DISSERTATION

Network Alertness

Towards an adaptive, collaborating Intrusion Detection System

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften

unter der Leitung von

o. Univ.-Prof. Dipl.-Ing. Dr.techn. Mehdi Jazayeri
E184-1
Institut für Informationssysteme

eingereicht an der

Technischen Universität Wien
Fakultät für Technische Naturwissenschaften und Informatik

von

Dipl.-Ing. Christopher Krügel
chris@infosys.tuwien.ac.at

Matrikelnummer: 9525589
Schiffmühlenstrasse 116/7/36
A-1220 Wien, Österreich

Wien, im Mai 2002

Kurzfassung

Die vorliegende Dissertation beschäftigt sich mit Network Alertness, einem neuartigen Ansatz für anpassungsfähige, kooperative Intrusion Detection (Erkennen von Eindringlingen in einem Netzwerk). Der eigentliche Erkennungsprozess wird von kooperierenden Rechnern durchgeführt, die aus den verstreuten Einzelteilen ein einziges, zusammenhängendes Bild der gerade stattfindenden Angriffe ermitteln. Die Information über diese Angriffe wird dann dazu benützt, den eigentlichen Erkennungsvorgang zu verbessern, indem Daten von verdächtigen Quellen besonders genau analysiert werden.

Das Hauptaugenmerk des Designs liegt auf der Verteidigung von großen Unternehmensnetzen. Nachdem die Aktivität von Hackern und Kriminellen im Internet in den letzten Jahren stark zugenommen hat, muss sich praktisch jedes Unternehmen der Bedrohung stellen, die diese auf eines der wichtigsten Unternehmensgüter ausüben, nämlich der Bedrohung des eigenen Netzwerks und der daran angeschlossenen Ressourcen.

Gängige Intrusion Detection Systeme sind dieser Aufgabe nicht gewachsen, da sie mit dem enormen Datenaufkommen nicht fertig werden, das von den Sensoren in solchen Netzen produziert wird. Spezielle Knoten, deren einzige Aufgabe die Analyse der ankommenden Information ist (so wie zentrale Rechner), sind anfällig für Fehler und gezielte Angriffe. Außerdem begrenzen sie, weil sie leicht überlastet werden können, die Skalierbarkeit des Systems.

Unser Ansatz basiert auf einem verteilten Design, wo gleichberechtigte Knoten miteinander kooperieren, um auftretende Angriffe zu erkennen. Diese Angriffe werden als Muster dargestellt, die in der von uns entwickelten Attack Specification Language beschrieben werden können. Diese Beschreibungssprache hat den Vorteil, dass Intrusions von Experten intuitiv und einfach spezifizierbar sind.

Um eine exponentielles Ansteigen von Nachrichten zwischen den Knoten zu vermeiden, musste die Mächtigkeit der Beschreibungssprache reduziert werden. Das erlaubte uns die effiziente Entwicklung eines verteilten Suchalgorithmus, der gefährlichen Muster finden kann. Durch die Einschränkung des Identifikationsprozesses auf jene Knoten, wo die eigentlichen Indizien gefunden werden, und jeglichen Verzicht auf zentrale Stellen, war es möglich, eine Lösung zu entwerfen, die den jetztigen Systemen in Bezug auf Skalierbarkeit und Fehlertoleranz überlegen ist.

Der verteilte Algorithmus zusammen mit der Beschreibungssprache und Komponenten zur Verwaltung und Inbetriebnahme des Systems wurden implementiert. Die Architektur des gesamten Systems, das Quicksand getauft wurde, wird in der vorliegenden Arbeit beschrieben. Quicksand verfügt über ein flexibles Interface um zusätzliche Komponenten (wie zum Beispiel Sensoren von Drittanbietern) einbinden zu können.

Im Zuge der Arbeit wurde ein Sensor entwickelt und in Quicksand integriert, der Anfragen an öffentlich zugängliche Netzwerkdienste (wie das Web oder das Domain Name Service) auf mögliche Anomalien untersucht. Dieser Sensor ist in der Lage, sich auf Daten von unterschiedlichen Quellen einzustellen und sein Verhalten entsprechend zu verändern. Das erlaubt es den kooperierenden Knoten, die Sensoren so anzupassen, dass Anfragen von verdächtigen Benutzern genauer kontrolliert werden.

Der Beitrag dieser Dissertation ist die Entwicklung eines Konzepts und dessen Realisierung, welches es verteilten Knoten in einem Netzwerk erlaubt, effizient Eindringlinge zu entdecken. Der verwendete Algorithmus ist als peer-to-peer Prozess gestaltet, der es ermöglicht, die riesigen Datenmengen zu untersuchen, welche in großen Netzen entstehen. Zusätzlich erlaubt es unser Ansatz auch, die gewonnene Erkenntnis nicht nur als Alarm an einen Administrator zu schicken, sondern auch den Erkennungsprozess selbst (mittels der vorher erwähnten Sensoren) an sich entwickelnde Gefahrenszenarien anzupassen.

Abstract

This dissertation introduces the concept of Network Alertness, a novel approach to perform adaptive, collaborating intrusion detection. The detection process itself is realized by collaborating nodes that correlate and assemble the pieces of evidence, which are scattered over many hosts in the victim's network, into a single and coherent picture of ongoing attacks. The information of emerging threats is then fed back into the system and utilized to selectively adapt to data from suspicious sources.

The main focus of the proposed design is the protection of huge enterprise networks against coordinated attacks. As the activity of cyber-criminals has risen dramatically in the past few years, virtually any organization faces an increasing threat to one of its most valuable assets, namely its network and the attached resources.

Current systems fall short in dealing with the immense data volume that is produced by the sensors that are deployed in these large network installations. Dedicated nodes such as centralized processors become vulnerable to faults or targeted denial-of-service attempts and often represent performance bottlenecks.

Our approach is based on a distributed framework that enables nodes to collaborate in a point-to-point fashion to identify emerging hostile patterns. These patterns can be described in a declarative manner in our proposed Attack Specification Language. This helps domain experts to express intrusion scenarios in a more intuitive way.

In order to prevent an explosion in the number of messages that need to be transmitted, the specification language had to be restricted. The consequential decentralized algorithm to find events that satisfy such patterns was implemented and exhibits superior scalability and fault tolerant properties when compared to existing solutions. This is achieved by restricting the detection to only those hosts that witness actual parts of the attack. We abandon the idea of nodes with a dedicated task of correlating events as used in traditional centralized or hierarchical approaches because they limit scalability and are vulnerable to faults or attacks. The correlation framework, together with an interface to allow deployment and management of our collaborating sensors, was implemented and named Quicksand. We describe the system architecture and implementation of Quicksand and introduce a mechanism to integrate components such as third-party sensors into our design.

One component that we have developed for Quicksand is an anomaly based network probe. It monitors requests to publicly available services such as the web or the domain name service. The sensor is capable of adapting its inspection mechanism to closer examine data from suspicious sources. This enables collaborating nodes to reconfigure the sensors to tighten the analysis of information from users which are believed to act maliciously.

The contribution of this dissertation is the creation of a framework that enables nodes to gain knowledge of intrusive behavior in large networks. This knowledge gaining process is realized in a peer-to-peer fashion to solve the problem of managing the massive data volume produced in such network installations. The information extracted by this process is not only used to send alerts to a system administrator but also to modify the detection process itself and adapt it to currently evolving scenarios.

Acknowledgments

This work has been supported in part by the European Community under the Information Societies Technology Programme IST-12637 and by the 'Fonds zur Förderung der wissenschaftlichen Forschung (FWF)' under the project OPELIX (P13731-MAT).

I would like to thank Richard Kemmerer and Giovanni Vigna at the University of California in Santa Barbara for the great research opportunity they provided during my stay there.

I would also like to thank my colleagues at the Distributed Systems Group for their fruitful discussions and their support while doing research at this department. Special thanks to Engin Kirda and Wolfgang Kastner who proof-read this dissertation and made valuable comments and to Mehdi Jazayeri who guided me during my life as a Ph.D. student.

Special thanks to Thomas Toth who experienced together with me the good and bad times of undertaking a real world project. We spent many nights programming and were rewarded with unforgettable moments.

Most of all I would like to thank my family who provided a perfect environment for my studies and have supported me unconditionally at all times.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> It is easy to run a secure computer system. You merely have to disconnect all dial-up connections and permit only direct-wired terminals, put the machine and its terminal in a shielded room, and post a guard at the door.
>
> – F.T. Gramp and R.H. Morris

As a matter of fact, computer systems are not operated this way. In order to provide useful services or to allow people to perform tasks more conveniently, computer systems are attached to networks and get interconnected. This resulted in the world-wide collection of local and wide-area networks known as the Internet. Although ease of use and convenience are trade-offs with security, people often cannot or do not want to forfeit services provided by remote machines. Therefore they have to deal with a loss of security. This dissertation deals with an approach to mitigate their risks.

When a computer system is attached to a network, three areas of increased risk can be identified [20].

First, the number of points that can potentially serve as the source of an attack against your computer is increased. For a stand-alone system, physical access to the machine is a prerequisite to an intrusion. In the networked case, each host that can send packets to the victim can be potentially utilized by a hacker[1]. As certain services (such as web or name servers) need to be publicly available, each machine on the Internet might be the originator of malicious activity. This fact makes attacks very likely to happen on a regularly basis.

Second, the physical perimeter of the computer system is extended. For a single machine, everything is considered to be 'inside a box' (or at least in a close vicinity). The processor fetches data from memory which is read from secondary storage. Such data is (very well) protected from tampering and eavesdropping while transferred between the different entities. The same assumption is not true for data transferred over the network. Packets on the wire often pass areas and are forwarded by infrastructure devices that are completely out of control of the receiver. Messages can be read, recorded and later replayed

[1] The term hacker is used to describe persons with the malicious intend to gain unauthorized access to network resources. They are often referred to as crackers as well.

as well as modified on their journey. Especially in large networks such as the Internet, it is not trivial to authenticate the source that claims to be the message's origin.

Third, the number of services that networked machines typically offer is greater than the single authentication service of a stand-alone system. Although such a service (usually realized as login with password) may contain vulnerabilities, it is still only a single program which is comparatively simple. The authentication service mediates the ability to access files or to send e-mails through a single point. Networked computers, on the other hand, often offer by default a variety of remote connection possibilities to log in, access data or relay mail. All service processes (called *daemons*) implementing remote access may contain exploitable programming bugs or configuration errors which can lead to system compromise.

The classical solutions to reduce the risks introduced by connecting computer systems to larger networks are firewalls and the use of cryptographic techniques.

Firewalls provide a parting line between the outside Internet (or untrusted third party networks) and a trusted inside set of machines under the administrative control of the firewall owner. A firewall acts as a central point that allows specification of an access control mechanism which regulates the accessibility of services running behind it to hosts on the outside. This minimizes the number of potential targets and leads back to the situation where only a few controlled services (similar to the login service of a stand-alone system) can be publicly utilized.

The use of cryptographic techniques prevents the data from being read or modified when transferred over the network. The sender and the receiver agree upon a symmetric key or an asymmetric key pair that allows transformation of the sender's message (or clear text) into the corresponding ciphertext and its reverse operation back into the clear text at the receiver. This should prevent attacks against data in transit. Notice, however, that the sender is not authenticated by simply encrypting the message. Additional mechanisms such as digital signatures and a trusted third party (called certification authority) are needed to provide this level of security.

The strategies described above are important ways to improve the security of a computer system and to reach the level of a stand-alone system that is located in a shielded and guarded room. Nevertheless, certain services (e.g. `HTTP`, `DNS`) have to be anonymously available by everyone connected to the Internet. They can neither be protected by cryptography nor by firewalls.

1.1 Motivating Scenario

The following typical scenario describes the possible actions of an attacker and illustrates that an additional security approach is necessary which supports and augments the strategies sketched above. We assume that the intruder is targeting a network installation run by a small company as depicted in Figure 1.1. In order to sell their articles, this company has set up the small web site `www.victim.com` allowing on-line purchase of their articles. Being security aware, the web server is located behind a firewall which only allows inbound

standard HTTP requests, secure HTTPS requests and DNS queries. The standard queries have to be permitted to transfer the web page requests and replies between users and the server. The secure HTTPS connection is used to cryptographically protect the customer's credit card information when it is transmitted to process the payments and DNS queries have to be accepted to be able to resolve the victim.com domain names.

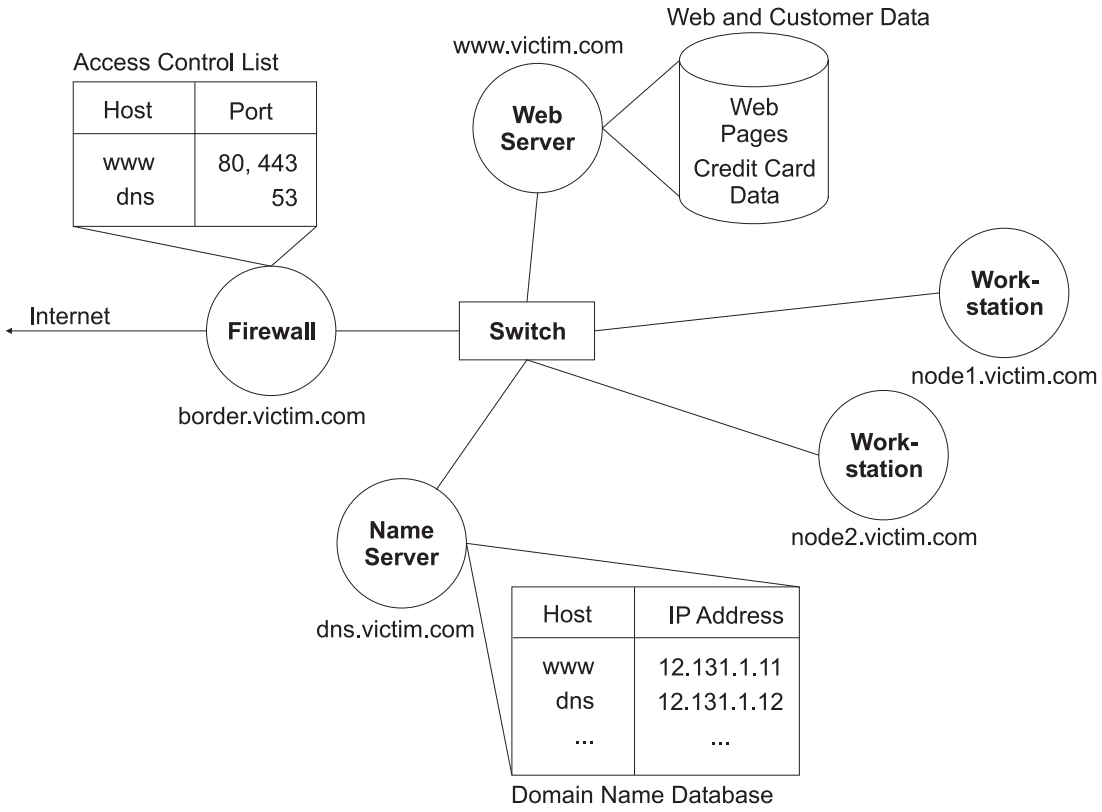

Figure 1.1: Victim Network Installation

A successful intrusion into a computer system can be divided into three stages [7], called

- Surveillance Stage,
- Exploitation Stage and
- Masquerading Stage.

During the surveillance phase, the attacker attempts to learn as much information as possible about its target to discover vulnerable services and configuration errors. The exploitation step describes the activity of actually elevating the attackers privileges by abusing an identified weakness and the masquerading stage covers all activity performed by an intruder after the successful break-in (e.g. deleting log entries or patching the vulnerability he used to get in).

The intruder starts his attack by obtaining the range of IP addresses owned (or controlled) by victim.com. This is done by querying their DNS server. Next, he launches a

port scan against each address he has found in the previous step. Such a scan attempts to find running services on a machine by sending packets which pretend to establish a connection to interesting ports at the target host and interpret the answers. When a server that is actually listening on a tested port receives such a request, it confirms the setup of the virtual connection with a reply packet. Otherwise an `ICMP` error message or no answer at all is returned. As the firewall in our example is properly configured, most of the connection request packets are discarded there. The attacker learns that only two machines provide a potential entry into the system. One is the `DNS` server, the other the web server.

He chooses to attack the web server and tries to figure out the type and version of the server software. It is very common that a weakness in a service is tightly bound to a certain version of the program implementing it. In addition, it is important to know the underlying operating system and hardware architecture because many exploits use short machine code programs (called *shell-code*) and utilize memory addresses with constant values. These values and the architecture specific programs obviously depend on both factors mentioned above.

To get that information, the intruder retrieves the complete `DNS` entry for the web server from the domain name database. The web master is a diligent person and has filled out the hardware information field, identifying the server as running `Windows 2000`. This allows the attacker to conclude that the web server is a `Microsoft Internet Information Service (IIS)`, an educated guess that can be verified by looking at the header of the reply to a standard web page request. For this product, a number of vulnerabilities [18] have been released recently that allow a hacker to send a carefully crafted packet to exploit a buffer overflow weakness and obtaining administrator access. This weakness has also been used by `Code Red` [19], a virus that targets `Microsoft`'s web server and has received much attention recently.

The intruder downloads one of the exploit programs readily available on the Internet and gives it a try. As the administrator has not applied the latest patches, he is successful. The exploitation stage has been very short. Notice that the firewall accepted and forwarded all relevant attack packets because everyone on the Internet is permitted to communicate with the company's web and name service. No violation against the access control policy has occurred. Nevertheless, an important network resource has been compromised. Also encryption cannot be used to prevent this kind of attack. As the data is not modified by a third party during transmission, protecting the packet's payload is futile. Even the authentication of the sender would not be beneficial, as the origin of the attack is not disguised in our example. Potential suspicious log entries at the server in case of authentication of every communication partner could later be removed by the attacker in this case.

After the compromise, the intruder enters the masquerading stage and immediately starts to remove the marks of his attack by cleaning entries from the web server and the operating systems logs that relate to his machine. In addition, he installs a *root kit* - a collection of programs that replace system binaries with trojaned ones. These trojan versions usually open a back door access for the hacker to provide future login possibility. They also modify administrative commands (e.g. file or process monitors, logging facilities) to hide the activity of the back door itself.

Now, the hacker owns a new base for launching attacks against other machines of `victim.com`. He can operate from a machine inside the network which is not subject to the firewall's access control. Alternatively, he could launch intrusions against other sites from there, hiding his true origin.

The given example makes it obvious that firewalls and cryptography cannot defend computer systems against certain classes of malicious behavior. When a machine is connected to the Internet and has to provide publicly accessible services, it automatically becomes a target in a hostile environment. Although the installation of updates and patches for used software is beneficial and necessary, it is naive to assume that one is always faster in installing bug fixes than the attacker who tries to exploit such a program error.

This makes it necessary to install an additional defense to cope with intrusions when the first perimeter of defense has been penetrated. Systems that attempt to detect malicious behavior that is targeted against a network and its resources are called *intrusion detection systems (IDSs)*. They are network security tools that process local audit data or monitor network traffic and operating system activity. IDSs can either search for specific patterns, called *signatures*, in their input stream (*misuse based*) or detect certain deviations from expected behavior (*anomaly based*) which indicate hostile activities against the protected network.

Intrusion detection systems constitute the third building block (together with firewalls and cryptography) of a secure computer system installation and can discover intrusions in all of the three stages listed above. Current state-of-the-art IDSs notice the huge amount of packets from a single source that are dropped at the firewall and report reconnaissance activity. It is also possible to check incoming packets for malicious (exploit) payload and flag or drop suspicious messages. This would have potentially prevented the compromise of the web server in the example above. In addition, an IDS can also monitor the integrity of the host's files. When the attacker installs trojaned binaries, the IDS reports these modifications to the system administrator.

1.2 The Vision - Network Alertness

While intrusion detection systems are efficient supplements to more traditional security mechanism, they are no panacea. A vile hacker may perform the scan over a long period of time or gather information directly from the `DNS` database thus evading the detection capability of the intrusion detection system. Or he can modify the exploit code obtained from the Internet to craft a packet that looks innocent.

The problem is that although an attacker leaves many traces at different spots in the target network during an intrusion attempt, most IDSs consider these pieces independent of each other and classify all of them as benign. Evidence of attacks against a network and its resources is often scattered over several hosts. IDSs have to collect and correlate information from different sources to spot complete attack scenarios such as the one described in the previous Section 1.1.

The aim of this dissertation is the creation of a framework where all hosts are aware

of activities that manifest themselves at different parts of the network and incidents that result in a distortion of regular network traffic. The nodes should develop an understanding of the complex connections inside the network and react selectively to intrusive behavior.

The main focus of our work is on a system that is scalable enough to operate in enterprise networks and capable of detecting coordinated large-scale attacks. Such an approach is needed as the number of cyber-crime incidents has been rising dramatically in the last few years. Virtually all organizations have experienced attacks from the Internet and suffered from malicious activity conducted by inside users. Many of them possess networks that are larger than current IDSs can handle. This leads to a situation where only small, important parts are thoroughly monitored while the rest is at most protected by a firewall. Unfortunately, as we have seen in the previous section, this is often not sufficient.

The problem of current state-of-the-art designs is the fact that the process of *event correlation is only done in a rudimentary fashion* by simply forwarding the distributed data to dedicated hosts where it is further processed. It is *not possible to specify complex attack scenarios* together with fine grain response that is evaluated *on more than one host in parallel.*

As networks and traffic grow, the central correlators become *performance bottlenecks.* While current approaches work reasonable well for mid-sized networks, large installations with several thousand hosts push them to and above their limits. A *scalable solution* is necessary which allows the correlation of events from different sources even for the largest enterprise networks. This requires a design where the total amount of traffic between all involved machines as well as the peak load at any single spot is manageable.

Because almost all systems rely on (a few) special hosts that are essential for the correct operation of the detection process, it is obvious that they are vulnerable to malicious attacks or faults. Fault tolerance is a property which specifies the percentage (fraction) of nodes of the complete network which have their events correlated after a single machine running parts of the IDS (sensor or correlator) fails or is taken out. This indicates the fraction of distributed patterns that can still be detected and is a measure for the resilience of the system. For *systems with central entities*, the *loss of important hosts can significantly reduce this percentage of covered nodes* - a fact which indicates their limited fault tolerance.

Our solution is a completely decentralized approach that models an intrusion as a pattern of events that occur at different hosts. Each of our protected nodes are equal partners in the detection process and collaborate in a peer-to-peer fashion. This allows us to perform the necessary cooperation in a scalable and fault tolerant way.

A scalable and fault tolerant schema to detect attack scenarios relies on detailed information from the underlying sensors that deliver the single pieces of evidence. Such sensors have to be integrated into the detection process by tuning them to analyze data in more detail from machines that are suspected to participate in an emerging intrusion.

