation, available at http://java.sun.com/xml/jaxb/jaxb-docs.pdf, 2001.

[15] Sun Microsystems, Inc. "Web Services Made Easier: The Java APIs and Architectures for XML.", Technical White Paper, Online Documentation, available at http://java.sun.com/xml/webservices.pdf, 2002.

[16] Lieberherr, K. J. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[17] Fowler, M. *Analysis Patterns: Reusable Object Models*. Addison Wesley, 1997.

[18] Foote, B., and J. W. Yoder. "Metadata and Active Object-Models." Technical Report, WUSC-98-25, Department of Computer Science, Washington University, 1998.

[19] Yoder, J. W. and R. Razavi. "Metadata and Adaptive Object-Models." *ECOOP '2000 Workshop Reader*, Lecture Notes in Computer Science, LNCS v 1964, Springer Verlag, 2000.

[20] Norvig, P. and D. Cohn. "Adaptive Software." PC AI Magazine, Jan 1997.

[21] Schwartz, D. G., S. Stoecklin, E. Yilmaz . "A Case-Based Approach to Network Intrusion Detection." Proceedings of the Fifth International Conference on Information Fusion, IF'02: 1084-1089, 2002.

[22] Guha, R., O. Kachirski, D. G. Schwartz, S. Stoecklin, E. Yilmaz. "Case-Based Agents for Packet-Level Intrusion Detection in Ad Hoc Networks." Seventeenth International Symposium On Computer and Information Sciences*,* 2002.

[23] Aamodt, A., E. Plaza. "Case-based reasoning: foundational issues, methodological variations and system approaches." *AI Communications 7*: 39-59, 1994.

[24] Thrift, P. "A neural network Model for cased-base reasoning." *Prooceedings of the DARPA Case-Based Reasoning Workshop*, editted by Morgan Kaufmann, 1989.

[25] Marling, C., M. Sqalli, E. Rissland, H. Munoz-Avila, D. Aha. " Case-based reasoning integrations." AI Magazine: 69-86, Spring 2002.

[26] Desmarais, N. *The ABC's of XML: The Librarians's Guide to the eXtensible Markup Language*. New Technology Press, 2000.

[27] Snort, The Open Source Network Intrusion Detection System. http://www.snort.org/.

[28] XML, Extensible Markup Language.http://www.w3c.org/XML/.

[29] INRECA Consortium. "CASUEL: A Common Case Representation Language." http://wwwagr.informatik.uni-kl.de/~bergmann/casuel/CASUEL_toc2.04.fm.html, 1994.

[30] Aha, D. W. "Navy Conversational Decision Aids Environment." http://www.aic.nrl.navy.mil/aha.

[31] NFR Security. "NFR: Network Flight Recorder." http://www.nfr.net/.

[32] Internet Security Systems, Inc. "RealSecure Network Sensor." http://iss.net/.

[33] Hochberg, J. et al. "NADIR, An Automated System for Detecting Network Intrusion and Misuse." *Computers and Security 12*, no. 3: 235-248, 1993.

[34] Sebring, M. M., E. Sellhouse, M. E. Hanna, R. A. Whitehurst. "Expert system in intrusion detection: A case study." *Proceedings of the 11th National Computer Security Conference*: 74-81, 1988.

[35] Tripwire Open Source Project. "Tripwire." http://www.tripwire.org/.

[36] Openwall Project. "Information security software for open environments." http://www.openwall.com/.

[37] LIDS Project. "LIDS: Linux Intrusion Detection System." http://www.lids.org/.

[38] Sullivan, K., J. Knight, X. Du, S. Geist. "Information Survivability Control Systems." *Proceedings of the 21st International Conference on Software Engineering*: 184-192, IEEE Computer Society Press, 1999.

[39] Eckmann,S. T., G. Vigna, R.A. Kemmerer. "STATL: An Attack Language for State-based Intrusion Detection." *Journal of Computer Security 10*: 71-104, 2002.

[40] Knight, J. C., M. C. Elder, X. Du. "Error Recovery in Critical Infrastructure Systems." Proceedings of Computer Security, Dependability, and Assurance: From Needs to Solutions, CSDA-98, IEEE Computer Society, 1999.

[41] Elson, D. "Intrusion Detection, Theory and Practice." Security Focus, 2000. http://www.securityfocus.com/focus/ids/articles/davidelson.html.

[42] Cover, T., P. Hart. "Nearest neighbor pattern classification." *IEEE Transactions on Information Theory 13*: 21-27, 1967.

[43] CERT Coordination Center. http://www.cert.org/.

[44] Anderson, D., T. Frivold, A. Valdes. "Next-generation intrusion detection expert system (NIDES): A summary." Technical Report, SRI—CLS—95—07, Computer Science Laboratory, SRI International, 1995.

BIOGRAPHICAL SKETCH

Erbil Yilmaz was born in Istanbul, Turkey, in 1977. He got his Bachelor of Science degree in Computer Engineering and his Bachelor of Science degree in Mathematics from the Bogazici University, Istanbul, Turkey, in July 1999. In January 2001, he joined to the Department of Computer Science at The Florida State University as a Ph.D. student. During this period, he completed an educational program certified by NSTISSC as compliant with NSTISSI No. 4011 for Information Systems Security (INFOSEC) Professionals, and received his INFOSEC certificate in 2002. In the same year, he got his Master of Science degree on Computer Science. Since May 2001, he has been working as a research assistant, and conducting research on information security and artificial intelligence applications on information security.