Most intrusion detection systems operate similar to virus scanners and simply compare their input data to a database of known attacks. Only when the input matches such a pre-defined attack signature, an intrusion can be detected. This approach *lacks the possibility to spot previously unknown intrusions* or derivations thereof. For a system that attempts to combat skilled intruders who are able to adapt given attacks and change

their signature, more sophisticated techniques are needed. Some heuristics based systems follow the opposite approach where the input stream is matched against a profile of valid behavior. Any deviation that exceeds a certain threshold is considered an intrusion. Such systems are well suited to find previously unknown attack attempts but suffer from the shortcoming that *many times legal activity is reported as intrusive*. This often results in their deactivation by administrators after a short period of time.

We propose to introduce some a-priori knowledge into the detection process that allows modification of the threshold level to adapt to the input data from possible malicious sources. As *all current systems base their network anomaly detection on connection flows (i.e. multiple packets) instead of single ones*, this desired adaptation is not feasible there. That made the *development* of our own network based *adaptive anomaly detectors* necessary which can operate on a per-packet basis.

The combination of scalable, decentralized correlation and adaptive anomaly detection leads to our envisioned behavior of an intrusion detection system that we denote *Network Alertness*. Such a design has the capability of protecting large enterprise networks by assembling distributed pieces of evidence into a coherent picture of a coordinated attack in a scalable and fault tolerant manner **and** can selectively adapt and react to emerging threats (and their sources).

1.3 Contribution

The dissertation addresses the problems mentioned above and contributes to the following domains in the field of intrusion detection.

- The definition (grammar) of a language that is capable of describing attack scenarios as distributed patterns of events, which may occur at different places.
- The design and evaluation of a peer-to-peer algorithm which detects described scenarios without relying on dedicated correlation entities.
- The specification and evaluation of an `IP` based protocol that allows temporal ordering of events between communicating hosts with no bandwidth overhead.
- The implementation of the proposed distributed correlation framework in a prototype called *Quicksand*.
- A design to distribute the load when performing centralized correlation of events in high-bandwidth streams. The centralized probes are a complementary mechanism to the peer-to-peer framework in order to cover scenarios that cannot be modeled by the distributed patterns.
- The design and evaluation of an adaptive, anomaly based sensor that analyzes packet content instead of connection patterns to detect potential malicious behavior. The heuristical detection mechanism allows novel intrusions to be found, while the adaptive technique keeps the false positive rate acceptably low.
- The integration of all components into a coherent system that realizes our vision of *Network Alertness*.

1.4 Organization

This dissertation is organized as follows. Chapter 2 introduces the basic concepts of computer security and explains the key issues of the three main areas in network security, namely access control, cryptography and intrusion detection. The generic architecture and features of intrusion detection systems are evaluated in more detail.

Chapter 3 details previous work done in the areas of event correlation and adaptive anomaly sensors.

The next part, consisting of Chapter 4 and Chapter 5, explains one of the two cornerstones of *Network Alertness* and describes the concepts and implementation of the distributed event correlation. In addition, the protocol to perform event synchronization between communicating nodes in a network is introduced. Chapter 6 then explains a complementary approach to deal with events from a centralized perspective.

Chapter 7 is centered around the second cornerstone of *Network Alertness* and deals with the design and implementation of our adaptive anomaly sensors.

An evaluation of all system components is presented in Chapter 8. The last chapter concludes and outlines further research possibilities.

Chapter 2

Computer Security and Intrusion Detection

> The superior man, when resting in safety, does not forget that danger may come. When in a state of security he does not forget the possibility of ruin. Thus this person is not endangered, and his states and all their clans are preserved.
>
> – Confucius

The scenario in Section 1.1 described an exemplary threat to computer system security in the form of a hacker attacking a company's web server. This chapter attempts to give a more systematic view of system security requirements and potential means to satisfy them. We define properties of a secure computer system and provide a classification of potential threats to them. We also introduce the mechanism to defend against attacks that attempt to violate desired properties.

Before one can evaluate attacks against a system and decide on appropriate mechanisms against them, it is necessary to specify a *security policy* [95]. A security policy defines the desired properties for each part of a secure computer system. It is a decision that has to take into account the value of the assets that should be protected, the expected threats and the cost of proper protection mechanisms. A security policy that is sufficient for the data of a normal user at home may not be sufficient for a bank, as a bank is obviously a more likely target and has to protect more valuable resources.

2.1 Security Attacks, Properties and Mechanisms

For the following discussion, we assume that the function of a computer system is to provide information. In general, there is a flow of data from a source (e.g. host, file, memory) to a destination (e.g. remote host, other file, user) over a communication channel (e.g. wire, data bus). The task of the security system is to restrict access to this information to only those parties (persons or processes) that are authorized to have access according to the security policy in use.

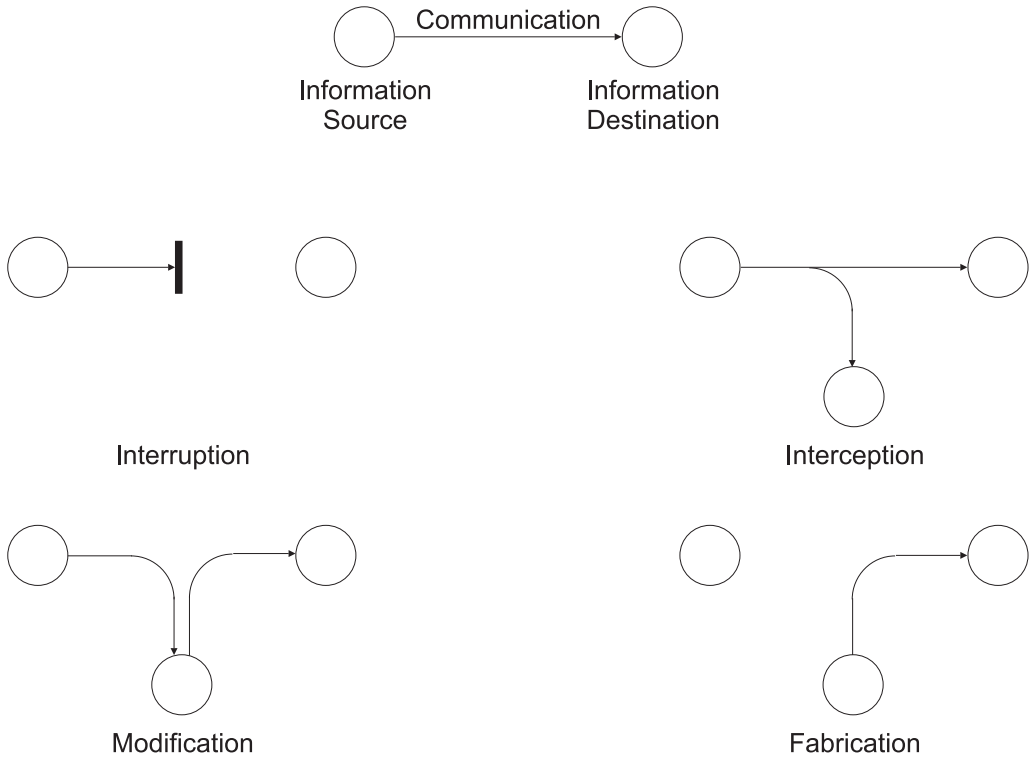

Figure 2.1: Security Attacks

The normal information flow and several categories of attacks that target it are shown in Figure 2.1 and explained below (according to [90]).

1. **Interruption:** An asset of the system gets destroyed or becomes unavailable. This attack targets the source or the communication channel and prevents information from reaching its intended target (e.g. cut the wire, overload the link so that the information gets dropped because of congestion). Attacks in this category attempt to perform a kind of *denial-of-service (DOS)*.
2. **Interception:** An unauthorized party gets access to the information by eavesdropping into the communication channel (e.g. wiretapping).
3. **Modification:** The information is not only intercepted, but modified by an unauthorized party while in transit from the source to the destination. By tampering with the information, it is actively altered (e.g. modifying message content).
4. **Fabrication:** An attacker inserts counterfeit objects into the system without having the sender doing anything. When a previously intercepted object is inserted, this processes is called *replaying*. When the attacker pretends to be the legitimate source and inserts his desired information, the attack is called *masquerading* (e.g. replay an authentication message, add records to a file).

The four classes of attacks listed above violate different security properties of the computer system. A security property describes a desired feature of a system with regards to a certain type of attack. A common classification is listed below [23, 66].

- **Confidentiality:** This property covers the protection of transmitted data against its release to non-authorized parties. In addition to the protection of the content itself, the information flow should also be resistant against traffic analysis. Traffic analysis is used to gather other information than the transmitted values themselves from the data flow (e.g. timing data, frequency of messages).
- **Authentication:** Authentication is concerned with making sure that the information is authentic. A system implementing the authentication property assures the recipient that the data is from the source that it claims to be. The system must make sure that no third party can masquerade successfully as another source.
- **Non-repudiation:** This property describes the feature that prevents either sender or receiver from denying a transmitted message. When a message has been transferred, the sender can prove that it has been received. Similarly, the receiver can prove that the message has actually been sent.
- **Availability:** Availability characterizes a system whose resources are always ready to be used. Whenever information needs to be transmitted the communication channel is available and the receiver can cope with the incoming data. This property makes sure that attacks cannot prevent resources from being used for their intended purpose.
- **Integrity:** Integrity protects transmitted information against modifications. This property assures that a single message reaches the receiver as it has left the sender but integrity also extends to a stream of messages. It means that no messages are lost, duplicated or reordered and it makes sure that messages cannot be replayed. As destruction is also covered under this property, all data must arrive at the receiver. Integrity is not only important as a security property, but also as a property for network protocols. Message integrity must also be ensured in case of random faults, not only in case of malicious modifications.

Different security mechanisms can be used to enforce the security properties defined in a given security policy. Depending on the anticipated attacks, different means have to be applied to satisfy the desired properties. Three main classes of measures against attacks can be identified, namely attack prevention, attack avoidance and attack detection. They are explained in detail in the following sections.

2.2 Attack Prevention

Attack prevention is a class of security mechanisms that contains ways of preventing or defending against certain attacks before they can actually reach and affect the target. An important element in this category is access control, a mechanism which can be applied at different levels such as the operating system, the network or the application layer.

2.2.1 Access Control

Access control [95] limits and regulates the access to critical resources. This is done by identifying or authenticating the party that requests a resource and checking its permissions against the rights specified for the demanded object. It is assumed that an attacker is not legitimately permitted to use the target object and is therefore denied access to the resource. As access is a prerequisite for an attack, any possible interference is prevented.

The most common form of access control used in multi-user computer systems are access control lists for resources that are based on the user and group identity of the process that attempts to use them. The identity of a user is determined by an initial authentication process that usually requires a name and a password. The login process retrieves the stored copy of the password corresponding to the user name and compares it with the presented one. When both match, the system grants the user the appropriate user and group credentials. When a resource should be accessed, the system looks up the user and group in the access control list and grants or denies access as appropriate. An example of this kind of access control can be found in the `UNIX` file system which provides read, write and execute permissions based on the user and group membership. In this example, attacks against files that a user is not authorized to use are prevented by the access control part of the file system code in the operating system.

2.2.2 Firewall

An important access control system at the network layer is a firewall [20]. The idea of a firewall is based on the separation of a trusted inside network of computers under single administrative control from a potential hostile outside network. The firewall is a central choke point that allows enforcement of access control for services that may run at the inside or outside. The firewall prevents attacks from the outside against the machines in the inside network by denying connection attempts from unauthorized parties located outside. In addition, a firewall may also be utilized to prevent users behind the firewall from using certain services that are outside (e.g. surfing web sites containing pornographic material).

For certain installations, a single firewall is not suitable. Networks that consist of several server machines which need to be publicly accessible and workstations that should be completely protected against connections from the outside would benefit from a separation between these two groups. When an attacker compromises a server machine behind a single firewall, all other machines can be attacked from this new base without restrictions. To prevent this, one can use two firewalls and the concept of a *demilitarized zone (DMZ)* [20] in between as shown in Figure 2.2.

In this setup, one firewall separates the outside network from a segment (DMZ) with the server machines and a second one this area from the rest of the network. The second firewall can be configured in a way that denies all incoming connection attempts. Whenever an intruder compromises a server, he is now unable to immediately attack a workstation. Such a setup might have helped the company running the network in our sample scenario in

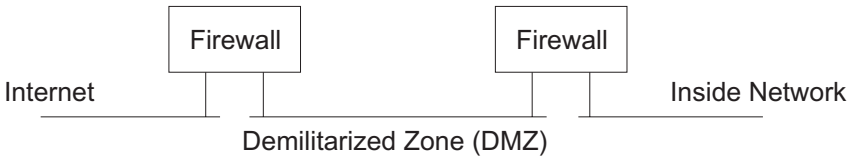

Figure 2.2: Demilitarized Zone

Section 1.1. Although the attacker gained access to the web server the rest of the network would still remain protected.

The following design goals for firewalls are identified in [20].

1. All traffic from inside to outside, and vice versa, must pass through the firewall. This is achieved by physically blocking all access to the internal network except via the firewall.
2. Only authorized traffic, as defined by the local security policy, will be allowed to pass.
3. The firewall itself should be immune to penetration. This implies the use of a trusted system with a secure operating system. A trusted, secure operating system (e.g. `Trusted-Solaris` [97]) is often purpose-built, has heightened security features and only provides the minimal functionality necessary to run the desired applications.

These goals can be reached by using a number of general techniques for controlling access. The most common is called *service control* and determines Internet services that can be accessed. Traffic on the Internet is currently filtered on basis of `IP` addresses and `TCP/UDP` port numbers. In addition, there may be proxy software that receives and interprets each service request before passing it on. *Direction control* is a simple mechanism to control the direction in which particular service requests may be initiated and permitted to flow through. *User control* grants access to a service based on user credentials similar to the technique used in a multi-user operating system. Controlling external users requires secure authentication over the network (e.g. such as provided in `IPSec` [42]). A more declarative approach in contrast to the operational variants mentioned above is *behavior control*. This technique determines how particular services are used. It may be utilized to filter e-mail to eliminate spam or to allow external access to only part of the local web pages.

A summary of capabilities and limitations of firewalls is given in [90]. The following benefits can be expected.

- A firewall defines a single choke point that keeps unauthorized users out of the protected network. The use of such a point also simplifies security management.
- It provides a location for monitoring security related events. Audits, logs and alarms can be implemented on the firewall directly. In addition, it forms a convenient platform for some non-security related functions such as address translation and network management.

- A firewall may serve as a platform to implement a virtual private network (e.g. by using `IPSec`).

The list below enumerates the limits of the firewall access control mechanism.

- A firewall cannot protect against attacks that bypass it, for example, via a direct dialup link from the protected network to an ISP (Internet Service Provider). It also does not protect against internal threats from an inside hacker or an insider cooperating with an outside attacker.
- A firewall does not help when attacks are against targets whose access has to be permitted.
- It cannot protect against the transfer of virus-infected programs or files. It would be impossible, in practice, for the firewall to scan all incoming files and e-mails for viruses.

Firewalls are usually divided into three common categories (refer to Figure 2.3) which are explained in more detail below.

Packet-Filtering Router

A packet-filtering router (or short packet filter) is an extended router that applies certain rules to the packets which are forwarded. Usually, traffic in each direction (in- and outgoing) is checked against a rule set which determines whether a packet is permitted to continue or should be dropped. The packet filter rules operate on the header fields used by the underlying communication protocols, for the Internet mostly `IP`, `TCP` and `UDP`. Packet filter rules are organized in a list with a certain default policy enabled. Every incoming packet is compared to the rules starting at the head of the list until the first list entry matches. In this case, the corresponding action is taken. When no matching rule can be identified, the default policy is consulted. It can either be a default discard or default forward decision. When a default discard policy is enabled, the packet is simply dropped, otherwise it is forwarded. A default discard policy is more conservative as it demands the system administrator to explicitly enable the needed services. The default forward policy increases the ease of use but reduces security as it requires the administrator to actively react on each threat and deny a certain connection. Packet filters have the advantage that they are rather cheap as they can often be built on existing hardware. In addition, they offer a good performance for high traffic loads. An example for a packet filter is the `iptables` package which is implemented as part of the `Linux 2.4` routing software.

Application-Level Gateway

An application-level gateway (also called proxy server) does not forward packets on the network layer but acts as a relay on the application level. The user contacts the gateway which in turn opens a connection to the intended target (on behalf of the user). A gateway

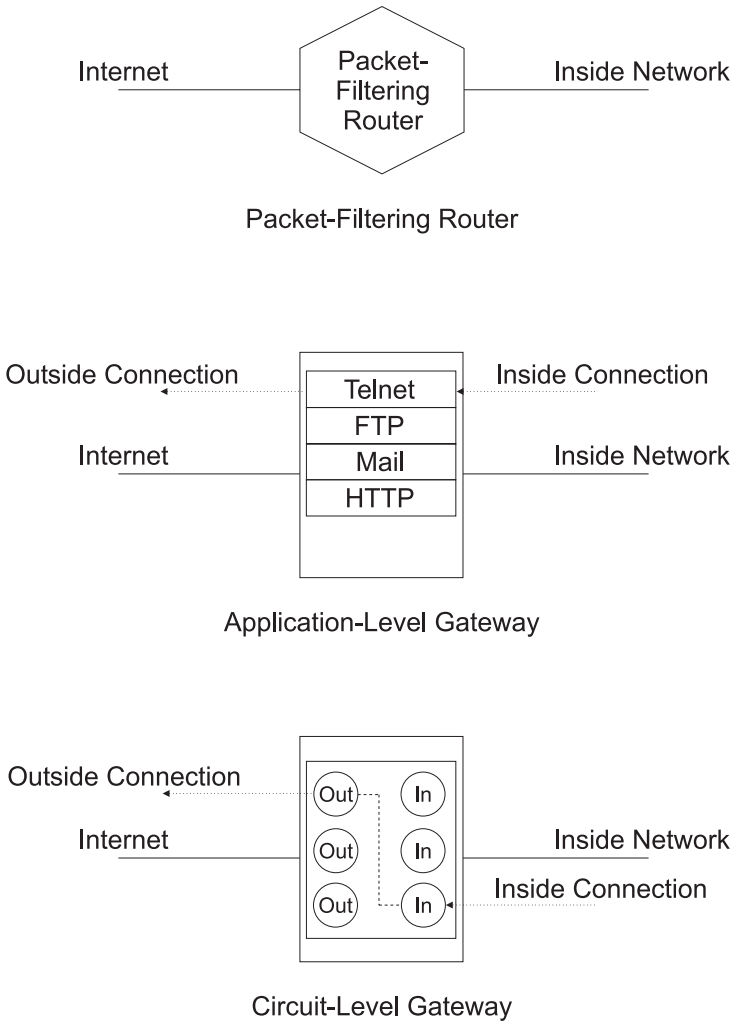

Figure 2.3: Firewall Types

completely separates the inside and outside networks at the network level and only provides a certain set of application services. This allows authentication of the user who requests a connection and session-oriented scanning of the exchanged traffic up to the application level data. This feature makes application gateways more secure than packet filters and offers a broader range of log facilities. On the downside, the overhead of such a setup may cause performance problems under heavy load.

Circuit-Level Gateway

A hybrid variant between the two firewall types mentioned above is a circuit-level gateway. Similar to the application-level gateway, the user first establishes a connection to the firewall which then contacts the specified target machine. In contrast to proxy servers however, a gateway in this category operates similar to a packet filter at the network level

after the connections have been set up. It relays segments between the connections without inspecting their contents. The security function of a circuit-level gateway consists of determining which connections are allowed. Such gateways are often used for outbound connections as they are faster than their application based cousins. In this case, connections can be restricted to certain users or times, but one is not really concerned with the transferred content.

2.3 Attack Avoidance

Security mechanisms in this category assume that an intruder may access the desired resource but the information is modified in a way that makes it unusable for the attacker. The information is pre-processed at the sender before it is transmitted over the communication channel and post-processed at the receiver. While the information is transported over the communication channel, it resists attacks by being nearly useless for an intruder. One notable exception are attacks against the availability of the information as an attacker could still interrupt the message. During the processing step at the receiver, modifications or errors that might have previously occurred can be detected (usually because the information can not be correctly reconstructed). When no modification has taken place, the information at the receiver is identical to the one at the sender before the pre-processing step.

The most important member of this category is cryptography which is defined as the science of keeping messages secure [82]. It allows the sender to transform information into a random data stream from the point of view of an attacker but to have it recovered by an authorized receiver (see Figure 2.4).

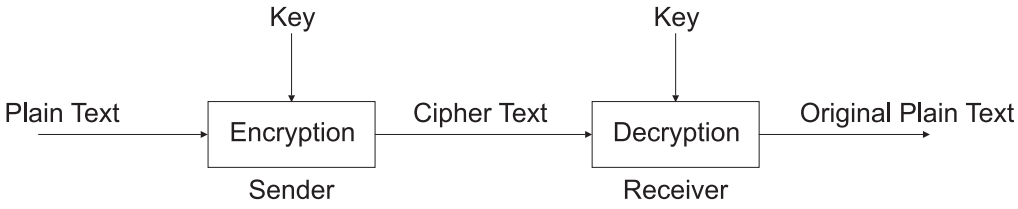

Figure 2.4: Encryption and Decryption

The original message is called *plain text* (sometimes clear text). The process of converting it through the application of some transformation rules into a format that hides its substance is called *encryption*. The corresponding disguised message is denoted *cipher text* and the operation of turning it back into clear text is called *decryption*. It is important to notice that the conversion from plain to cipher text has to be loss-less in order to be able to recover the original message at the receiver under all circumstances.

The transformation rules are described by a cryptographic algorithm. The function of this algorithm is based on two main principles: *substitution* and *transposition*. In the case of substitution, each element of the plain text (e.g. bit, block) is mapped into another element

of the used alphabet. Transposition describes the process where elements of the plain text are rearranged. Most systems involve multiple steps (called rounds) of transposition and substitution to be more resistant against cryptanalysis. Cryptanalysis is the science of breaking the cipher, i.e. discovering the substance of the message behind its disguise.

When the transformation rules process the input elements one at a time the mechanism is called a *stream cipher*, in case of operating on fixed-sized input blocks it is called a *block cipher*.

If the security of an algorithm is based on keeping the way how the algorithm works (i.e. the transformation rules) secret, it is called a restricted algorithm. Those algorithms are no longer of any interest today because they don't allow standardization or public quality control. In addition, when a large group of users is involved, such an approach cannot be used. A single person leaving the group makes it necessary for everyone else to change the algorithm.

Modern crypto systems solve this problem by basing the ability of the receiver to recover encrypted information on the fact that he possesses a secret piece of information (usually called the *key*). Both encryption and decryption functions have to use a key and they are heavily dependent on it. When the security of the crypto system is completely based on the security of the key, the algorithm itself may be revealed. Although the security does not rely on the fact that the algorithm is unknown, the cryptographic function itself and the used key together with its length must be chosen with care. A common assumption is that the attacker has the fastest commercially available hardware at his disposal in his attempt to break the cipher text.

The most common attack, called *known-plain text attack*, is executed by obtaining cipher text together with its corresponding plain text. The encryption algorithm must be so complex that even if the code breaker is equipped with plenty of such pairs and powerful machines, it is infeasible for him to retrieve the key. An attack is infeasible when the cost of breaking the cipher exceeds the value of the information or the time it takes to break it exceeds the lifespan of the information itself.

Given pairs of corresponding cipher and plain text, it is obvious that a simple key guessing algorithm will succeed after some time. The approach of successively trying different key values until the correct one is found is called *brute force* attack because no information about the algorithm is utilized whatsoever. In order to be useful, it is a necessary condition for an encryption algorithm that brute force attacks are infeasible.

Depending on the keys that are used, one can distinguish two major cryptographic approaches - public and secret key crypto systems.

2.3.1 Secret Key Cryptography

This is the kind of cryptography that has been used for the transmission of secret information for centuries, long before the advent of computers. These algorithms require that the sender and the receiver agree on a key before communication is started.

It is common for this variant (which is also called single key or symmetric encryption) that a single secret key is shared between the sender and the receiver. It needs to be

communicated in a secure way before the actual encrypted communication can start and has to remain secret as long as the information is to remain secret. Encryption is achieved by applying an agreed function to the plain text using the secret key. Decryption is performed by applying the inverse function using the same key.

Secret Key Block Cipher

The most obvious way of using a block cipher is called *electronic codebook (ECB).* In this setup, each block of plain text encrypts into a block of cipher text. This mode is called codebook because the same block of plain text is always mapped onto the same cipher text. Therefore it is (theoretically) possible to pre-calculate all mappings and store them in a book. The main advantage of this approach is the fact that blocks can be encrypted independently of each other. This allows parallelization of work or random access of cipher text blocks in database applications. On the downside, the ECB mode helps the cryptanalyst to mount statistical attacks against the underlying plain text. Especially when communication protocols with well-defined headers and footers are used, analysis can be much efficient. Also a block replay attack is possible. In this attack, the communication is modified without knowing the key by simply exchanging blocks for which the cryptanalyst knows the plain text.

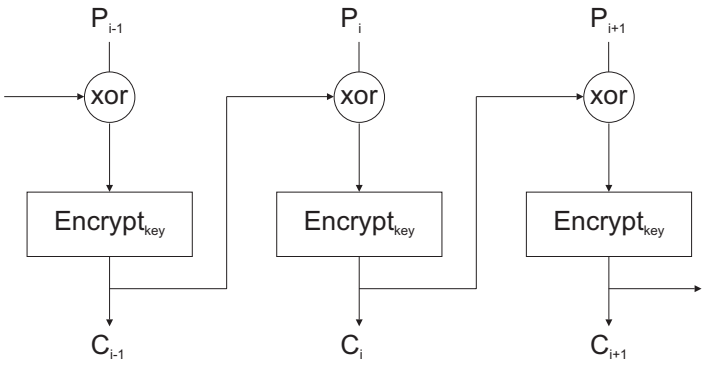

Figure 2.5: Cipher Block Chaining

A mechanism to prevent the simple mapping from plain text to cipher blocks is called *cipher block chaining (CBC).* In this variant, feedback is added to the encryption by adding the result of the previous block to the current block (see Figure 2.5). In other words, each block is used to modify the encryption of the following one. A simple way is by using the `XOR` operator to connect the current block with the one before. This usually maps identical plain text blocks to different cipher blocks. Notice, however, that identical messages are still encrypted into the same cipher texts. Moreover, the first difference in the cipher text corresponds to the first difference in the plain text. As this behavior is not desirable as well, most crypto systems utilize a random block called *initialization vector (IV)* in the beginning to prevent this problem.

The classic example of a secret key block cipher which is widely deployed today is the *Data Encryption Standard (DES)* [26]. DES has been developed in 1977 by IBM and adopted as a standard by the US government for administrative and business use. Recently, it has been replaced by the *Advanced Encryption Standard (AES - Rijndael)* [1]. It is a block cipher that operates on 64-bit plain text blocks and utilizes a key with 56-bits length. The algorithm uses 16 rounds that are key dependent. During each round 48 key bits are selected and combined with the block that is encrypted. Then, the resulting block is piped through a substitution phase (implemented by the well-known S-Boxes) and a permutation phase (implemented by P-Boxes). These S- and P-Boxes use known values and are independent of the key. They are simply utilized to make cryptanalysis harder.

Although there are no known weakness of the DES algorithm itself, its security has been much debated. The small key length makes brute force attacks possible and several cases have occurred where DES protected information has been cracked. A suggested improvement called 3DES uses three rounds of the simple DES with three different keys. This extends the key length to 168 bits while still resting on the very secure DES base.

Secret Key Stream Cipher

Secret key stream ciphers operate in a mode where a key stream is combined (usually via XOR) with the stream of plain text elements. In order to produce a key stream of arbitrary length from a key with a fixed length, random number generators are utilized. The secret key serves as an initialization vector for the random number generator whose output is then used to encrypt plain text. Obviously, the quality of the random number generator is important for the security of the crypto system. To improve the quality of the approach, the output of the random number generator does not only depend on the initial key value but is modified by the previous n cipher text elements as well. Such a feedback mechanism improves the randomness of the key stream.

The most popular secret key ciphers are based on feedback shift registers. A feedback shift register relies on a feedback function and a regular shift register which is a sequence of bits. Whenever a new key stream bit is needed the least significant bit is taken from the register and all other bits are shifted to the right. The new most significant bit is then calculated from the contents of the register by the feedback function. These kind of algorithms can be efficiently implemented in hardware but they are comparatively slow to simulate in software.

A well known stream cipher that has been debated recently is *RC4* [78] which has been developed by RSA. It is used to secure the transmission in wireless networks that follow the IEEE 802.11 standard and forms the core of the WEP (wired equivalent protection) mechanism. Although the cipher itself has not been broken the implementations are flawed and reduce the security of RC4 down to a level where the used key can be recovered by statistical analysis within a few hours [94].

2.3.2 Public Key Cryptography

Since the advent of public key cryptography, the knowledge of the key that is used to encrypt a plain text also allowed the inverse process, the decryption of the cipher text. In 1976, this paradigm of cryptography was changed by Diffie and Hellman [30] when they described their public key approach. Public key cryptography utilizes two different keys, one called the *public key*, the other one called the *private key*. The public key is used to encrypt a message while the corresponding private key is used to do the opposite. Their innovation was the fact that it is infeasible to retrieve the private key given the public key. This makes it possible to remove the weakness of secure key transmission from the sender to the receiver. The receiver can simply generate his public/private key pair and announce the public key without fear. Anyone can obtain this key and use it to encrypt messages that only the receiver with his private key is able to decrypt.

Mathematically, the process is based on the *trap door* of *one-way* functions. A one-way function is a function that is easy to compute but very hard to inverse. That means that given x it is easy to determine $f(x)$ but given $f(x)$ it is hard to get x. Hard is defined as computationally infeasible in the context of cryptographically strong one-way functions. Although it is obvious that some functions are easier to compute than their inverse (e.g. square of a value in contrast to its square root) there is no mathematical proof or definition of one-way functions. There are a number of problems that are considered difficult enough to act as one-way functions but it is more an agreement among crypto analysts than a rigorously defined set (e.g. factorization of large numbers). A one-way function is not directly usable for cryptography, but it becomes so when a trap door exists. A trap door is a mechanism that allows one to easily calculate x from $f(x)$ when an additional information y is provided.

A common misunderstanding about public key cryptography is thinking that it makes secret key systems obsolete, either because it is more secure or because it does not have the problem of secretly exchanging keys. As the security of a crypto system depends on the length of the used key and the utilized transformation rules, there is no automatic advantage of one approach over the other. Although the key exchange problem is elegantly solved with a public key, the process itself is very slow and has its own problems. Secret key systems are usually a factor of 1000 (see [82] for exact numbers) faster than their public key counterparts. Therefore, most communication is stilled secured using secret key systems and public key systems are only utilized for exchanging the secret key for later communication. This hybrid approach is the common design to benefit from the high-speed of conventional cryptography which is often implemented directly in hardware and from a secure key exchange.

A problem in public key systems is the authenticity of the public key. An attacker may offer the sender his own public key and pretend that it origins from the legitimate receiver. The sender then uses the faked public key to perform his encryption and the attacker can simply decrypt the message using his private key.

In order to thwart an attacker that attempts to substitute his public key for the victim's one, *certificates* are used. A certificate combines user information with the user's public

key and the digital signature of a trusted third party that guarantees that the key belongs to the mentioned person. The trusted third party is usually called a *certification authority (CA)*. The certificate of a CA itself is usually verified by a higher level CA that confirms that the CA's certificate is genuine and contains its public key. The chain of third parties that verify their respective lower level CAs has to end at a certain point which is called the root CA. A user that wants to verify the authenticity of a public key and all involved CAs needs to obtain the self-signed certificate of the root CA via an external channel. Web browsers (e.g. `Netscape Navigator`, `Internet Explorer`) usually ship with a number of certificates of globally known root CAs. A framework that implements the distribution of certificates is called a public key infrastructure (PKI). An important protocol for key management is `X.509` [106]. Another important issue is revocation, the invalidation of a certificate when the key has been compromised.

Examples of Public Key Crypto Systems

As mentioned above, the first public key system has been presented by Diffie and Hellman [30]. Its intended use is for exchanging a secret key between two communication partners. The security of system is based on the difficulty to determine the discrete logarithm in a finite field modulo a prime number q. The public information consists of the values for q and α, a primitive root[1] of the field modulo q. The generation of key K is done as shown below where X_s and X_r are secret random integers chosen by the sender and the receiver respectively.

$$Y_s = \alpha^{X_s} \; mod \; q \tag{2.1}$$

$$Y_r = \alpha^{X_r} \; mod \; q \tag{2.2}$$

Y_s and Y_r are then made public and each party can independently calculate the shared secret key K as follows.

$$K = Y_s^{X_r} = \alpha^{X_s * X_r} = \alpha^{X_r * X_s} = Y_r^{X_s} \tag{2.3}$$

The best known public key algorithm and textbook classic is `RSA` [79], named after its inventors Rivest, Shamir and Adleman at MIT. It is a block cipher in which the message text and the cipher text are integers between 0 and $n - 1$. The security is based on the infeasibility of factorizing large integers, as the public and the private keys are functions of large primes. The algorithm for encryption and decryption of a plain text M and the corresponding cipher C is as follows.

$$C = M^e \; mod \; n \tag{2.4}$$

$$M = C^d \; mod \; n = (M^e)^d = M^{(ed)} \; mod \; n \tag{2.5}$$

[1] A primitive root is a number whose powers generate all the integers from 1 to $q - 1$.

In the equations above, e and n are the public key while d is the private key used by the receiver. For this algorithm to be satisfactory as a public key crypto system, equation 2.5 must be correct and it must be infeasible to determine d from e and n. When e, d and n are chosen as follows, it can be shown that both requirements are met.

$$n = pq \; (where \; p \; and \; q \; are \; large \; primes) \tag{2.6}$$

$$\phi(n) = (p-1)(q-1) \tag{2.7}$$

$$Select \; e \; such \; that \; gcd(\phi(n), e) = 1 \tag{2.8}$$

$$d = e^{-1} \; mod \; \phi(n) \tag{2.9}$$

Although RSA is still utilized for the majority of current systems, the key length has been increased over recent years. This has put a heavier processing load on applications, a burden that has ramifications especially for sites doing electronic commerce. A competitive approach that promises similar security as RSA using far smaller key lengths is elliptic curve cryptography. However, as these systems are new and have not been subject to sustained crypto analysis, the confidence level in them in not yet as high as in RSA.

2.3.3 Authentication and Digital Signatures

An interesting and important feature of public key cryptography is its possible use for authentication. In addition to making the information unusable for attackers, a sender may utilize cryptography to prove his identity to the receiver. This feature is realized by *digital signatures*. A digital signature must have similar properties as a normal handwritten signature. It must be hard to forge and it has to be bound to a certain document. In addition, one has to make sure that a valid signature cannot be used by an attacker to replay the same (or different) messages at a later time.

A way to realize such a digital signature is by using the sender's private key to encrypt a message. When the receiver is capable of successfully decrypting the cipher text with the sender's public key, he can be sure that the message is authentic. This approach obviously requires a crypto system that allows encryption with the private key, but many (e.g. RSA) offer this option. It is easy for a receiver to verify that a message has been successfully decrypted when the plain text is in a human readable format. For binary data, a checksum or similar integrity checking footer can be added to verify a successful decryption. Replay attacks are prevented by adding a timestamp to the message (e.g. Kerberos [93, 48] uses timestamps to prevent that messages to the ticket granting service are replayed).

Usually, the storage and processing overhead for encrypting a whole document is too high to be practical. This is solved by one-way hash functions. These are functions that map the content of a message onto a short value (called *message digest*). Similar to one-way functions it is difficult to create a message when given only the hash value itself. Instead of encrypting the whole message, it is enough to simply encrypt the message digest and send it together with the original message. The receiver can then apply the known hash function (e.g. MD5 [77]) to the document and compare it to the decrypted digest. When both values match, the messages is authentic.

2.4 Attack and Intrusion Detection

Attack detection assumes that an attacker can obtain access to his desired targets and is successful in violating a given security policy. Mechanisms in this class are based on the optimistic assumption that most of the time the information is transferred without interference. When undesired actions occur, attack detection has the task of reporting that something went wrong and then to react in an appropriate way. In addition, it is often desirable to identify the exact type of attack. An important facet of attack detection is recovery. Often it is enough to just report that malicious activity has been found, but some systems require that the effect of the attack has to be reverted or that an ongoing and discovered attack is stopped. On the one hand, attack detection has the advantage that it operates under the worst case assumption that the attacker gains access to the communication channel and is able to use or modify the resource. On the other hand, detection is not effective in providing confidentiality of information. When the security policy specifies that interception of information has a serious security impact, then attack detection is not an applicable mechanism. The most important members of the attack detection class are intrusion detection systems.

This section introduces basic definitions used in the intrusion detection field to clarify common terms and describes the key concepts. The definitions are given according to [3].

- *Intrusion* - An intrusion is a sequence of related actions by a malicious adversary that results in the occurrence of unauthorized security threats to a target computing or network domain. An intrusion consists of a number of related steps performed by the intruder that violate a given security policy. The existence of a security policy that states which actions are considered malicious and should be prevented is a key requisite for an intrusion. Violations can only be detected, when actions can be compared against given rules.
- *Intrusion Detection (ID)* - Intrusion Detection is the process of identifying and responding to malicious activities targeted at computing and network resources. This definition introduces the notion of intrusion detection as a process, which involves technology, people and tools. The way malicious activities are identified will be subject of the remainder of this section, as it forms the central part of the ID process. A number of studies have attempted to provide taxonomies for intrusion detection, with the research done by `SRI` [89] and by Lindquist and Jonsson [60] being the most influential. These taxonomies identify key indicators associated with each intrusion type and divide them into different classes (nine in case of the SRI model).

A computer system, which performs the task of intrusion detection is called an *intrusion detection system (IDS)*. According to [28], an intrusion detection system has to fulfill the following requirements.

- *Accuracy* - An IDS must not identify a legitimate action in a system environment as an anomaly or a misuse (a legitimate action which is identified as an intrusion is called a *false positive*).

- *Performance* - The IDS performance must be sufficient enough to carry out real-time intrusion detection (real-time means that an intrusion has to be detected before significant damage has occurred - according to [76] this should be under a minute).
- *Completeness* - An IDS should not fail to detect an intrusion (an undetected intrusion is called a *false negative*). One has to admit that it is rather difficult to fulfill this requirement because it is almost impossible to have a global knowledge about past, present, and future attacks.
- *Fault Tolerance* - An IDS must itself be resistant to attacks.
- *Scalability* - An IDS must be able to process the worst-case number of events without dropping information. This point is especially relevant for systems that correlate events from different sources at a small number of dedicated hosts. As networks grow bigger and get faster, such nodes become overwhelmed by the increasing number of events.

2.4.1 System Architecture

In recent years, a large number of different intrusion detection systems have evolved which pursue the common goal of identifying intrusions. These systems utilize different approaches to accomplish the goal, but a common architectural framework can be identified (see Figure 2.6).

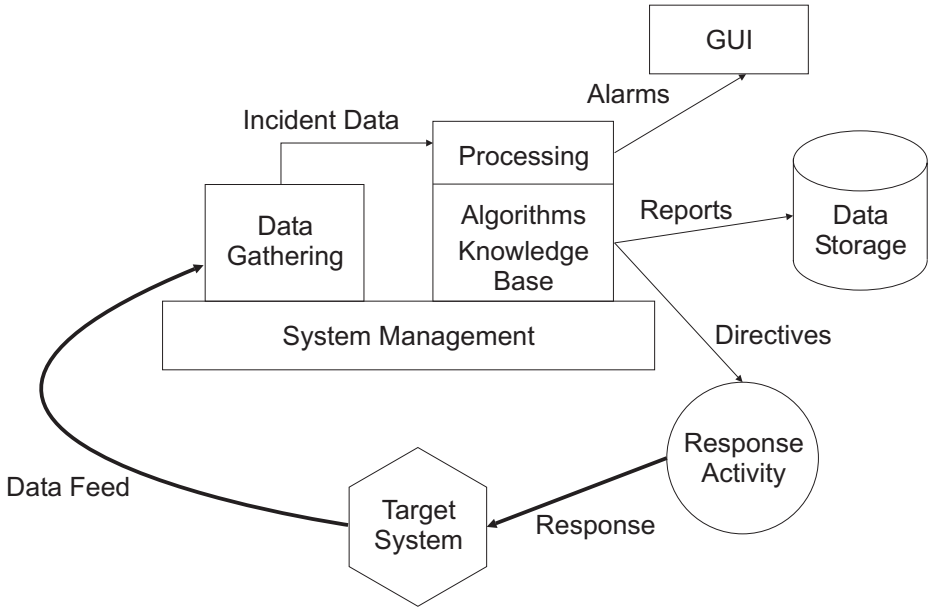

Figure 2.6: Intrusion Detection System Architecture

Intrusion detection systems monitor and collect data from a target system that should be protected, process and correlate the gathered information and initiate responses, when

evidence for an intrusion is detected. All intrusion detection systems basically consist of the following components.

- Data Gathering Component
- Data Processing Component
- Data Storage Component
- Response Component

2.4.2 Data Gathering

The data gathering component (often called sensor) is responsible for collecting data from the system that is being monitored.

A possible way of classifying different intrusion detection systems is by the location, where sensors are placed. In [8], three different approaches are listed. The first type is called *host based* and monitors events at operating-system level (e.g. connection attempts to a port or system calls). The second type describes sensors that collect data from network traffic (e.g. `IP` packet headers by network interfaces in promiscuous mode) and is called *network based*. The third type, called *application based*, receives input from running applications (e.g. log files) and can be considered as a special case of a host based IDS.

It is not a trivial problem to determine the points (called probe points) where event data should be recorded. While application based systems often rely on well-defined debugging or logging facilities provided by the program that is monitored, the situation is more difficult for the other two variants.

When considering host based systems, the variety of different operating system flavors and their different auditing facility poses a problem. In addition, it is often not clear which parts of the huge amount of data an OS kernel produces is actually relevant for the detection process. One suggestion for `UNIX` kernels provided by the US government is called the Orange Book [15] and lists 23 points where interesting data should be collected (e.g. IPC creation, fork). Another effort has been conducted by `Sun Microsystems` which has developed and implemented a standard security model called *Basic Security Model (BSM)* in its `Solaris OS` versions.

For network based systems, switches pose a big challenge in choosing a correct spot where the traffic should be gathered. The star-like network topology results in packets that are only routed between the relevant communication partners. When the IDS is just deployed on a single outgoing wire it would miss most of the traffic. An alternative placement of a sensor at every link is not very cost effective. One solution is to locate the ID probe at the uplink and monitor all traffic between the inside and outside networks (similar to a firewall). This has the problem of missing all communication inside the protected facility. Another variant is the *tap port* offered by many switch vendors in their products. This is a special port that mirrors and outputs all packets that pass through the device. Unfortunately, this port can be easily overloaded when the switch is forwarding traffic between different hosts in parallel. The internal bandwidth of the switch is sufficient

to handle many active ports at once but the traffic often exceeds the capacity of the tap (resulting in lost packets).

In recent years, there has been much debate about the superiority of host or network based approaches. Nowadays, most systems try to unify both variants and offer a hybrid solution.

The advantage of network based systems is that they can be set up in a non-intrusive manner with no effect on existing systems or infrastructure. As they do not reside on hosts that may be targeted by attacks, they are more tamper resistant. Additionally, most network based systems are OS independent and can derive information on a network level (e.g. packet fragmentation) that cannot be provided by a host based approach. One could think of these systems as intelligent switches that remain transparent to the rest of the system. A disadvantage is the weak scalability of this approach. Network based systems are infamous for dropping packets under heavy load and can hardly keep up with the speed of `Fast Ethernet` (not to mention `Gigabit Ethernet`). They also have difficulties when obsolete protocols are used (basically anything non `IP`). A major problem is the increasing use of cryptography for common network operations such as surfing the web via `SSL` [68] protected connections or working at remote machines with `SSH` [67]. This renders the packet payload unreadable for a network based design, reducing its detection capabilities tremendously.

Host based systems can collect high quality data, which can easily be configured (tuned) and may contain accurate kernel information. The data also has a high density, as logs often contain pre-processed information. On the other hand, host based systems often seriously impact performance of the machine they are running on.

Data gathering components store their data as a sequence of tuples in special files (called *audit-files*) to have them later examined by the data processing components. These tuples contain at least the originator (subject) and type of the action as well as the object, that is acted upon. An important aspect of data gathering components is the security and integrity of the collected data. The information collected by sensors form the base for the decisions made by the IDS. Therefore, they have to be secured against modifications.

2.4.3 Event Processing

The event data processing component forms the core of the IDS and has the responsibility to operate on the data collected by sensors to infer possible intrusions. There are a number of ways how intrusions might get detected and how intrusion detection systems can be categorized. Two broad main classes (introduced in [64]) are called *misuse* based and *anomaly* based.

Misuse Based ID

Misuse based intrusion detection relies on the a-priori knowledge of sequences and activities that form an attack. Such systems scan for the exploitation of well-known vulnerabilities or typical attack patterns. The sequences or patterns are called *signatures* of an attack and

can be compared to virus signatures. A misuse based IDS compares monitored activities with signatures that are stored in a database and raises an alarm when suspicious actions are found.

The following list describes how misuse detection can be performed. The first four are from [55].

- *Expert Systems* - Such systems code knowledge in databases as 'if-then' implication rules [44, 62]. The left side of the rule (*if*) defines the preconditions requisite for an attack. When all conditions are met, the rule triggers and the actions of the right side of the rule (*then*) are executed. This might lead to the firing of more rules or the inference of an intrusion. These system have the advantage of separating the control logic from the problem domain, but suffer from the limitation that rules are basically sequence-less. This makes it difficult to specify timing based steps of an intrusion.
- *Model Based Reasoning System* - This approach (proposed in [36]) is based upon a database of attack scenarios (stored as sequence of behaviors or activities). At any moment, the system considers a subset of these attacks as currently being carried out and tries to verify its assumptions using given audit-trails. When evidence for certain attacks are found, the likelihood of these attacks is increased, otherwise decreased. When the probability reaches a certain threshold, an intrusion is assumed and an alarm raised. This approach builds upon a sound mathematical theory of reasoning in case of uncertainty, but the attack models are difficult to build.
- *State Transition Analysis* - State Transition Analysis [40, 72] demands the construction of a finite state machine. The states of this automaton represent different system states with the transitions describing certain events that cause system states to change. A system state could represent the state of the network protocol stack, the validity and integrity of certain files or current running processes. When an intruder performs activities which cause the automaton to reach a state that is flagged as security threat, an intrusion is reported. The transitions between the states of the automaton allow to specify time dependent steps (sequences) of an attack.
- *Keystroke Monitoring* - This technique records user's keystrokes to determine possible intrusions. It simply watches for certain keystroke sequences to detect an attack. Unfortunately, the approach suffers from the possibility that an attacker can express the same actions with different keystrokes or use aliases.
- *Signature Detection* - This variant (an example is `Snort` [81]) watches for the occurrence of special strings or string patterns (called 'dirty words') that might be considered suspicious (e.g. watch for occurrences of `/etc/passwd` in a `telnet` session).

The advantage of misuse based ID is a reliable detection of known attack patterns. As with anti-virus software, malicious behavior can be identified with acceptable accuracy and a low false positive rate. The disadvantage originates from the fact that the attack pattern has to be known in advance so new intrusions will mostly remain undetected and the system can be easily fooled with slight deviations of the attack signatures.

Anomaly Based ID

Anomaly based intrusion detection attempts to identify malicious activities by comparing actual user behavior to a profile (which acts as a representation for expected, normal behavior). A profile serves as a metrics (a measurement of several variables) for regular, normal user behavior. Anomaly detection depends on the assumption that users behave regularly enough such that any significant deviation can be considered as evidence of an intrusion. Anomaly based ID has to be initialized with a default profile that is gradually adapted to users' behavior. Heuristics and statistical mechanisms are used to accommodate changes in users' behavior as well as to detect sudden changes. Other approaches try to incorporate artificial intelligence techniques such as neural networks to perform this task. The foundations of anomaly based intrusion detection have been established in a well known paper written by Dorothy Denning [29], who first suggested the use of profiles and statistical means (mean, standard deviation) to compare them to actual behavior.

This method has the clear advantage of being able to identify a-priori unknown attacks. Independent of the intruder's way to access the system, as soon as his activities (e.g. chance system files, try to obtain root access) are different enough from normal usage, the IDS might detect him. The disadvantage of this process is the involved lack of clarity (fuzziness). An intruder might act slowly and perform his actions carefully to modify the user's profile so that his activities are eventually accepted as legal when they should raise an alarm. The main point of criticism of anomaly based systems is the fact that they create too many false positives/negatives. Behavior that deviates from the expected profile does not necessarily indicate an intrusion and current implementations cause up to a few hundreds false alarms per day when used in a mid-size network. Especially network based systems face many 'weird' packets that populate the Internet nowadays [70, 10] caused by a massive number of different implementations of protocol stacks and services. Network administrators tend to ignore warnings occurring with such a high frequency or disable the system altogether. Therefore most commercial systems follow a misuse based approach. Another problem involves the classification and accurate description of the attack. Often, it is not sufficient to simply report a misbehavior (i.e. just state that 'something is abnormal') without possible origins.

2.4.4 Audit and Incident Data Storage

It has already been mentioned that data which is collected by sensors need to be stored. The amount of information that has to be archived can quickly become very large (e.g. when packets from a `Fast Ethernet` are captured and stored). Therefore, a storage hierarchy has to be introduced that allows one to reduce the data volume, but still makes it possible to relate events which are separated by days or even months. Usually, one can distinguish between three different types of data - short, medium and long-term information. Short-term data is stored in sensor buffers or data structures directly after it has been acquired. When the data is pre-processed and stored in audit-files to be examined by processing components, the data is considered medium-term information with a storage duration of a

couple of days. Eventually, the data is filtered (which will likely cause lossy compression) and stored into databases, where it can be queried and retrieved for identifying long-term activities (with a storage duration of months to years). The possibilities of data storage range from proprietary file formats to fully-fledged SQL database systems.

2.4.5 Response Component

The response component is the part of the IDS that has to initiate actions when an intrusion is detected. Responses can either be automated (active) or involve human interaction (passive). An automated response can immediately act against the attacker when an intrusion is detected and still in progress. This could be done by terminating network connections or tracing back the attacker's origin. A response that involves human interaction (e.g. by raising alarms or notifying the system administrator) is considered passive, because the system itself does not initiate response activities but relies on external intervention. Current systems mainly offer log and report responses where an e-mail or an `SMS` is sent to a responsible human. The problem with active counter-measures against intruders is a possible self denial-of-service. It has already been stated that many alarms are incorrectly raised. An IDS that immediately terminates connections or kills processes in case of suspicious activities may impact innocent users whose work flow is interrupted. E-commerce sites, for example, are especially afraid of losing customers in this way. Interestingly, large electronic warehouses (e.g. `amazon.com`) employ people that manually monitor traffic on-line to detect and to react on malicious activity.

2.4.6 International Standardization Efforts

Many vendors and research centers have developed their own version of an intrusion detection system. These systems apply different methods to perform their tasks and most have a class of intrusions which they can detect best. As a result, it might be desirable to combine several IDSs to protect a computing facility. Unfortunately, most of them are built in a monolithic way and use proprietary formats to store and transmit data. To circumvent these limitations, the *Common Intrusion Detection Framework (CIDF)* was introduced [46]. The CIDF aims to allow IDSs to share information and to communicate via well defined interfaces. It consists of the following three parts (the interested reader is referred to [21] for more details).

- A CIDF architecture with generic components (modules) and their APIs that build up the IDS. The components are called boxes and divide the IDS into the same four components which have been described in Section 2.4.1. The data gathering components are called event generators (*E-boxes*), the processing modules are event analyzers (*A-boxes*), the storage modules event databases (*D-boxes*) and the response component response units (*R-boxes*). These components communicate with each other in the form of *generalized intrusion detection objects (gidos)* which encapsulate the occurrence of a certain activity at a particular point of time.

- A language (*Common Intrusion Specification Language - CISL*) that allows these components to exchange data in semantically well-defined ways.
- A message and directory specification that defines the way in which components can locate and authenticate each other.

Chapter 3

Related Work

> Works of great intellect are great only by comparison with each other.
>
> – Ralph Waldo Emerson

The following section surveys work which has been previously done in the areas of intrusion detection that are related to our proposed vision of *Network Alertness*. As stated earlier, *Network Alertness* is based on two main concepts. One deals with the correlation of events that occur at different nodes to increase the overall understanding of malicious activity inside the network. The other suggests adaptive sensors relying on the gathered information to aim their detection processing especially at traffic from suspicious sources.

The first three sections examine how different system designs handle event correlation. The approaches are introduced in increasing level of sophistication while shortcomings of each variant are highlighted. It is underlined why a new approach as presented in the following Chapters 4 and 5 of this dissertation is necessary and beneficial.

The last two sections deal with the work previously done to perform anomaly based detection. After a brief introduction of host based variants, the features of network based designs are analyzed. It turns out that all of them follow a network traffic model, an approach where statistical properties of many packets are aggregated and processed. As *Network Alertness* demands the capability to look at single packets, a new system had to be developed. Our suggestion for an adaptive sensor that fulfills our demands is shown in detail in Chapter 7.

3.1 Centralized Event Correlation

The first attempts to extend intrusion detection from a single node to a set of hosts only addressed data gathering. The proposed designs included sensors deployed at each host of a local network that forwarded their collected data to a central point where it was further analyzed (see Figure 3.1).

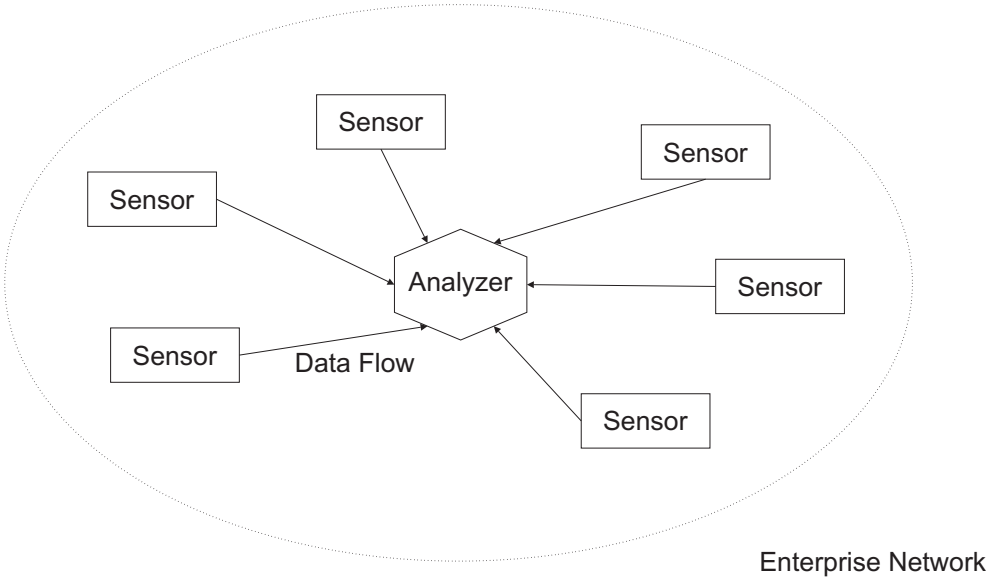

Figure 3.1: Centralized Correlation Schema

It has been noticed very early [86, 91] that the amount of data created at many hosts could overload the central processing node. Therefore, data reduction schemes have been introduced that select interesting parts of the audit stream and compress them.

3.1.1 DIDS

The earliest system that combined distributed monitoring and data reduction with centralized data analysis is called Distributed Intrusion Detection System (DIDS) [86]. The components of DIDS are a single *DIDS director*, a host monitor on each host and a *LAN monitor* for every broadcast segment in the monitored network.

The host and LAN monitors are responsible for the collection of evidence of unauthorized or suspicious activity while the DIDS director is responsible for its evaluation. The monitors can asynchronously send reports containing indication of malicious activity to the director. While these messages make up for the majority of transferred data, the underlying infrastructure provides for bidirectional communication. This enabled the director to request more details from its monitors.

Monitors scan audit records for notable events. These are defined as transactions that are of interest independent of any other records. Such events include failed user authentications, changes to the security state of the system and any network access such as `rlogin` and `rsh`. The LAN monitor uses and maintains profiles of expected network behavior. These profiles consist of expected data paths (e.g. which systems are expected to establish communication paths to which other systems and by which service) and service profiles (e.g. what a typical `telnet`, `mail` or `finger` session is expected to look like). LAN monitors utilize heuristics in an attempt to identify the likelihood that a particular

connection represents intrusive behavior. The host monitor is based on `C2`-secured [15] operating systems (e.g. `Sun Solaris` with security package) that produce audit records for most transaction on the system. These records include file accesses, system calls, process executions and logins. The director is realized as a rule-based (or production) expert system written in `Prolog`. The system considers the events received from its sensors in a spatial and temporal context to detect threat scenarios.

DIDS implements an interesting approach to track users as they move across the network. As an intruder may use several different accounts on different machines during the course of an attack, it is necessary to aggregate this data under a single identity. This is solved by a unique *network-user identification (NID)* that is assigned the first time a user enters the monitored environment. Whenever he accesses remote machines, his NID is moved along with him.

The two main threat scenarios that DIDS has been designed to handle are so-called *doorknob attacks* and *network browsing*. In a doorknob attack, the intruder's goal is to discover and gain access to insufficiently protected hosts. This is done by trying a few common account and password combinations on each of a number of computers. These simple attacks can be remarkably successful [58]. As the attacker only performs a few logins on each machine (usually with different account names), the IDS on each host may not flag the attack. Network browsing occurs when a user is looking through a number of files on several different computers within a short period of time. The browsing activity level on any single host may not be sufficiently high to raise any alarm by itself.

Because events that are generated by doorknob or network browsing attacks are independent of each other and potentially suspicious for themselves, a large amount of low-level filtering and some analysis can already be performed by each host monitor. This helps to minimize the use of network bandwidth in passing evidence to the director. When the notability (and suspicion level) of events depends on other events that occur somewhere else, this simple data reduction scheme does not work anymore. In that case, all events that *might* be relevant need to be transmitted to the director.

3.1.2 NSTAT

NSTAT [47] is part of the STAT (State Transition Analysis Technique) tool collection [102, 40, 72] which is based on a common detection technique (namely state transition analysis) and is still under active development. In state transition analysis, a penetration is viewed as a sequence of actions performed by an attacker that leads from some initial state on a system to a target compromised state, where a state is a snapshot of the system representing the values of all volatile and permanent memory locations on the system. The initial state corresponds to the state of the system just prior to the execution of the penetration and the compromised state corresponds to the state of the system resulting from the completion of the penetration. Between the initial and compromised states are one or more intermediate state transitions that an attacker performs to achieve the compromise.

After the initial and compromised states of a penetration scenario have been identified, the key actions (called *signature actions*) are identified. Signature actions refer to those

actions that, if omitted from the execution of an attack scenario, would prevent the attack from completing successfully. The information produced by the above steps are represented graphically as a state transition diagram. The signature actions depend on the domain where STAT is used. To date, host based extensions to handle `UNIX`, `Windows` and `Apache` audit files as well as network based sensors have been developed.

Figure 3.2 shows a state diagram that models the exploit of a vulnerability in the `mail(1)` utility of a `4.2 BSD` systems to obtain `root` privileges. The weakness is that `mail(1)` fails to reset the `s(et)uid` bit of the file to which it appends a message and changes the owner. The attacker can create a link to the system shell, name it after `root`'s mail file and enable the `suid` bit. When `mail(1)` is invoked to deliver a message to `root`, it changes the owner of the mail file back `root` but fails to reset the `suid` bit. This results in a link to a shell that is executable with `root` privileges.

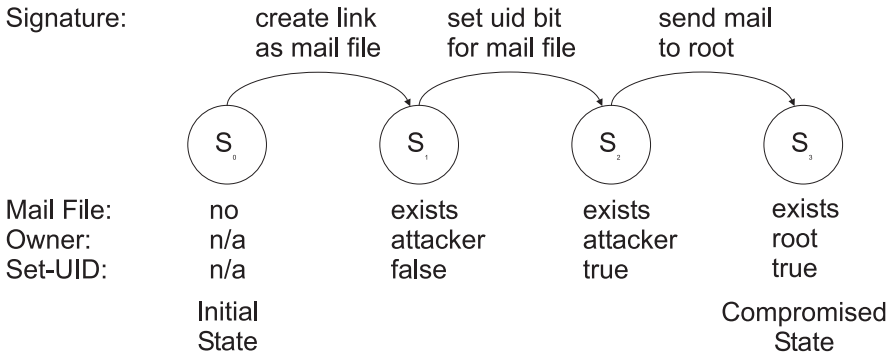

Figure 3.2: State Transition Diagram

A natural extension to the STAT effort is to run it on audit data collected by multiple hosts. This is realized as a single STAT process with a single, chronological audit trail. This trail is produced by distributed sensors that forward their data to a central entity which merges them together. To make sure that all distributed data is inserted into the global stream in a correct temporal order, the system relies on periodical synchronization points between the analyzer and the sensors as well as on synchronized clocks on all nodes. The advantage of a single audit file is the fact that actions of cooperating attackers can be identified. When transitions between states fire, it is also possible to record the responsible user (i.e. the one who has initiated the corresponding event). In addition, it is not possible to miss important events because no pre-filtering is utilized. Unfortunately, moving all data to a single node results in a scalability problem. When too many sensors are deployed, the central host is simply overloaded as probes in a large network often produce more data than the bandwidth of the single node can cope with. During the time the synchronization is performed, the detection process is halted. This causes an additional delay before the system can react to an intrusion. The use of a single node also induces a fault tolerance problem. When it crashes or becomes the victim of a denial-of-service attack, the system is completely blinded.

NSTAT is a straightforward extension of a single node intrusion detection setup. It

exhibits the scalability limitation of a simple, centralized correlation approach. While DIDS attempts to circumvent that problem by applying a simple pre-filtering technique at the sensor level, it misses potential important events because they look innocent when considered locally. Both, DIDS and NSTAT suffer from a vulnerable single central node.

3.2 Hierarchical Event Correlation

Early in the development of distributed intrusion detection systems, the single node bottleneck of a centralized approach became evident. In addition, it was noted that an efficient data reduction scheme at the leaves that is capable of forwarding only the relevant data for arbitrary threat scenarios was very difficult to realize. This lead to the development of hierarchical variants [24, 91, 9, 73] were the computational and network load is distributed over a number of intermediate analyzers. These analyzers attempt to perform detection for a small domain of the whole network and send all reports that might indicate attacks against the complete installation to a master (or root) node (as shown in Figure 3.3). This master host correlates all cross-domain incidents to gain a complete picture of the network.

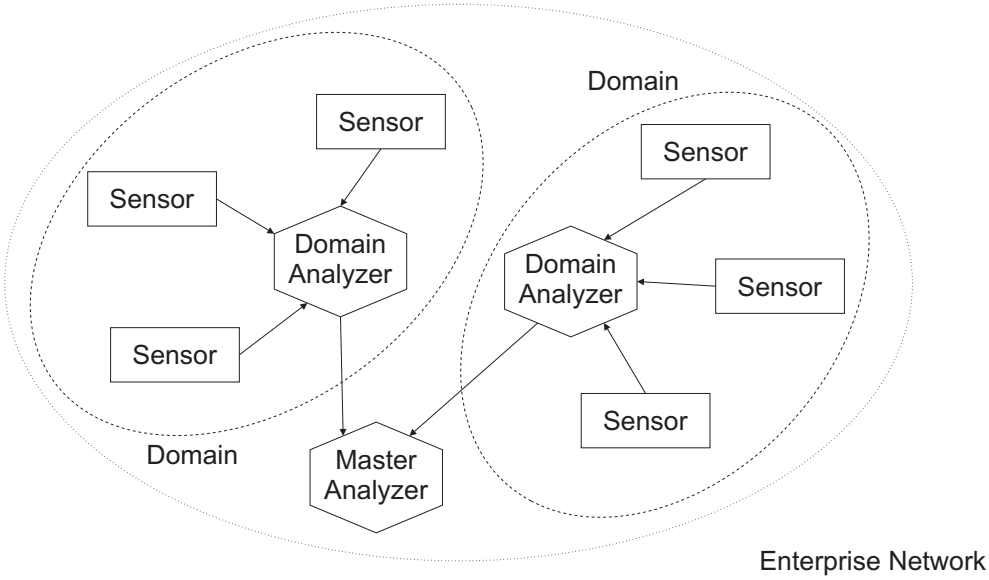

Figure 3.3: Hierarchical Correlation Schema

The following sections first introduce one of the earliest hierarchical systems and then describe two current state-of-the-art designs that serve as a reference for novel approaches.

3.2.1 GrIDS

GrIDS [91] (GRaph-based Intrusion Detection System for large networks) is an IDS that has been built to detect distributed attacks against large networks (distributed scans and

worms). Its primary design goals are scalability and an easy integration into large enterprise networks. Instead of simply forwarding all audit data collected from the operating system and the network, the main idea is the construction of activity graphs that only represent hosts and the network activity between them. Each node of the graph represents a single host or alternatively a set of aggregated machines. The edges represent network traffic between nodes. GrIDS can utilize standard host based IDSs (and port surveillance programs such as `tcpwrapper` [100]) as well as network sniffers as information source. The information is fed into a graph engine module which is responsible to construct graphs.

These graph engines can be deployed in an hierarchical fashion, where a higher level engine takes subgraphs from lower level modules and treats each subgraph as a single node in its graph representation. An organization is broken down into a number of departments, which each run their own graph engine. Every department passes its resulting graphs upward to parent graph engines, which in turn build reduced graphs by aggregating the information of the subgraph into a single node.

This mechanism increases the scalability of the system as no engine has to manage the whole graph (which could get very large when an enterprise network with thousands of computers should be monitored). An access control system controls the permissions of users to view and manage parts of the hierarchy. A centralized *organizational hierarchy server (OHS)* and software manager modules, which run on every host with a GrIDS engine, enforce the access policy. Each user may only see a subset of the global hierarchy and perform transactions (such as adding/deleting a new host) within his view. The OHS maintains a global view of the whole system and has the potential to limit scalability as a central point for the whole installation. Fortunately, this shortcoming does not really present a burden, as it is only involved in updating the system view but does not participate in regular operations.

An important aspect of the system is the way graphs are built. Each graph represents a causally connected set of events on the network. It has already been stated that a node represents a single machine or department (consisting of a set of hosts) while edges represent traffic between them. Each node or edge can be annotated with additional attributes that hold supplementary information. As GrIDS searches for different kinds of network abuse, each graph engine has to build and maintain different graphs. Each kind of graph features special attributes and is responsible for the detection of a certain class of attacks.

Every graph is constructed in a flexible way by specifying rule sets. A rule set defines how a given graph is constructed from the input data provided by the data sources in the form of reports. It also determines the actions that have to be taken as a result of a detected attack (e.g. alert an administrator). Rule sets operate independently from each other and prevent the graph engine from building one huge graph containing all activities. Instead, a number of smaller graphs tailored to special kinds of attacks are created.

A rule set consists of preconditions, combining rules and actions. The precondition determines whether certain input data (i.e. a report) should be considered for the graph space. When the precondition is satisfied, the corresponding report is imported as a partial graph in the appropriate rule set's graph space. The combining rules define ways how new subgraphs can be connected or inserted into existing graphs. When a new graph has been

added, the result is evaluated and action rules determine activities which might be initiated (e.g. raise alarm). All rule sets are present at every graph engine and each engine can assemble graphs independently of others.

To prevent the graphs from growing indefinitely, edges are removed after a certain time period. This poses the problem of missing attacks that deliberately take place very slowly. One interesting application of rule sets is their potential use as security policies. One can specify rules to enforce certain security constraints (e.g. disallow logins between different departments) and GrIDS automatically monitors the network for violations.

GrIDS offers a scalable aggregation mechanism with the introduction of different layers that only monitor their relevant parts of the graph. By reducing the signatures of threat scenarios to graph patterns, a simple and elegant aggregation mechanism can be achieved. This allows processing of all data at the graph engines without overloading them. Nevertheless, the system offers no protection against attacks (e.g. denial-of-service) and redundancy in case of failures. Additionally, scenarios that do not manifest themselves as connection graphs cannot be caught. This includes all local attacks and intrusions that produce normal network traffic (i.e. buffer overflow exploit).

3.2.2 Emerald

The Emerald (Event Monitoring Enabling Responses to Anomalous Live Disturbances) system [73] is an intrusion detection system developed by `SRI` [89]. It combines anomaly based and misuse detection mechanisms and focuses on providing a system for large-scale enterprise networks. These networks can be divided into independent domains that exhibit different trust relationships and security policies. Emerald builds upon IDES [63] (Intrusion Detection Expert System) and NIDES [4] (Next-Generation Intrusion Detection Expert System) and extends their work by providing a higher degree of distribution to enhance scalability and interoperability.

Emerald uses *service monitors* as the basic piece of its architecture. It introduces a hierarchically layered approach to network surveillance consisting of three tiers. The lowest level (called service analysis) covers the misuse of individual components and network services within the boundary of a single domain. The medium layer (called domain-wide analysis) covers misuse across multiple services and components and the highest layer (called enterprise-wide analysis) coordinates activities across multiple domains.

The objective of the service analysis is to decentralize the surveillance of the domain's network interfaces for activity that may indicate misuse or significant anomalies in operation with the use of service monitors. These monitors perform localized analysis of the infrastructure (e.g. routers or gateways) and services. Information that is collected by a service monitor can be disseminated to other monitors through a publish/subscribe based communication scheme which allows messages to be sent via a push or pull data exchange. A domain monitor (residing in the medium tier) is responsible for managing the whole domain or parts of it. Domain monitors can establish a domain-wide perspective of malicious activity (or patterns of activity) and are additionally responsible for reconfiguring system parameters, interfacing with higher level monitors and reporting threats

against the domain to administrators. Enterprise monitors enable a global abstraction of the cooperative community of domains. They correlate actions reported by domain monitors and focus on network-wide threats such as spreading worms or large scale network scans. It is important that the enterprise itself needs not to be stable in its configuration or centrally administered, but only has to consist of stable domains. The system's ability to perform inter-domain event analysis is necessary to address global attacks against the entire enterprise or even multiple companies.

The generic Emerald monitor architecture allows components to perform signature analysis and statistical profiling to provide complementary forms of analysis of the operation of network services and infrastructure. It is designed to enable the flexible introduction and deletion of analysis engines from the monitor as necessary. The basis for the analysis engine is provided by a target-specific event stream that can be derived from a variety of sources (e.g. audit data, network data, `SNMP`). The analysis engine itself can implement arbitrary intrusion detection algorithms and Emerald provides two packages that incorporate misuse based (signature) and anomaly based (profile based) variants. A resolver unit that implements a response policy based on intrusion summaries produced by the local analysis engines or other monitors may also be present. The resolver is an expert system that receives the intrusion and suspicion reports produced by the profiler and signature engines and invokes the various response handlers based on their results.

Emerald's signature-analysis strategy departs from previous centralized rule-based efforts by employing a highly distributed analysis strategy that, with respect to signature analysis, effectively modularizes and distributes the rule-base and inference engine into smaller and more focused signature engines. The scope is narrowed and focused on only a small set of attacks, thereby reducing the noise and the generation of wrong detects (false positives). Instead of keeping a global knowledge-base that contains a representations of all known malicious activity, Emerald distributes a tailored set of signature activity with each monitor's resource object. The objectives of signature analysis depend on the layer in Emerald's hierarchical scheme where the detection is performed. At a low layer (service tier), the engine scans for known exploits against single services, on higher tiers, coordinated attempts are detected. The signature engine is based on the expert system shell `P-BEST` (for an example application - see [61]). This approach suffers from the problem that a complete description of the enterprise network together with all network services must be available. This is necessary to move the correct signatures to the places where they are needed. As this is usually not provided and changes are frequent, Emerald actually needs all signatures present at each host.

According to [65], a significant amount of software engineering effort has been invested into the system to keep it as modularized as possible. The layered, hierarchical approach allows coordinated detection and communication between components. The creators claim to have obtained very promising results with the system. Unfortunately, little details have been provided about the implementation of the publish/subscribe infrastructure. This approach usually requires a communication backbone that is responsible for managing subscriptions and message routing. As both publishers and subscribers change very frequently, this backbone might become a bottleneck and is vulnerable to direct attacks (for

example, denial-of-service which prevents the forwarding of messages). While the misuse and anomaly sensors are arguably one of the best currently available, the hierarchical correlation framework is rather undocumented.

3.2.3 AAfID

In [9], the Architecture for Intrusion Detection using Autonomous Agents (AAfID) is presented. A hierarchical design has been chosen to combat the disadvantages of a central and monolithic architecture. The goals of this design are the improvements of the system configurability and of the resistance against denial-of-service attacks.

It consists of three layers each using the direct subsystem below. At the base level, so called agents perform the security monitoring task. These are independently running entities that scan for suspicious actions on each host by analyzing data which they directly get from the network or from audit trails. Agents have the ability to communicate with other agents that run on the same host and can evolve over time (using genetic programming) to adapt to new situations. In an earlier paper [24] a method for training agents is presented. This is realized by confronting agents with raw network packets and providing the information whether they can be classified as malicious or not. The gained 'experience' is then saved in a kind of parse tree, which is later consulted during regular operation. If agents detect a possible intrusion, they have the ability to inform other agents of their perceptions. The information collected by agents is forwarded to an entity called transceiver, which exists only once on each host.

Transceivers channel the information on every machine, control and configure agents and may analyze and forward the collected information (possibly in a reduced representation) to entities called monitors. Monitors represent the highest level in the proposed hierarchy and can collect and evaluate data from more than one transceiver. As a result, they operate on data which is provided by multiple hosts. Monitors can be organized in a hierarchical fashion by having a root monitor (which is used for human interaction) controlling other monitors. The user interface, which is considered as a crucial part in the AAfID architecture, provides the user with the ability to command and configure the whole IDS.

A key point for the group working on the AAfID is the inter-host communication which was designed to be performant, reliable and secure. It was evaluated whether it is reasonable to use an existing protocol (e.g. `TCP`, `UDP`) or to design a new one which has more of the desired properties. Unfortunately, no general solution could be given, as it was concluded that the decision depends heavily on the application which uses the communication system.

Although an advanced communication subsystem is introduced that operates even in case of ongoing denial-of-service attacks, the basic system structure is rather simple. While agents may communicate locally with each other and agree upon a common suspicion level at every node, no special pre-filtering of information is performed. All possible relevant data is simply forwarded to monitors (via transceivers) where human interaction is necessary to detect highly distributed intrusions. The system created a high degree of interest

in the research community when first introduced, but except the use of the term agent for monitoring processes and the influence of machine learning techniques on the detection process, no novelty can be identified. The system design not only suffers from the vulnerability of a central monitor, but it also requires human interaction at the highest level. This might decisively delay the response to an attack and fail to prevent damage where an automated system might have succeeded.

3.3 Alternative Correlation Approaches

The discussion in the previous sections underlines that the need for distributed event correlation for intrusion detection has been early recognized. Nevertheless, the involved problems have remained the same from the first suggested prototype to the currently most advanced systems.

Distributed event collection creates a massive amount of data that must be searched under stringent real-time constraints. Because the pieces of evidence of an attack may be scattered over arbitrary locations, the processing cannot be easily parallelized. Two possible solutions have been proposed and implemented that both have their shortcomings.

On one hand, massive pre-filtering at the sensors itself makes the data stream that is transferred to a central analyzer manageable. This has the limitation that it is often impossible to decide locally whether information is relevant for detection or not. Systems following this approach can either forward all possible interesting data or take the risk of dropping needed events. In the first case, the original problem is still unsolved, in the latter, the system can miss attacks. This is clearly undesirable.

On the other hand, recent designs try to distribute the correlation to certain dedicated intermediate nodes that see all relevant data for a small and manageable area. Only data which is considered to be important for the whole protection domain is forwarded to a central node which is responsible for the complete installation. Although this approach moves the pre-filtering decisions to a higher level, the basic shortcoming is still present. While each area is completely monitored by an intrusion detection system, the global correlation is either overloaded or cut off from relevant data.

In addition, both approaches are vulnerable to faults or deliberate attacks against the processing infrastructure. When a central or high-level node fails, the system is blinded. This is also true for publish/subscribe variants as implemented in Emerald (see Section 3.2.2). The message routing hosts that manage subscriptions and set up the paths for published messages can also be attacked.

In order to solve the inherent difficulties with centralized data gathering and dedicated processing nodes, it is interesting to investigate the possibility of decentralized data analysis. Although it is difficult to identify independent chunks and analyze them in parallel, a distributed detection algorithm might still be feasible and beneficial. Two different variants are discussed. One suggests that only the nodes where the intrusion is actually witnessed cooperate on its detection. The other approach is based on mobile code where agents roam the network and carry the relevant data with them. The next subsection describes

an approach that realizes the first variant while the following two subsections deal with mobile agent based designs.

3.3.1 Cooperating Security Managers

In contrast to hierarchical or centralized IDSs, the Cooperating Security Manager (CSM) design [105] tries to eliminate the need for dedicated elements by introducing an interesting new architecture. Instead of having a central monitor station to which all data has to be forwarded there exist independent uniform working entities at each host (called security managers) which all perform similar basic operations. In order to be able to detect distributed attacks, the different monitors have to coordinate their intrusion detection activities and cooperate (hence the name of the system) as described below.

Each local security manager consists of a CMDMON (command monitor) and a CSM module. The CMDMON intercepts and records user commands and attempts to identify single suspicious actions. The system performs misuse based detection and searches for specific pre-defined user actions that are considered security sensitive (e.g. password file changes using nonstandard means such as `vi /etc/passwd`).

When a malicious signature is found, it is communicated to the local CSM unit. The distributed detection process is based on the tracking of users. A *trail* is a data structure that stores the information on which host a user originally performed his login and how he continued to connect to different machines from there. Along each trail, all suspicious activities are aggregated and an alarm is raised when too many of those operations have occurred (even when they happened on different hosts). The number of suspicious operations are tracked utilizing the notion of a *suspicion level*. Each detection of a sensitive operation results in an increase of the current suspicion level. It is assumed that a user with a longer trail will have done more sensitive actions so the length of the trail is considered when calculating the suspicion level. The design assumes that although a skilled intruder will perform only a small amount of malicious activities on each host, in summary there will be many. By using the trail data, an overall suspicion level for each user can be determined.

Whenever an interesting action is noticed, it is communicated to others managers. It is important to notice that data is not broadcasted but only sent to the CSMs along the trail of the user. The CSM where the action is recognized sends the information only to its upstream neighbor (previous host in the current trail) which then continues to forward the data to its own upstream CSM. At the CSM where the user has initially logged in, the final suspicion level can be calculated.

When a certain limit is exceeded an intrusion is assumed. In this case, an intruder handling component decides how to proceed. The following two possibilities are implemented.

- *Limit Damage* - This passive action attempts to limit the damage when an intruder has already taken over a host. CSM tries to backup critical files in order to minimize the impact.

- *Active Defense* - A more aggressive defense is invoked when network critical machines are endangered. In this case, the active suspicious session is interrupted and the corresponding account locked.

The presented fully distributed architecture has the advantage that no single point of failure or bottlenecks are inherent in its design. CSM is intended to scale well and can be used in larger environments. One problem of the implementation of the described approach is its operating system dependency. The command monitor and the local IDS differ significantly in the `UNIX` and `Windows` implementations. Nevertheless, such drawbacks could be solved by an improved design of the components and are not a consequence of the distributed approach. A more severe shortcoming of the presented system is the fact that only activities of a single user are correlated. When a number of different accounts are compromised, cooperating attackers might remain undetected. Additionally, only threat scenarios that require a user to be logged in and attack the network from inside can be covered. Intrusions from outside that target inside network services will remain unnoticed.

3.3.2 Micael

Micael [27] suggests an approach for intrusion detection with distributed and decision making agents. Autonomous agents investigate possible intrusions and are capable of initiating counter-measures against attackers.

The architecture defines different kinds of static and mobile agents. *Sentinels* are static agents (i.e. sensors) that reside at every protected host. They collect data (using `SNMP`) about host activity and have no specific knowledge about different classes of attacks, although they are able to protect themselves against simple denial-of-service attacks. When anomalies are detected, reports are sent to a *headquarter agent*.

Headquarters are centralized agents, which get data about possible intrusions from sentinels. They relate the information of different sentinels and create statistics, but they do not do any detection by themselves. In contrast to hierarchical systems, where such central points are static, headquarters can dynamically move and thereby perform load balancing and evade potential attacks. When anomalies reach a certain level of suspicion, *detachment agents* are created and sent to interesting hosts.

Detachments are mobile agents that are used to face a possible intrusion. They move to positions where a potential attack has been detected and start detailed analysis of log files. Additionally, they can initiate counter-measures against the attacker (ranging from a simple disconnect operation to an active counterattack).

Micael has been designed with possible attacks against the IDS itself in mind. So called *auditor agents* are deployed to monitor the system's integrity. They are occasionally launched by the headquarter and check other agents by comparing pre-calculated hash values to results of a special function that each agent has to support. When an agent has been corrupted, it is forced to terminate. If the auditor cannot find the headquarter, it assumes that it has been destroyed and creates a new one.

Micael uses the Aglet Mobile Agent System [2] and is written in `Java` to assure a

high degree of portability. The communication between agents is realized using the Aglet Transfer Protocol (ATP) where agent proxies (which act as forwarding references to the actual agents) and relays allow inter-agent communication, independent of the machine where the agents are currently executed.

The idea of distributing the detection procedure into different, mobile system parts has the advantage of keeping the whole system's load relatively modest. Mobile agents offer the advantage of consuming the resources only at the place where they run. Additionally, they are also able to react very quickly when an intrusion has been discovered. Unfortunately, the authors have only presented a very high-level design without any details regarding the implementation or any measurements. So it is difficult to evaluate how effective this approach actually is.

3.3.3 Sparta

Sparta [52, 49] (which is an acronym for Security Policy Adaptation Reinforced Through Agents) is the name of a project sponsored by the European Union. It is a system whose primary aim is to detect security violations in a heterogenous, networked environment. Nevertheless, the architecture which has been designed for Sparta allows a broader range of application, ranging from network management to intrusion detection.

Sparta is an architectural framework which helps to identify and relate interesting events that may occur at different hosts of a network. In addition to the detection of interesting patterns, Sparta can also be utilized to collect statistical data (i.e. extreme value or sum of attribute values) of certain events.

Each host has at least a local event generator, a storage component and the mobile agent platform installed. The local event generation is done by sensors which monitor interesting occurrences on the network or at the host itself. The exact types of events and their attributes are determined by the application's needs. Events are stored in the local databases for later retrieval by agents. The mobile agent subsystem is responsible for providing a communication system to move the state and the code of agents between different hosts and for providing an execution environment for them. Additionally, the system has to provide protection against security risks involved when utilizing mobile code. Most of the components are written in `Java` and the agent platform itself rests on Gypsy [45], a `Java` based system which has been developed at Technical University Vienna.

The goal of Sparta is the design of a mobile-agent based IDS that identifies and improves potential shortcomings of other intrusion detection system designs. The following three issues are addressed.

To support the detection algorithm and to address the problem of systems which only offer an implicit way of specifying attack scenarios, an attack pattern language was designed. This language allows the expression of offending event constellations in a declarative manner where one can specify what to detect instead of how to detect. The primary language design objective is the reduction of the needed amount of transferred data while still retaining enough expressiveness to be usable for most situations. When a system uses mobile code (i.e. mobile agents), it should aim at performing flexible computation remotely at the

location where the interesting data is stored. This resulted in the limitation of restricting patterns to tree-like structures when events between nodes are involved but still allows arbitrary correlations locally.

The correlation mechanism that identifies these tree-like patterns does not rely on (one or more) central server locations where data is gathered and related. Instead, it follows a fully distributed approach. Mobile agents roam the network to search for suspicious events to start a more detailed investigation. When an agent spots the mark of a possible intrusion, it decides which data to carry with it on its next hop and which place to visit next. The basic idea is that agents work their way recursively up the tree from the root to the leaves of the pattern. When evidence for all events specified in the leaves of the pattern are found, an intrusion has been detected. The detection algorithm itself is performed by multiple agents in parallel which improves scalability, fault tolerance and performance of the system when compared to a client-server variant.

While many agent systems use some way of encryption and authentication when agents are sent over the network, most of them lack a pubic key infrastructure (PKI). This issue is addressed in Sparta by providing a PKI to manage the cryptographic framework with user and agent permissions. Sparta utilizes an asymmetric (public key) crypto system to secure agents when they are transferred over the network. To protect the platforms, agent code is signed and authenticated before it is executed.

Sparta and its specification language has been developed at the Technical University Vienna and therefore has clearly influenced the design of the distributed event correlation presented in this dissertation. However, we identified the use of mobile code as unnecessary and counterproductive in the given setup. The agents are solely used as data containers, a task that can be fulfilled more efficiently by sending the information encapsulated in messages and deploy the processing modules at all nodes in advance. Mobile code introduces additional security risks and causes a performance penalty without providing any clear advantages. Although the approach used in Sparta heads in the right direction, message based communication and minor modifications to the specification language improve the robustness, scalability and performance of the whole system. The resulting design and its implementation is described in detail in Chapter 4 and Chapter 5.

3.4 Host Anomaly Detection

The presented concept of *Network Alertness* is based on the analysis of network traffic and transmitted packets. Although a number of network based anomaly sensors have been suggested by the research community, most of the work has been done on host based designs. As the first intrusion detection systems ever built used that approach, this section presents a brief overview of host anomaly detectors. The following section then reviews the (for our perspective more interesting) work on network anomaly analyzers.

3.4.1 User Behavior

In the early stage of IDS design, most researchers attempted to build systems that compared pre-recorded user profiles to the actual witnessed behavior. Anderson [5] and Denning [29] used metrics such as the time of day when a user logged in and the length of active sessions. They were later refined to take into account the sequence of commands a user types or the frequency with which he accesses certain resources. NADIR [39] as well as Emerald (Section 3.2.2) along with its predecessors IDES and NIDES [63, 4] incorporate modules that do such analysis based on statistical tests. A variant using neural networks is presented in [35].

More recent work [84] attempts to profile users based on keystroke characteristics. This approach assumes that each user exhibits a distinguishable pattern when typing on a keyboard. One can measure the time between certain keystrokes but also the time between pressing a key and releasing it[1].

Although analyzing user behavior seems to be a natural way of performing intrusion detection, it is very inaccurate. Users are humans and usually tend to act without following strict patterns. They may change their behavior simply because they install and use new programs, get a new job assignment or switch their working hours due to private reasons. In order to prevent a huge amount of false alarms, such systems have to constantly adapt to the user behavior and use very high suspicion thresholds before an alarm is raised. An attacker may easily exploit this by slowly training the system to consider his malicious activity as normal. Additionally, such systems cannot react properly when network services get compromised as no single user profile can be associated to a daemon program.

3.4.2 Program Behavior

Instead of profiling users, an alternative variant is to turn to program execution. Although a program may be utilized by many different clients, it acts in a regular and predictive manner. When a service daemon is compromised, one can expect that it is possible to monitor changes in its behavior.

Two variants have been suggested to capture a program's behavior. One concentrates on the sequence of system calls that running processes issue during their execution while the other checks the abnormality of the input and output data.

The analysis of system call sequences has been first proposed by Forrest [34]. It is done by tracing the system calls of interesting processes and try to predict the next system call with a certain probability. The idea is that after a good learning phase, it should be possible to do 'not too bad' prediction of the next call given the past calling sequence. If too many predictions fail to be correct, then an alarm is raised. The system cuts the audit trail of system calls into fixed-length sequences by a sliding window mechanism. These sequences are then stored in a database using a tree data structure where each path from the root to a leaf represents a sequence that has been seen. In addition, a label at each leaf is added which represents the probability of occurrence of this call given that the past

[1] Keystroke analysis is also used in an attack against `SSH` to retrieve password information [88].

calls have been monitored. A simple tree database for a sequence of 4 system calls with a sliding window of length 2 is presented in Figure 3.4. An open source implementation of this approach is provided as well [32].

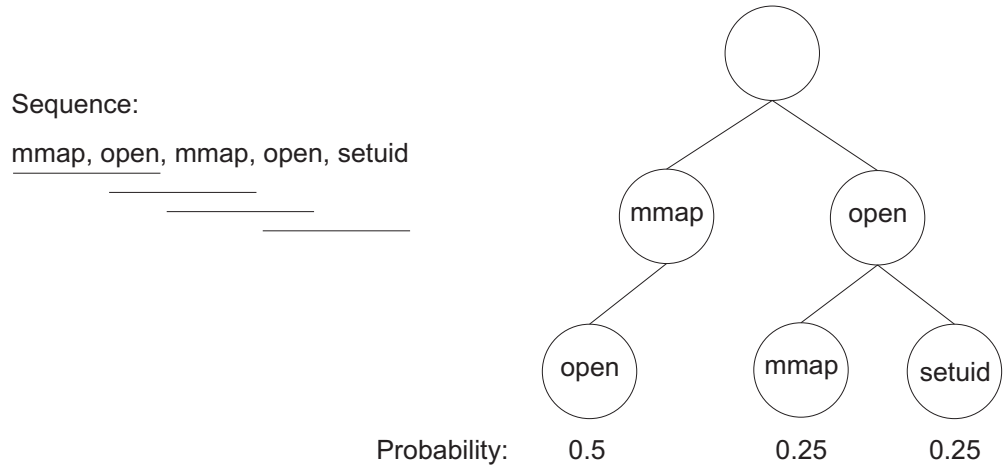

Figure 3.4: System Call Monitoring

The analysis of input data using artificial neural networks (ANNs) has been suggested by Ghosh [37]. Neural networks offer, similar to statistical tests, the capability to generalize from incomplete data and to be able to classify on-line data as being normal or intrusive.

An artificial neural network is composed of simple processing units (nodes) and connections between them. The connection between any two units has some weight which is used to determine how much one unit will affect the other. A subset of the units of the network acts as input nodes, and another subset acts as output nodes. By assigning a value (called activation) to each input node and allowing the activations to propagate through the network, a neural network performs a functional mapping from one set of values (assigned to the input nodes) to another set of values (retrieved from the output nodes). The mapping itself is stored in the weights of the network. For this system, a back-propagation neural network has been utilized. During a training phase, the network has been exposed to classified input data (data known to be normal or non-normal). It gradually adjusts itself to be able to classify the given training set. The resulting network is then used to check and classify new process input and output data and raise an alarm when too many non-normal instances are witnessed.

3.5 Network Anomaly Detection

Network Alertness aims at protecting important network services by intensifying the analysis of (i.e. adapting to) input from certain suspicious sources. Although host based anomaly detectors can monitor the behavior of known users and executing processes, they are not suitable for this task. When services are attacked by unknown outsiders the analysis of inside users is rendered useless. The problem with process behavior is the fact that

it is hard to account system call sequences (or similar properties) to network sessions. Our approach demands that a sensor is capable of becoming more sensitive to data exchanged in a certain network session. When a daemon serves many requests simultaneously, it is possible to monitor the result of a successful attack but it is hard to tell the origin of the intrusion attempt. In this case, a sensor has to evaluate the process properties without being able to adapt its suspicion thresholds. Therefore, we use network anomalies for our analysis, as individual sessions can be easily distinguished at this level.

Network anomaly detectors focus on the packets that are sent over the network. Depending on the type of information that is used for performing the detection, one can distinguish between *traffic* and *application* models.

Systems that use traffic models monitor the flow of packets. The source and destination `IP` addresses and `TCP/UDP` ports are used to determine parameters such as the number of total connection arrivals in a certain period of time, the inter-arrival time between packets or the number of packets to/from a certain machine. Such parameters can be used to detect port scans or denial-of-service attempts. Most current network based systems rely on traffic models to perform the bulk of their anomaly detection. The next sections introduce two current systems and their analysis algorithms.

Unfortunately, traffic models aggregate over packets from many sessions to build a profile that describes quantitative properties of network connections. For our purpose, a model that incorporates qualitative information (i.e. the content of a single packet) is needed. The system must be able to detect intrusions targeted at a single process. Such attacks usually exploit a vulnerability of a service at the victim machine. This is done by sending invalid input which causes a buffer overflow or an input validation error in the code running the service. The attacker sends one (or a few) carefully crafted packets including shell-code which is executed at the remote machine on behalf of the intruder to elevate his privileges. As the attacker only has to send very few packets (most of the time a single one is sufficient) it is nearly impossible for systems that use traffic models to detect such anomalies.

To circumvent this drawback, application models attempt to utilize application specific knowledge and deepen their detection for single packets. Unfortunately, these designs are currently very simple and include mainly additional `TCP` header information or count the number of bytes that are exchanged in a session between client and server. The last section surveys work done in this area and highlights the improvements necessary to implement a *network alert* system. The details of our application model anomaly sensor are presented in Chapter 7.

3.5.1 Adaptive Bayesian Model

This section describes a traffic model [74, 99] that is based on Bayesian inference in trees. It has been implemented as part of the Emerald project (see Section 3.2.2) and extends the systems capabilities from evaluating user session to network anomaly detection.

The analysis is based on sessions which are defined as temporally contiguous traffic from a particular `IP` address. Whenever an interesting event is detected in the input

stream, it is either accounted to an already existing session or a new one is instantiated. Events themselves are created by a network monitor that analyzes `TCP` headers and extracts connection state changes from all currently active `TCP` sessions. It is not very important for the system to demarcate sessions exactly. The Bayesian inference analysis is done at periodic intervals in a session (where the interval is measured in number of events) or when the system believes that the session has ended.

The Bayesian inference is based on belief propagation in causal trees [71]. Knowledge is represented as nodes in a tree, where each node is considered to be in one of several discrete states. A node receives prior (or causal) support messages from its parent and likelihood (or diagnostic) support messages from its children as events are observed. The prior messages can be considered as propagating downward through the tree while the likelihood messages are propagating upward. The prior message incorporates all information not observed at the node and the likelihood at terminal (i.e. leaf) nodes corresponds to the directly observable evidence. A conditional probability table (CPT) links a child to its parent. According to the Bayesian theorem, the CPT lists the probability for each child by taking into account the a-priori knowledge of the probability of the parent.

Figure 3.5 shows an example of a simple Bayesian tree. It represents the a-priori belief that the probability of many open connections (0.8) and different unique ports (0.7) will be high while the probability for different `IP` address will be low (0.3) when a port scan is observed. This is due to the fact that a single attacker scans many ports of a single victim machine in parallel. When the three evidence parameters are determined during system run-time, the probability for the attack class (i.e. port scan) can be calculated.

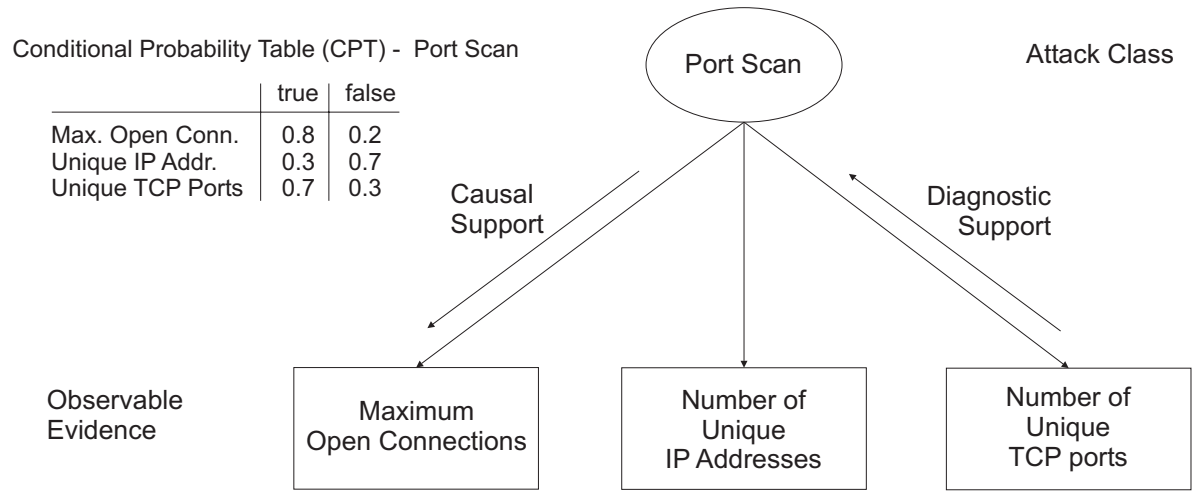

Figure 3.5: Bayesian Inference Tree

The task of the inference process is to 'quantify the support for each hypothesis class given the observed events'. That means that the inference engine determines for a session the a-posteriori probability that it belongs to a certain class given the a-priori knowledge stored in the Bayesian tree and the actual observations. This is used to classify a session,

for example, as being normal, a denial-of-service attack, a port scan or an event caused by a non-malicious misconfiguration. The used model has the advantage that it does not require classified training data, in fact, it automatically classifies traffic without a prior learning phase. In addition, the probability tables evolve and the system improves its accuracy when used.

The presented system uses a sophisticated statistical model to classify network sessions. The only shortcoming, which unfortunately makes it unsuitable for our use, is the fact that no payload is inspected.

3.5.2 Spice

Spice [92] is an acronym for Stealthy Portscan and Intrusion Correlation Engine which has been developed by the creator of GrIDS (see Section 3.2.1). Spice is interesting mainly because of one of its its components that assigns an anomaly score to every single packet (called Spade - Statistical Packet Anomaly Detection Engine). This aims into a direction where each packet can be rated individually in contrast to considering complete bursts of traffic (or whole sessions).

The anomaly score that is assigned is based on the observed history of the network. The fewer times a particular kind of packet has occurred in the past the higher its anomaly score will be. Packets are classified by their joint occurrence of packet field values. To do this, a probability table is maintained that reflects the occurrences of different kinds of packets in history - with higher weight on more recent events. Unfortunately, Spade analyzes only certain parts of the packet header, but completely neglects the packet contents.

All packets that receive a higher anomaly value than a certain defined threshold are forwarded to Spice. Currently, Spice is designed to detect port scans. This is realized by a correlation engine which attempts to cluster packets together that it assumes to be part of the same scan. Combining suspicious packets helps to reduce the needed storage space and allows keeping the data longer. This makes it possible to detect long lasting scans that are performed very slowly by an attacker. When the available storage is used up, all systems need a policy to decide which information should be discarded. A common decision is to throw away the oldest data. By aggregating data of packets that belong together and transform the information into a simpler cluster data structure, old samples that would have been timed out when considered individually might survive and add up to a suspicious scan pattern.

The clustering algorithm itself is explained by using a metaphor from physics[2]. Packets behave like atoms in an empty space and are linked to each other by a certain bonding energy. When the bonding energy between two packets is strong enough they get coalesced into a single bonding graph that can attract other packet 'atoms'. The bonding energy itself is expressed (and calculated) via heuristics that compare certain packet features and timing properties.

[2]The author holds a Ph.D. in physics.

3.5.3 Application Based Analysis

The difference between traffic based and application models is introduced in [14]. Although this work only focuses on traffic modeling it is the first work that notices the possibility to extend network based intrusion detection to the application level.

This integration of application knowledge into a statistical detection approach is described by [13]. The authors have chosen the `telnet` protocol as the basis for their analysis. Unfortunately, the proposed evaluation of the application data itself is very simple. It turns out that most of their statistical properties are affected by flags in the `IP` and `TCP` headers. The system monitors the total number of `TCP` packets from a certain source and the ratio of this sum to the number of packets with a set `RST` flag. This value determines the number of times a connection is reset compared to the total amount of packets that are transferred. Similar to traffic models, this helps to detect denial-of-service attacks. The only feature that is monitored in the application payload is a string which indicates a denied login attempt. When the number of wrong authentication attempts exceeds a certain threshold an alarm is raised. This number is also combined with the total number of packets to adapt the threshold to the actual number of connection attempts (obviously allowing higher number of failed tries when the number of attempts is high). Although information retrieved from the payload is combined with a stochastic approach, the detection process itself is misuse based. The system performs simple pattern matching to determine failed logins. This clearly does not allow adaptation of the sensor's alert threshold with regard to suspicious sources, a requirement needed for implementing *Network Alertness*. In order to provide an application based anomaly sensor, we designed and implemented our own variant which is currently focused on the analysis of `DNS` traffic. It is discussed in detail in Chapter 7.

3.6 Summary

This chapter has introduced work in two areas which are relevant to the system that we propose to perform intrusion detection.

The first one introduces different approaches to correlate event information which is distributed over many sensors throughout a network. As we propose a novel mechanism to increase the scalability and fault tolerance of this correlation process, an evaluation of the current designs was necessary. Three concepts could be identified that followed either a centralized, a hierarchical or a completely distributed paradigm.

Centralized systems are straightforward as they simply collect data at a central node and perform the computation there. Such approaches have been the first which were suggested by the research community and exhibit the poorest scalability and fault tolerance properties. To mitigate their weaknesses, hierarchical approaches with several layers of processing nodes have been introduced. Although they behave in a more desirable way, they still rely on a few dedicated nodes for detection. This lead to the development of alternative variants, where the processing load is completely distributed over all machines.

While such a design has promising characteristics, current systems have been very simple and suffered from several restrictions.

A central target of our system, that is presented in this dissertation, is its potential to be utilized for large network installations. This requires a completely distributed design where no predefined weak spots (such as central servers or hierarchy nodes) exist. The next two chapters detail our proposed mechanisms that exploit the advantages of a completely distributed design but overcome the limitations that are present in current systems.

But *Network Alertness* is not limited to the detection of attack scenarios alone. It also requires the adaptation of sensors to potentially malicious sources when emerging patterns are identified. This made it necessary to review the work in a second area, the domain of intrusion detection systems with the potential to adjust their detection processing to different input streams. Because the probes have to adapt to data from varying sources, network based systems, that can easily distinguish between different connections at the network layer, are most desirable. Unfortunately, current designs aggregate over network sessions from many origins for their detection. Therefore, we had to design a network based sensor that is capable of adjusting its detection threshold to data from only a single source. Our design is presented in Chapter 7. This sensor is realized as a component that can be plugged into our correlation framework and enables cooperating nodes to tune their detection to potentially malicious senders.

Chapter 4

Distributed Correlation Framework

Divide et Impera!

– Roman Proverb

We have already underlined the necessity for collecting and relating audit data that occurs at different, distributed nodes in a network. In this chapter, we present a framework [50] for completely distributed intrusion detection which restricts the detection to only the affected nodes. The framework consists of a specification language that is needed to describe attack scenarios and a decentralized search algorithm that relates events from different sources to detect them.

By relating events from multiple nodes, one can detect a number of attacks that would remain unnoticed by only focusing on local activity. It might also increase the chances for anomaly detection sensors to catch an attack when they can adapt to specific input sources. The previous chapter introduced centralized and hierarchical designs that attempt to solve this problem. Nevertheless, these systems use dedicated nodes that act as central points for collecting data from remote sensors. Therefore, such structures are still vulnerable to faults and overloading of nodes - especially those that are close to the root of the hierarchy or the single central analyzer.

We solve the inherent problems of centralized, dedicated nodes by proposing a completely decentralized approach where the detection of an intrusion is restricted to those nodes where parts of the attack are directly observable. This approach is influenced by the work done on alternative correlation variants which are described in Section 3.3.

Our design is related to CSM (Cooperating Security Managers - see Section 3.3.1) in the way that it distinguishes between a local ID component and an information forwarding unit at each node. In contrast to this system however, we do not wish to limit the information exchange between nodes to the login chain of users. Sparta (Security Policy Adaptation Reinforced Through Agents) [52] follows a similar, distributed correlation mechanism as described in this dissertation, but is based on mobile agents.

The following sections describe an algorithm that is capable of detecting patterns of events that occur at multiple nodes of a network in a completely decentralized fashion.

In addition, the specification language to define distributed patterns is explained. This language allows the definition of patterns that are more complex than those presented in the related work in the previous chapter, but can nevertheless be detected without any dedicated coordination instances.

In our system, an intrusion is defined as a pattern of basic events which can occur at multiple hosts. A basic event is characterized as the occurrence of something of interest that could be the sign of an intrusion (e.g. the receipt of a certain `IP` packet, a failed authentication or a password file access). Such events could either stem from a local misuse or an anomaly incident. We also identified the requirement to integrate third-party systems to perform the local detection and feed their event data into our correlation algorithm. Our distributed patterns and the detection algorithm can describe and detect situations where a sequence of events occurs on multiple hosts. This is needed to model a scenario similar to the one presented as our motivating example in Section 1.1 where a port scan is followed by abnormal packets. Relating events from different sources is also helpful to modify the reaction of sensors at nodes that become aware of emerging, hostile patterns. This results in a framework that enables nodes to develop a *network alert* behavior when confronted with threats form intruders.

The decentralized pattern detection process finds distributed patterns by sending messages between nodes where interesting events occur. Therefore, each node of the protected network has to run a process that executes the distributed pattern detection algorithm. The detailed description of the layout of such events and patterns as well as the detection algorithm forms the core of the chapter. The renunciation of dedicated central components and the effort of designing a fully distributed system results in favorable scalability and fault tolerance properties of our IDS. When a single node in our system fails (or is compromised), it stops its local detection and ceases to forward pattern information. This prevents the detection of pattern instances where events occur at the compromised host, but the rest of the system remains intact. In addition, messages are not sent to designated nodes but are exchanged between equal peers. This helps to distribute the complete message traffic over the network without some pre-defined central bottlenecks. We are aware of the fact that a distributed system design might result in tremendous message overhead. This potential danger is addressed at the levels of pattern specification and detection.

4.1 Pattern Specification

The design of our pattern specification language is guided by two conflicting goals. The first one demands a language that should be as expressive as possible. It would be desirable to allow the description of complex relationships between events on different hosts using regular or tree grammars. As our system relies on peer-to-peer message passing between hosts without a central coordination point, arbitrarily complex patterns might cause an explosion in the amount of data that needs to be exchanged. In the worst case, each node has to send all its data to every other node. This conflicts with the second goal, which demands that the amount of data that has to be transferred between hosts should be as

small as possible. Therefore we have to impose limitations on the expressiveness of our pattern language.

As stated above, an event is the smallest entity of a pattern and defined as the occurrence of something that might be part of an intrusion. We have designed a simple language called *Event Definition Language (EDL)* that allows us to specify an event as an object together with a set of attributes and their corresponding types (e.g. string, integer). In order to provide the system with EDL data, we have to install sensors that watch for the occurrence of interesting events and transform them into EDL objects by setting the given attributes to the actual values derived from the observed event instance. The actual implementation and the role of EDL for data collection and its transformation into data structures suitable for the detection process is explained in more detail in Section 5.1.1.

4.1.1 Definitions

A pattern describes activities on individual hosts as well as interactions between machines. The basic building block of a pattern is a sequence of basic events that happens locally on one machine (called *host sequence*). One can specify a list of events at a local host by enumerating them and imposing certain constraints on their attributes. We distinguish between constraints which relate single event attributes with constant values and constraints which relate different attributes of events using variables. One can use the standard logical operators for both types and an extended set of operators (including `in` and `range`) to relate attributes with constants. A connection (context) between event sequences on different hosts is established by *send events*.

Definition:

> *A pattern* P*, relating events that occur at n distinct hosts, consists of n sequences of events, one for each node (an event sequence at a single node is called host sequence).*
>
> *A set of events* S_A *at host* A *is linked to a set of events* S_B *at host* B*, iff* S_A *contains a send event to host* S_B*. Any event that refers to a remote host (e.g. the sending of a packet to a host, the reception of a packet from a host) might be used as a send event.*

It is only required for a send event that its target B can be determined locally at S_A from the event data.

Consider the case of a port scan that attempts to identify active `HTTP` ports on hosts of a target network. When the firewall detects such a scan **and** knows the address of the network's web server, the occurrence of that port scan can be utilized as a send event. In this case, the target host (i.e. the web server) is locally known to the node that identifies the suspicious behavior. Therefore, the firewall has all information available to inform the web server.

The first event of S_B has to be the next event to occur after the send event in S_A. It is required that the send event is the last event in S_A.

Definition:

Pattern P *is valid, iff the following properties hold.*

1. *Each set of events is at least linked to one other set.*
2. *Every set except one (called the root set) contains exactly one send event as the last event of the host sequence. The root set contains no send event.*
3. *The connection graph contains no cycles. The connection graph is built by considering each event set as a vertex and each link between two sets as an edge between the corresponding vertices.*

These definitions allow only tree-like pattern structures (i.e. the connection graph is a tree), where the node with the root set is the root of the tree. Although this restriction seems limiting at first glance, most desirable situations can be described. Usually, activity at a target host depends on events that have occurred earlier at several other hosts. This situation can be easily described by our tree patterns where connection links from those hosts end at the root set.

The case where events on two different nodes both depend on the occurrence of a single event at a third node cannot be directly expressed in our pattern language (as there would be two root sets). Nevertheless, a centralized application might split the original, illegal pattern into sub-patterns (each representing a legal tree-like structure) and relate the results itself.

4.1.2 Attack Specification Language

This section describes the syntax and semantics of our pattern description language called **Attack Specification Language (ASL)**. A pattern definition is written as follows

```
'ATTACK' "Scenario Name" '[' nodes ']' pattern
```

The *nodes* section is used to assign an identifier to each node that is later referred to in the pattern definition.

The *pattern* section specifies the pattern itself. It consists of a list of event sets, one for each node that appears in the node section. The event set, which represents the host sequence, is a list of identifiers, each describing an event. A pre-defined label called `send` is used to identify the target node of send events.

Each event can optionally be defined more precisely by constraints on the event's attribute values. These attribute values can be related to constant values or to variables by a number of operators (`=`, `!=`, `<`, `>`, `>=` and `<=`) or to constant values by a `range` or an `in` operator. The argument of `range` is a pair of values specifying the upper and lower bound of a valid range of values while the argument of `in` is a list that enumerates all possible values. More formally, these operators are defined below.

x range $(x_0, x_1) \leftrightarrow x_0 \leq x \leq x_1$
x in $(x_0, x_1, ..., x_n) \leftrightarrow \exists$i $(0 \leq$ i $\leq$ n) and x $= x_i$

A variable is defined the first time it is used. One must assign a value (bind an attribute value) to each defined variable exactly once while it may be used arbitrarily often as a right argument in constraint definitions. The scope of variables is global and its type is inherited from the defining attribute.

For each event, an optional response function can be specified. This function is invoked whenever the corresponding event description is fulfilled and it can take the values of already bound variables, event attributes or constants as arguments. A response function can be used to generate alerts for the system administrator or to perform active countermeasures against an intruder (e.g. hardening of firewall). As these functions are invoked locally at the node where the corresponding event description has been detected, our system possesses no central response component that can be taken out easily. In addition, response functions are used to create artificial event objects that are fed back into the detection process. An artificial event is not related to an actual activity in the environment but is created during a response by the system itself. It is piped back into the sensor's event input queue and can be utilized to satisfy constraints of different (or the same) pattern. Artificial events are a useful mechanism to exchange information between attack scenarios or to model timeouts. The output of a certain attack pattern can be used as input for another pattern to build hierarchical structures or to implement scenarios that count the number of times a certain basic pattern has occurred. A timeout can be implemented by starting a timer in a response function that then creates an artificial event when it expires.

4.1.3 Language Grammar

With these explanations, we introduce the syntax of the pattern section in BNF.

```
pattern       : {event set}+
event set     : node-id '{' {event}+ '}'
event         : ['send('target-id'):'] event-id '[' {constraint ';' }* ']'
constraint    : assignment | [label] relation [response]
assignment    : '$'variable-id '=' ( attribute | constant )
relation      : attribute operator ['('] {value ',' }* value [')']
value         : constant | '$'variable-id
attribute     : event-attr-id
operator      : '='|'!='|'<'|'>'|'>='|'<='| 'in'|'range'
response      : '<' function-id'(' arg-list ')' '>'
arg-list      : { arg-id ',' }* arg-id | e
node-id       |
target-id     |
event-id      |
variable-id   |
event-attr-id |
function-id   |
arg-id        : string
constant      : string | number
```

```
string        : ['a'-'z''A'-'Z''_''0'-'9']+
number        : digits | digits '.' digits
digits        : ['0'-'9'] | ['1'-'9'] ['0'-'9']+
```

The following example shows a classic distributed scenario, a login chain. It describes a common tactics of intruders to hide their tracks after they have compromised several machines in the network of the victim. In order to blur his tracks and hide his true origin, an attacker performs a couple of consecutive logins into different machines. All log files except one (the host where the hacker entered) show only connections from trusted machines. Different local time settings and audit file policies often make it difficult to trace back such a chain. Although the scenario itself does not describe an actual attack, such behavior is still suspicious enough to alert an administrator.

```
ATTACK "Telnet Chain" [ Node1, Node2 ]
Node1 {
  send(Node2): tcp_connect [ DstPort == 23; ]
}
Node2 {
  tcp_connect [ DstPort == 23; ] < alert(DstIP); >
}
```

The scenario above describes a telnet connection from `Node1` to port 23 at `Node2` and from there to port 23 of another remote machine. `Node2` describes the root set (i.e. has no outgoing send event). The target of the send event can easily be extracted as the destination `IP` address of the `tcp_connect` event attribute. This fact is specified in its EDL definition. The hacker continues from `Node2` to a third, remote machine. The `IP` address of that host (as `DstIP` attribute) can be extracted at `Node2` and is passed as an argument to the response function `alert` which notifies an administrator.

4.2 Pattern Detection

The purpose of the pattern detection process is to identify actual events that satisfy an attack scenario written in ASL. When a set of events fulfills the temporal and content based constraints of a scenario an alert is raised. Notice that instead of simply sending a message to a central system administration console (that yields again a single point of failure), more sophisticated responses can be implemented. The node itself can issue commands to reconfigure the firewall or to terminate offending network connections, thereby eliminating the single point of failure introduced by the central console of a human operator.

4.2.1 Basic Data Structures

In order to be able to process an attack description, it has to be translated from ASL into a data structure suitable for our system.

Pattern Graph

This is done by transforming a scenario into a directed, acyclic graph (called *pattern graph*). An attack scenario describes sequences of events located at different hosts that are connected by send events. Each single event specified by an ASL scenario is represented as a node of the resulting graph. The nodes of each host sequence are connected by directed edges. An edge leads from a node representing a certain event to the node which represents the immediate successor of that event in the ASL pattern description. Send events require a little different treatment as they are the last event in their host sequence and therefore do not have an immediate successor. In this case, a directed edge leads to the first node of the host sequence where the send event points to.

The resulting graph shows a tree shape and all paths through the graph end at the last event of the root set's sequence (called *root node*). Each node receives a unique identification number that consists of a part that identifies the attack scenario itself and a part that identifies each node within the scenario. The following example (see Figure 4.1 below) shows the result of such a transformation, which is actually straightforward as ASL only allows tree-shaped patterns. The attack scenario describes a pattern of a potential attack against a web server by a variant of the `CodeRed` worm. Similar to the hacker described in the introduction in Section 1.1, this virus does not only scan for an open port 80 but also attempts to retrieve the type of operating system the web server runs by asking the `DNS` server for the web server's `HINFO` (hardware and OS info) entry. This allows the virus to target `Microsoft IIS` servers more accurately. Whenever a port scan detector notices a scan against port 80 from a certain `IP` address and the `DNS` server gets `HINFO` queries from the same address and finally the web server receives an `HTTP` request from that source, an alarm is raised as such behavior is presumably suspect.

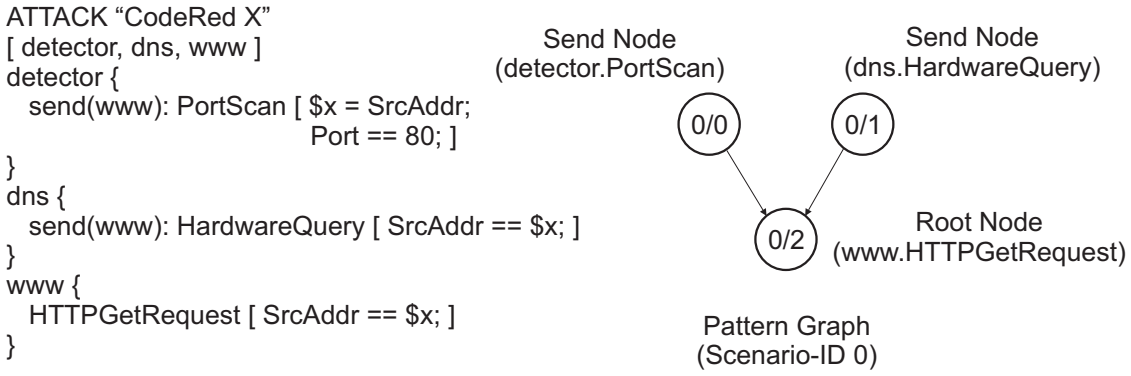

Figure 4.1: Pattern Graph Transformation

Messages

The detection algorithm does not deal with events itself, instead, it operates on messages. A message is a compact, more suitable representation of an event. Most attack descriptions

rely only on a small subset of the event's attributes for correlation (e.g. only `IP` addresses instead of the complete `IP` header). In ASL, only attributes that are assigned or compared to variables are of interest to the further detection process. Therefore, there is no need to operate on the complete event objects.

Obviously, a single event can match the description of multiple event patterns in an attack scenario. Thus, if more than one description is matched, several message instances (one for each matching pattern) are created. Whenever a message is created, all relevant attributes (i.e. the attributes that are assigned or compared to variables in the ASL description) are copied into it. Then, it is forwarded to the node representing the matching event description for further processing.

Each message can be written as a triple <id, timestamp, list of (attribute,value)>. The *id* of the message is set to the identification of the node of the pattern graph. The *timestamp* denotes the time of occurrence of the original event and the *attribute/value* list holds the values of the relevant event attributes which have been copied from the original event attributes. The id of a message defines its type. Different actual message instances with an identical id are considered to be of the same message type.

It is possible that messages of different types receive different attributes from a single event - depending on which ones are actually used in the attack description. In addition, the attribute/value list can be empty when the corresponding ASL event pattern does not reference any variables at all. In Figure 4.1, a port scan event that targets port 80 from `IP` address 128.131.0.1 would cause the creation of the message instance <0/0, time_of_occurrence, (SrcAddr, 128.131.0.1)> that has the type 0/0.

4.2.2 Constraints

An attack description in ASL imposes a number of different constraints on the events that must be taken into account by the detection algorithm. The set of constraints can be divided into a *temporal*, a *static* and a *dynamic* subset.

Temporal Constraints

The paths through the pattern graph reflect the temporal relationships of events. Event A has to happen before event B if and only if B is on the path which leads from A to the root node. The events of a host sequence have to occur in the same order as they are defined in the ASL description. When a host sequence is linked to another host sequence by a send event, all events of the destination sequence have to occur after the send event in the source sequence.

Static Constraints

An event pattern that relates an event attribute to a constant value imposes a static constraint onto events (e.g. the equality relation between the `Port` attribute and the value `80` in the `PortScan` event in Figure 4.1). Static constraints are easy to evaluate immediately

as soon as a new event of the appropriate type has been received. When an event satisfies all static constraints of a certain node (respectively its corresponding event pattern), a new message instance is created and forwarded to that node. Static constraints are used to decide which messages need to be created from a certain event but are not used later during the actual detection process.

Dynamic Constraints

A dynamic constraint is introduced by the use of variables in an attack description. The definition of a variable in an event pattern and the subsequent use of this variable in other event patterns introduces relationships between attributes of different events. Although it is possible to define and use the same variable within a single event pattern, such a variant can be trivially handled by the more general approach.

The definition of a variable by a certain event attribute and its subsequent use as an operand in a relation with another attribute creates a direct relation between these two attributes. In Figure 4.1 above, the definition of variable `x` as the the value of attribute `SrcAddr` in the `PortScan` event description and its use in the equality operations with the attributes of the `HardwareQuery` and `HTTPGetRequest` events create the following two dynamic constraints.

```
[PortScan.SrcAddr == HardwareQuery.SrcAddr ]
[PortScan.SrcAddr == HTTPGetRequest.SrcAddr]
```

Attributes that define or are related to variables are copied into messages. Therefore, it is possible to express the relationship between event attributes as (dynamic) constraints on the values of their corresponding message types. It cannot always be immediately determined whether an event satisfies its dynamic constraints, hence events that satisfy all static constraints of a certain event pattern cause a message to be created and passed to the appropriate node (the one which is associated with that event). It is the task of the actual detection process to resolve all dynamic and temporal constraints.

4.2.3 Detection Process

The basic detection process can be explained as follows. We have already stated that events cause messages to be forwarded to their corresponding nodes (to the nodes that are associated with matching events). The messages may then be moved along the directed edge of the graph to other nodes according to certain rules. The idea is that each node can be considered as the root of a subtree of the complete tree pattern. There are **node constraints** assigned to each node of the graph such that if there are messages which satisfy these node constraints, there are events that fulfill the dynamic and temporal constraints of the complete subtree above that node. Whenever the node constraints of a node are satisfied, certain messages may be moved one step closer to the root node, hence, they are pushed over the node's outgoing edge to its neighbor node below (as we have a tree-shaped graph, there is at most one outgoing edge for each node). Then, these messages are

processed at the destination node. This allows the process to successively satisfy subtrees of the complete pattern and move messages closer to the root node of the pattern graph. Whenever messages at the root node fulfill the constraints there, the pattern has been detected (i.e. there exist events that satisfies all constraints of the attack scenario).

The advantage of this approach is the fact that only local information (i.e. the set of node constraints) is necessary to decide which messages should be forwarded. This helps to actually distribute nodes of the pattern graph over several hosts and have each node make local decisions without a central coordination point. Different host sequences may potentially occur at different hosts.

Node Constraints

The node constraints have to make sure that all events described by the subtree pattern have occurred, that their temporal order is correct and that all dynamic constraints (which can be resolved up to this point) are met. The messages that are important for a certain node to satisfy its node constraints belong to one of the following three groups.

- Messages that are created from events that match the event description of the node itself (i.e. that have the same id as the node). Obviously, in order to satisfy a pattern, one event for each node of that pattern is needed. To fulfill a sub-pattern originating at a node, it is necessary to receive at least one message created from an event that matches the local event description itself (such a message is called a *local message* for that node).
- Messages that are created from events that match the event description of the node's immediate predecessors in the pattern graph. Usually each node has only one predecessor but this number can vary for the first node of each host sequence. Such nodes may have more than one predecessor or non at all.
- Messages whose value(s) are used in at least one dynamic constraint at that node.

The node constraints consist of

1. the set of temporal constraints between the local message and the predecessor nodes' local messages **and**
2. all dynamic constraints that can be resolved at this node.

The set of the temporal constraints between the local message and its predecessor messages guarantees that events described by the local node and by all its immediate predecessors have occurred. As messages from predecessor nodes may only be forwarded by them when the events at their predecessor nodes have occurred as well, it is assured that all events specified by a subtree pattern have taken place in the correct temporal order. The node constraints have to be modified for nodes without predecessors. For those, it is only necessary that the local message exists.

A dynamic constraint between attributes of two different events can be resolved as soon as both operands are available. When messages representing the two events are on-hand,

their relation can be evaluated and one can determine whether the dynamic constraint is satisfied or not. Therefore, every dynamic constraint (i.e. a variable definition at one node and its use at another one) is inserted into the pattern graph at the earliest node possible. The earliest possible node is determined by finding the first common node in the paths from each of the constraint operands to the pattern graph's root node. When one node is on the path of the other one, the constraint is inserted directly there, otherwise it is inserted at the node where both paths merge. A pattern graph with dynamic constraints is shown in Figure 4.2.

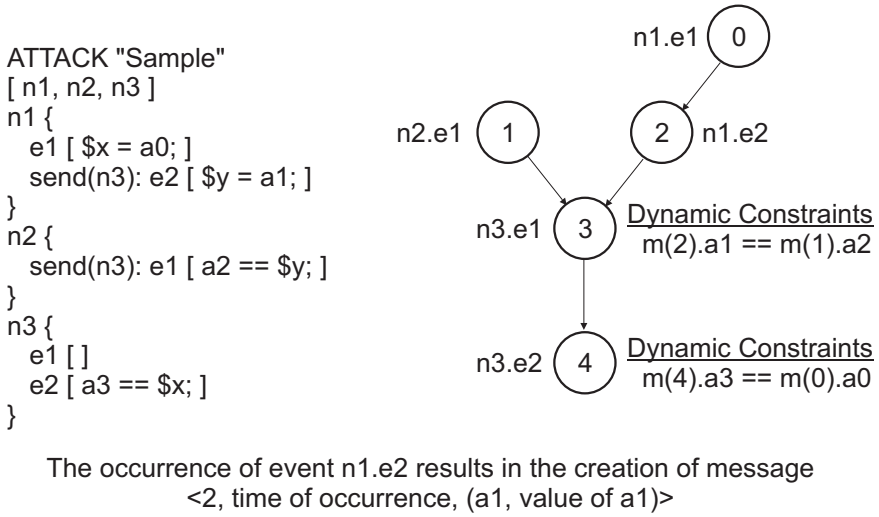

Figure 4.2: Complete Pattern Graph

A problem arises when transitive relations are introduced by relating a single message to several other messages. The attributes of events that are independent at first glance become linked by being related to a common, third event. In such a case, it is not enough to insert the constraints at the earliest possible node.

Consider the pattern graph in Figure 4.3 and suppose that the messages <0, t1, (a1,0)>, <0, t2, (a1,1)>, <1, t3, (a2,0)>, <1, t4, (a2,1)> and <3, t5, (a3,0)> are received in that order. The first four messages (the first two from node 0, the next two from node 1) are eventually passed to node 3 as the value of the first and the third message (which is 0) as well as the value of the second and fourth message (which is 1) are equal (dynamic constraint evaluated at node 2). As the attributes of messages with id 1 and 2 are not compared again at node 3, the value of the final (fifth) message is equal to the value of the first message and smaller than the value of the forth one. This results in an illegal report of a successful match.

To prevent this problem, all dynamic constraints that are connected by having attributes of common messages as their operands are combined in a subset of the scenario constraints called a *cluster*. When a dynamic constraint operates on messages that are used in no other dynamic constraints, the message itself becomes a cluster. In Figure 4.3 all three dynamic constraints are part of a single cluster.

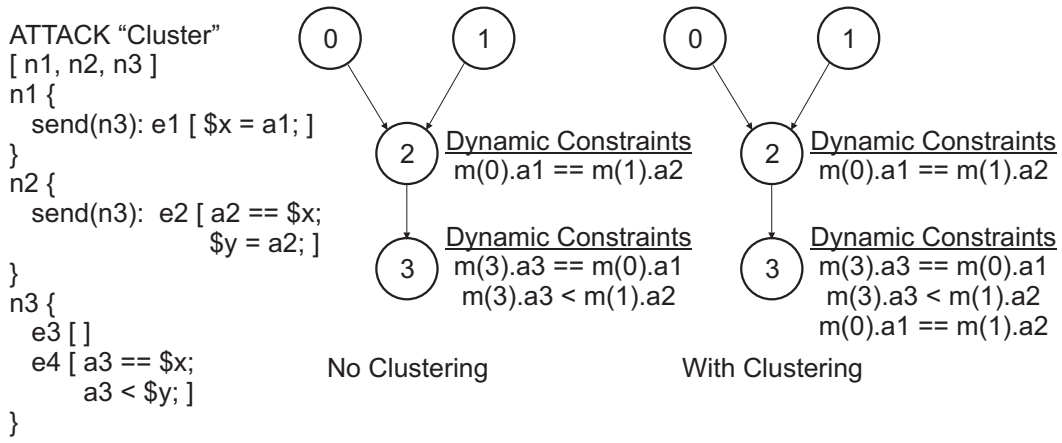

Figure 4.3: Constraint Clustering

In addition to the insertion of each constraint at the earliest possible node, all constraints of a cluster are additionally inserted at the cluster root node (but obviously, no duplicate constraints are inserted). Similar to the situation with a single constraint, the cluster root node is the first common node of all the paths that lead from each operand of every cluster constraint to the root node of the pattern graph. With these additional constraints, the example messages listed above do not result in a false detect.

Message and Bypass Pool

Each node has a *message pool* and a *bypass pool*. The message pool is a place that stores message instances that are needed to evaluate the local node constraints. The bypass pool holds message instances that can potentially satisfy node constraints of nodes that are closer to the root of the pattern graph (but which are not used for the current node constraints). Messages in the bypass pool are forwarded as soon as their temporal constraints are met.

After the node constraints have been determined, it is easy to calculate the types of the messages for the message pool and the bypass pool. Obviously, the message pool for each node consists of all message types that are used in at least one of its node constraints.

The message types needed for the bypass pools are determined next. For each message type, every node on the path between the first and the last use of messages of that type is examined. When the message type is not contained in the message pool of a node on that path, it is added to the bypass pool there. This assures that messages which are needed to determine node constraints at nodes closer to the root are correctly forwarded there.

The table below shows the node constraints and the types of messages that must be inserted into the message and bypass pools for the pattern graph in Figure 4.2.

Nodes	Node Constraints	Dynamic Constraints	Message / Bypass Pools
0	∃m(0)		{m(0)} / {}
1	∃m(1)		{m(1)} / {}
2	m(2).time > m(0).time		{m(0), m(2)} / {}
3	m(3).time > m(1).time m(3).time > m(2).time	m(2).a1 == m(1).a2	{m(1), m(2), m(3)} / {m(0)}
4	m(4).time > m(3).time	m(4).a3 == m(0).a0	{m(0), m(3), m(4)} / {}

Table 4.1: Node Constraints and Message / Bypass Pools

Detection Algorithm

Having determined the node constraints for each node (which makes sure that the subtree pattern above this node is satisfied) as well as the message and bypass pools, the algorithm to actually move messages between nodes can be explained. The id of a newly arrived message is checked to determine whether it should go to the message or to the bypass pool. When it belongs to neither group, it is simply discarded. This prevents messages from being moved further towards the root node when they are not needed anymore. When the message belongs to the bypass pool, it is put there and no immediate further actions are necessary. Otherwise it is added to the message pool. Whenever a new message is inserted into the message pool, the node constraints are checked. The algorithm attempts to find a tuple of messages of *different* type (i.e. all with distinct identifications) that match *all* the node constraints. The tuple has to include one actual message instance of each message type (i.e. message id) of the message pool and the new message has to be part of the tuple as well. Consider a potential tuple for node 3 in Figure 4.2 with its message pool {m1, m2, m3}. It must consist of message instances with the ids 1, 2 and 3. When such a tuple (or tuples, when more than one set of different messages match the node constraints) can be identified, the detection process has found messages that match the subtree pattern starting at the local node. All of the tuple's messages have to be moved over the outgoing link to the next node. Because messages in the message pool might be needed later to satisfy the node constraints together with newly arriving messages, the original messages remain in the pool and only copies are forwarded. To prevent the system from being flooded by duplicate messages, each message pool entry is only copied and forwarded to the next node once. For each tuple that matches the local constraints, the bypass pool is inspected. The temporal constraints between each message in the bypass pool and all messages of the matching tuple are checked. When a bypass pool message satisfies all temporal constraints between itself and every tuple message, it is removed from the pool and moved to the next node. This is needed to make sure that only messages which do not violate any temporal constraints are passed on.

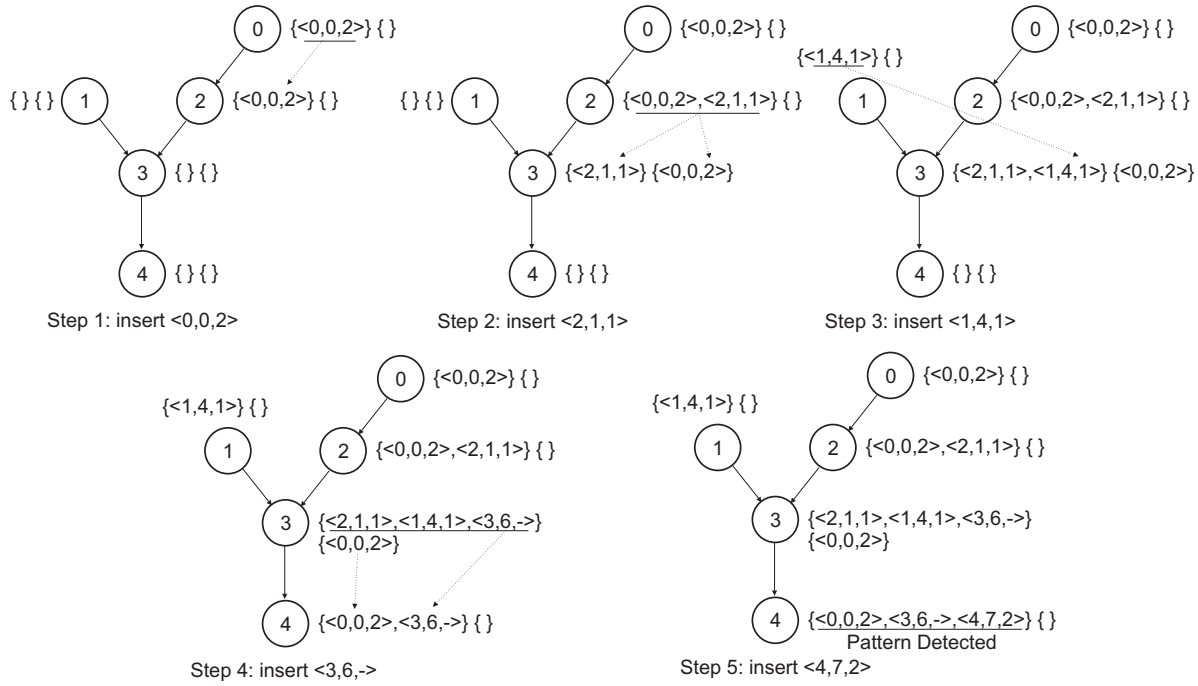

Figure 4.4: Sample Pattern Detection

The situation is slightly different for send nodes. As a send node can have different next neighbor nodes at different hosts (depending on the target of the send event), the copying of message pool entries and the deletion of bypass pool elements must be handled differently. The send node has to keep track which message pool entries have already been copied and which bypass pool elements have already been removed and forwarded to the destinations of the send events for each different destination. This implies that bypass pool elements can never be deleted because they might have to be sent to a completely new destination host. Elements cannot be kept infinitely long because memory is a limited resource. We use timers to remove elements from the message and the bypass pools after a certain, configurable time span. This means that patterns which evolve over a long time might remain undetected. Note that this is not a limitation of our approach but a problem that affects all systems that operate on-line and have to keep state. Such systems need a policy that decides which events to delete when the available memory is exhausted.

The example in Figure 4.4 shows a step-by-step detection of the distributed pattern which is described by the scenario in Figure 4.2. The node constraints of Figure 4.2 are used and each tuple is underlined in the figure. Dotted arrows indicate the copying of messages to the next neighbor. Associated with each node are two sets enclosed in brackets. The first holds the node's current message pool entries, the second its bypass pool elements.

Discussion

The following section briefly discusses the correctness of the detection algorithm while its performance is evaluated in Section 8.1.

The detection algorithm is correct when it reports an intrusion, if and only if events occur that satisfy the pattern which the algorithm attempts to detect. We briefly provide an informal explanation why this is the case.

The explanation is given recursively on the tree shape of the attack scenario pattern. A node only forwards event information to its (single) successor node in the pattern tree when events have occurred that match the pattern subtree above this node (which includes the node itself as the root of that subtree). This implies that all events of the subtree have occurred in the correct temporal order and satisfy all dynamic constraints which can be resolved up to this point.

A subtree above a node is satisfied when all local events at a node have been detected, the send events of all predecessors have been received and all dynamic constraints are satisfied. As stated above, the predecessor nodes may only forward messages when the subtrees above those nodes have been satisfied as well. That means that the current node only has to check for the occurrence of the local events and must consider all dynamic constraints which can be resolved at that node. This is exactly the way the algorithm works. It simply attempts to find the occurrence of local events in the correct order that meet all local dynamic constraints. These dynamic constraints have been inserted at the earliest node where they can be resolved.

When the subtree above the root of the pattern is satisfied, events that match the whole tree have been found. In such a case, a match is reported.

4.3 Summary

This chapter presents the design of a distributed pattern detection scheme to relate events that occur at different hosts. A pattern specification language (i.e. *Attack Specification Language*) has been defined that enables domain experts to express intrusion scenarios in an easy and intuitive manner. In order to prevent an exponential increase in the number of messages that nodes have to exchange to search for patterns, the expressiveness of the language was restricted. The consequential decentralized algorithm to find events that satisfy such patterns has been explained in detail. While this chapter focuses on an abstract description of the pattern detection algorithm with the needed data structures and its computational steps, the following chapter will provide details about the concrete implementation. The presented approach exhibits superior scalability and fault tolerance properties when compared to current solutions. This claim is supported by the results of our evaluation which are shown in Section 8.1.

Chapter 5

Quicksand: A Prototype Implementation

> The reward of a thing well done is to have done it.
>
> – Ralph Waldo Emerson

Quicksand [51] is the name of our system that implements the framework described in the previous chapter. In addition, it provides functionality to report detected intrusions to a system administrator and an interface to configure and manage a *Quicksand* installation.

As explained above, the system can be used to describe and detect situations where a sequence of events occurs on multiple hosts (such as a port scan detected at the firewall followed by a packet containing buffer overflow exploit code that targets the web server). This enables the sensors to modify their reaction at nodes that become aware of emerging, hostile patterns. *Quicksand* implements a flexible response mechanism that allows an administrator to specify actions for each step of an intrusion pattern. In contrast to traditional systems which usually only react (e.g. generate warnings or harden the firewall) after an incident has been detected, our mechanism allows fine-grain control over potential counter-activities while threatening situations emerge. A plug-in mechanism allows the extension of *Quicksand* with such response modules and additional sensors. An important plug-in to realize the system's ability to adapt to traffic from suspicious sensors is described in Chapter 7. This sensor is needed to extend *Quicksand's* detection framework with the capability to selectively react to emerging patterns, a behavior needed for *Network Alertness*.

Quicksand is implemented in `C` to allow a high degree of portability and to get good system performance. A `SSL` library [68] is utilized to secure network communication between nodes. The parser to process the pattern specifications is created with the help of `flex` [33] and `bison` [12].

5.1 System Architecture

This section describes the architecture and implementational details of the realized prototype (see Figure 5.1).

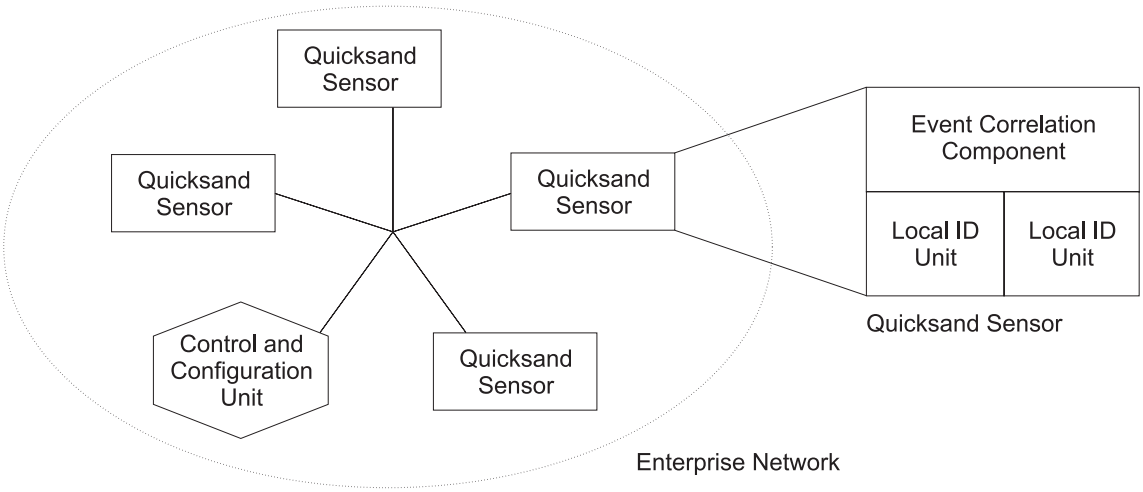

Figure 5.1: Quicksand System Architecture

Quicksand as a system basically consists of a set of hosts which are connected by a network. Each node runs a *Quicksand* sensor that is made up of several local intrusion detection units and an event correlation component. The collaborating event correlation units deployed throughout the network implement the decentralized detection process. A (possibly redundant) control component is utilized to perform system configuration and management. This component can also act as a central point where alerts can be collected. Notice that although only one control unit is sufficient for a *Quicksand* installation, it does not participate in the detection process and therefore cannot be considered as a single point of failure. As each detecting node itself can initiate a response, it is not necessary to forward an alert to the control unit.

5.1.1 Local Detection Unit

The local intrusion detection components are responsible for gathering audit data. They can be implemented as host based units which collect data from the operating system and user processes or as network based designs that monitor packets via the machine's network interface card(s). Local detection units can apply misuse or anomaly based methods to extract interesting information (i.e. events) from the local data stream. These components interface directly with the underlying operating system and hardware.

The task of a local detection unit is to identify occurrences which are relevant for distributed attacks and pass this information to the correlation unit. The interesting occurrences are called basic events (see Section 4.1). Each basic event that is detected in a sensor's input stream is transformed into an instance of the corresponding event class.

An event class has a type and a list of attributes with their respective types. All event types have to be determined during the compilation phase of the system (i.e. they are built into *Quicksand*) and every local detection unit is capable of producing instances of event classes of one or more types. The event classes are described by annotated `C structures` where each attribute is a member that can be a basic `C type` or a reference (pointer) to another structure (i.e. event class). This allows us to assemble more complex event descriptions as compounds of simpler objects. The type of the class corresponds to the name (i.e. type) of the `C struct`. An annotated class description can be directly transformed into the corresponding `C struct` by adding a few management fields.

The annotation is used to specify the attribute that determines the target of send events. The value of this distinguished attribute is later processed by the correlation unit to determine where messages need to be sent to. Obviously, it is possible to define classes without any send events. *Quicksand* uses different event classes that correspond to packets on the network, classes that reflect activity on a host machine and generic classes that allow an easy extension of the system when events of new types emerge and need to be integrated. Similar to object-oriented programming languages, classes can extend other classes and inherit their attributes.

Consider the case of a network based sensor that can process `IP` based traffic and produces event class instances of the types `IP`, `ICMP`, `TCP` and `UDP`. As a `TCP/UDP` packet is encapsulated (contained in the payload) of the `IP` packet, the corresponding `TCP/UDP` event classes have to extend and inherit the attributes from the `IP` event class. A brief excerpt from the `IP` and `TCP` class definitions (which are both classes that correspond to network packets) are given below. The `C structs` that represent those classes are shown next to the definitions. It is shown how inheritance, which can not directly represented in `C`, is realized with a pointer to the parent class.

```
class 'IP' {
   ...
   int32 source_addr;
   int32 dest_addr;
}

class 'TCP' extends 'IP' {
   int16 source_port;
   ...
   boolean syn_flag;
}
```

```
struct IP {
   ...
   unsigned long source_addr;
   unsigned long dest_addr;
}

struct TCP {
   unsigned short source_port;
   ...
   unsigned char syn_flag;

/* parent class - IP */
   IP *parent;
}
```

As mentioned above, the basic event classes and their respective attributes must be known to the system (i.e. sensors, correlation unit and the management component) in advance.

Although it is be possible to forward every detected event to the correlation engine, the required processing and network bandwidth capacity would soon exceed the available resources. Therefore, a filter has to be applied to extract only events that might be part of a larger pattern - which means that they are referenced in at least one ASL scenario description. Our *Event Definition Language (EDL)* is used to specify such filters that describe these interesting occurrences (i.e. basic events) in a sensor independent way. Whenever a distributed pattern is loaded into a correlation unit, the basic event specifications need to be forwarded to the local detection component to make sure that the correct objects are delivered. The following descriptions show two filters named `'udp-filter'` and `'tcp-connect'` written in EDL. The first one matches `UDP` packets originating from `IP` address 192.168.0.1 sent to port 7 while the second matches `TCP` packets that initiate a virtual connection (`TCP` 3-way handshake). Notice that the `source_addr` attribute is accessible in `'udp-filter'` because the `UDP` class is derived from `IP`.

```
'udp-filter': UDP {
  source_addr = '192.168.0.1';
  dest_port = 7;
}

'tcp-connect': TCP {
  syn_flag = true;
}
```

A problem arises when third party sensors should be directly integrated into our framework. As explained above, every sensor must accept EDL definitions that specify the relevant basic events for the corresponding intrusion scenario. In addition, they must also deliver reports of detected occurrences of those events as instances of our event classes.

This is solved by an *adaptation manager* that translates between the correlation unit and the local detection components. This manager accepts sensors as plug-ins and performs two important tasks. First, it collects reports from all active sensors and forwards them in a single, timely ordered stream to the input queue of the correlation unit. Second, it sends the EDL definitions to only those sensors that can detect events of the class specified by that definition. This prevents, for example, network based sensors from receiving filters for host based events.

Each sensor can either be directly connected to the adaptation manager or must provide a proxy that can map the sensor's objects into our object framework and vice versa. The proxy then translates from the sensor-independent EDL specification into a specific configuration suitable for the probe.

A number of different sensors have been integrated into *Quicksand*. For the first prototype, we have implemented a network based probe that puts the network interface card (NIC) into promiscuous mode. This allows capturing of all packets that are transmitted over the connected wire, not only those that are destined for the listening host. Currently, we support the `IP`, `TCP`, `UDP`, `ICMP` and `Ethernet` protocol. Packets are matched against

filters that look for unusual values of header fields or signs of known exploits in the payload. As a proof of concept for our adaptation manager, we substituted this sensor with `Snort` [81]. `Snort` is an open-source network intrusion detection system that accepts simple rules to filter `TCP`, `UDP` and `ICMP` packets. It is relatively wide-spread and especially known for its good performance and ease of use. Filtering is done by rules which are enumerated in a configuration file, one line per rule. The following two lines show the translation of the two EDL definitions shown above. These rules are automatically generated by our `Snort` proxy plug-in when it receives the EDL definitions from the correlation component. Any alerts received from `Snort` are taken by the proxy and converted into the correct event class instances. Notice that the variable `$host` is always defined by the proxy as the `IP` address of the host where the sensor is installed.

```
alert udp 192.168.0.1/32 any -> $host/32 7 (msg:'udp-filter')
alert tcp any any -> $host/32 any (flags:S; msg:'tcp-connect')
```

A simple sensor that is capable of processing `Linux syslog` entries has been written as well. This demonstrates that both host and network based sensors can be applied to our design. This sensor scans the OS log entry for authentication events and forwards failed attempts to the correlation unit.

To provide our system with the needed anomaly based, adaptive sensor, a local component that scans DNS traffic for suspicious content is integrated as well (for more details, see Chapter 7).

5.1.2 Event Correlation Component

The main task of the correlation unit is the implementation and execution of the distributed detection algorithm (which has been described in Section 4.2). It operates on the data provided by the local detection unit and on information received as messages from other, collaborating nodes. The structure of a correlation component is shown in Figure 5.2 and explained in detail below.

The event data is taken from the main input queue that is fed by the adaptation manager (see previous Section 5.1.1) and the communication subsystem. Whenever the system learns of new event data (either produced by a local sensor or encapsulated in a message), the corresponding object is added to the queue. A consumer thread (called *dispatcher thread*), which is started by the correlation unit, retrieves these event objects one at a time and forwards them to the relevant scenarios. A scenario is relevant for an event object when the scenario's description uses at least one basic event that has the same type as the input element. To prevent the overhead of copying the event object for each scenario, only references are passed. A simple garbage collector makes sure that the memory space of every object is freed (collected) after all scenarios have processed it.

A separate thread, each with its own input queue, is started for every scenario. The dispatcher thread appends the references to new input events to the input queues of all relevant scenarios. From there, they are removed by the scenario threads and further processed.

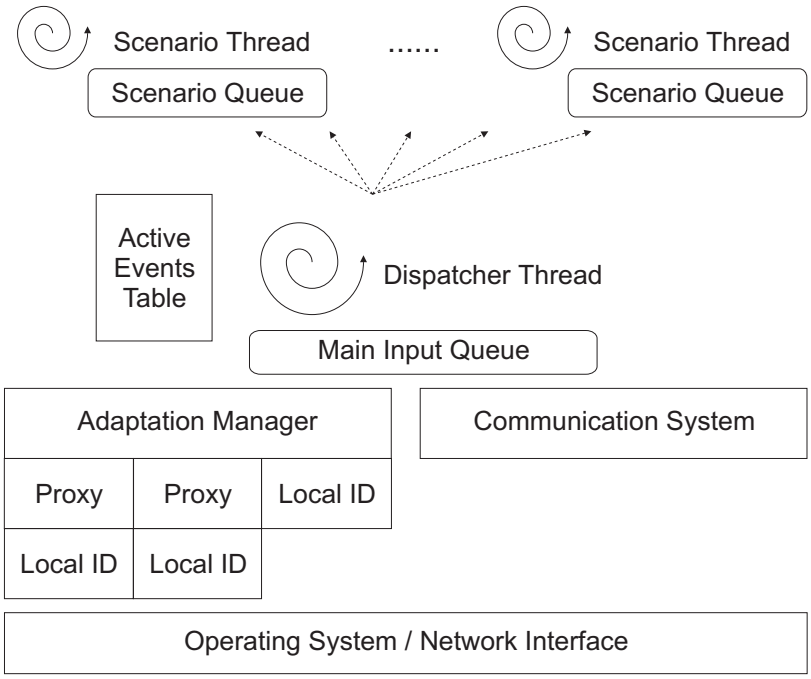

Figure 5.2: Correlation Component Architecture

An optimization has been implemented that prevents events of certain types from being forwarded to scenarios when it is impossible that they are part of a pattern even though the type is used in the scenario description. This is the case when a pattern includes local sequences with more than one event (i.e. node sequences of length 2 or longer). As it is required that all local events occur consecutively, it is not necessary to consider the second (or any later) position in the sequence when the first event has not been seen yet. This leads to the definition of *active* event types. The set of active event types for an attack scenario only includes event types that can potentially be part of an ongoing intrusion. That means that immediately after system initialization, only the types of the first events of all node sequences in the scenario need to be considered (and are therefore active). As an attack progresses, currently active events are witnessed. This causes their successor events (in the host sequence) to be activated too. It is also possible to deactivate events in the case that a predecessor event in the sequence times out and is removed from the detection process.

The active events for each scenario are stored in the *active events table* which is consulted by the dispatcher process before forwarding events to the relevant scenarios. It is important to synchronize the activity between the scenario threads and the dispatcher. It is possible (even common) that two events that are important for the detection of an attack pattern happen within a very short time period. When the occurrence of the first event activates the second event it is mandatory that the dispatching of the later event is not done before the scenario thread has processed the first event. Otherwise, the second event type would not have been activated yet and the event is lost.

The processing of an event by the scenario threads is straightforward. The important data structures for the algorithm are already present as they have been calculated offline. After moving the event to the correct node in the pattern graph (or possible more) the only difficult part is to determine efficiently the tuple of messages of *different* type that match *all* the node constraints (see Section 4.2.3).

To calculate the valid message tuples in a naive way, it is necessary to determine all subsets (derived from the set of messages in the message pool) that contain the newly arrived message and have members (messages) of the correct types. Then it is necessary to check for each subset whether the contained messages satisfy all node constraints and discard those that do not.

Unfortunately, the amount of these subsets is proportional to n^l where n is the number of messages stored at a node and l is the cardinality of the subset. Therefore it is computationally too expensive to proceed in this manner. *Quicksand* solves the problem by keeping all message combinations that do not violate any node constraint in a search tree. Instead of storing only the messages themselves and creating new tuples every time a new message arrives, all partially complete tuples which are valid (i.e. violate no node constraint) are stored. A simple search tree is shown in Figure 5.3. The message pool consists of three different message types with the `ids` 0 (`m(0)`), 1 (`m(1)`) and 2 (`m(2)`). The 'constraints table' lists the the constraints between the attribute values of the messages. Currently, the message pool contains five message instances. All partial tuples that satisfy the constraints have been inserted into the search tree which is a binary tree with 2^3 leaves (2^n, where n is the number of different message types used for the message pool). A path from the root to a leaf determines the types of messages that are associated with that leaf. Choosing the left child of a node implies that the message type which corresponds to this node is included, choosing the right child implies that the message type is excluded from the partial tuple. The number in each box that is associated with every terminal node shows the number of valid partial tuples that belong to this leaf.

Whenever a new message arrives, it is added to all partially complete tuples that do not include the type of the new message. When this results in a new partially complete tuple that is still valid, it is simply added to the search tree. When a complete tuple is the result, it is used for further processing. Most of the time, however, a violation of a node constraint is detected and the result is discarded. This approach trades storage for speed. Instead of checking all message subsets for their validity every time a new message is received, all valid partially complete tuples are already stored. Because the total amount of valid tuples is only a very small fraction of the amount of possible subsets, the size of the tree is manageable. Even in the case of several thousand events, the size of the tree remained under a few megabytes and the processing of new events could be handled in a few milliseconds.

The correlation unit is also responsible for reacting on detected security violations. The ASL definition of an attack contains a description of the response functions (i.e. counter-measures) which should be invoked. A possible active response is the hardening of firewalls or the interruption of open network connections. Such activities have to be performed fast and reliable, therefore no central entity is involved. In addition to that, a

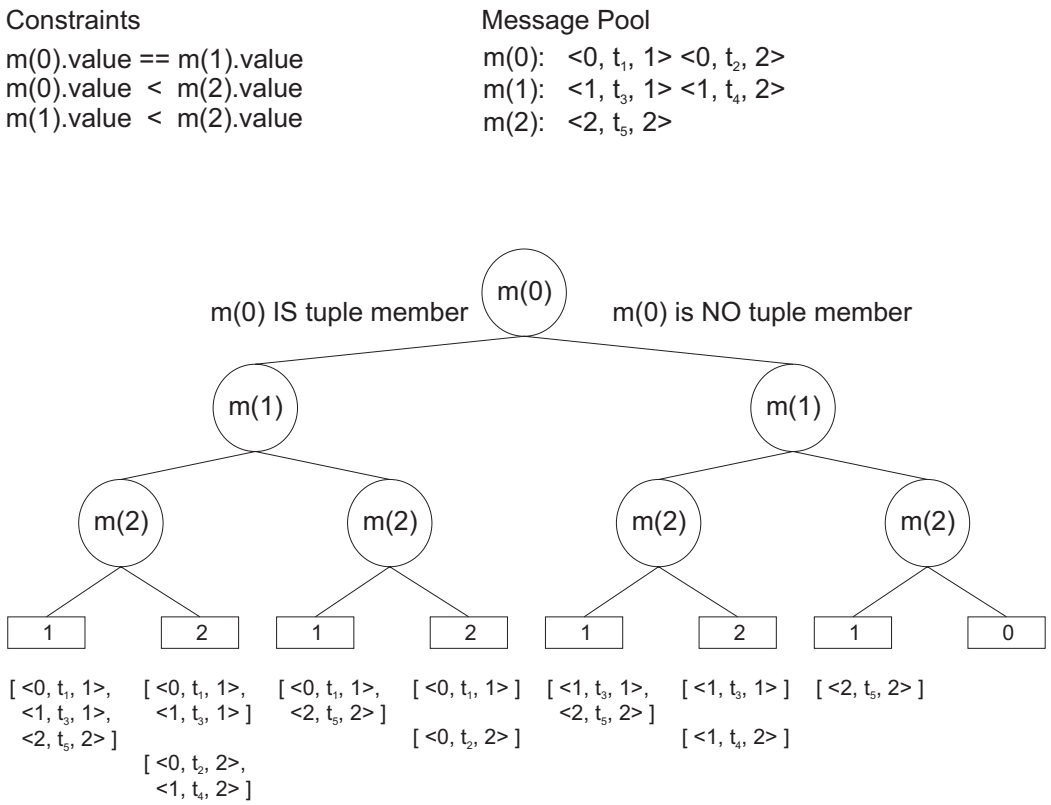

Figure 5.3: Message Tuple Calculation

passive response can be executed by transmitting an alarm message to a dedicated node running a control unit. Two default reactions have been currently implemented. One sends a message to the `syslog` daemon reporting an alert to the local machine. While this seems to be limited in use, consider that `syslogd` can be instructed to forward information to a central log server that can be monitored by a system administrator. The second response is more flexible as it invokes an operating system shell that allows execution of arbitrary scripts. This has been used to send mail to a system administrator. A more sophisticated response model is currently under development. This includes active response mechanisms (e.g. firewall reconfiguration) and balances the expected impact of the response with the detected threat. Especially for e-commerce sites, it can be undesirable that suspicious connections of actually innocent customers are interrupted.

The communication between nodes is secured by utilizing a `SSL` library [68]. Every node receives a private and a public key when the correlation unit is installed at a host. The public key is signed by a certification authority (CA) that is responsible for the protected network. The public key of this CA is present at each correlation unit as well. When a secure `SSL` channel is set up, the public/private key pairs allow the communication partners to authenticate each other which prevents malicious nodes from impersonating legal senders.

5.1.3 Control Unit

At least one host needs an installed control unit. This central module is utilized by the system administrator to configure the system. Its task includes the processing of attack specifications written in ASL and the configuration of the local intrusion detection and event correlation units that are distributed over the protected network.

Pattern specifications that are written in our *Attack Specification Language* need to be translated into data structures and functions suitable for the detection process. This is done by an ASL parser that translates attack patterns into `C` source files which are then compiled into object files. Response functions are specified as `C` functions and compiled into separate object files. Instead of using a generic interpreter to process pattern specifications, the detection algorithm is directly run as machine code. To remain flexible and to be able to integrate new patterns on-the-fly (i.e. without changing the code of the event correlation unit), the algorithm's code is divided into a pattern dependent and a pattern independent part. The independent part is implemented as shared code in the correlation unit. Similar to an operating system, this part provides basic services that can be accessed by the threads running the attack scenario dependent code. The pattern related part is stored in the libraries produced by the ASL parser and the response libraries. These libraries are dynamically loaded into the *Quicksand* sensors and execute the pattern dependent parts.

It is straightforward to translate the grammar introduced in Section 4.1.3 into a suitable LR(1)-description needed by the parser generator. The parser itself creates all necessary data structures (e.g. message and bypass pools) and a thin code layer that runs the pattern dependent code.

Object files are merged into shared libraries which may be shipped to sensors over a secure connection (via `SSL`). These libraries are automatically installed and can thereupon be utilized by the sensor. The control unit is used to remotely load and unload attack patterns. As described above, a dedicated thread is started to execute a certain pattern whenever it is loaded into the correlation engine.

The control unit uses a local database to store information about the libraries that have already been installed at each host. This database holds a list of symbols exported by each object file as well as a list of installed object files at each host. That information is used to prevent the installation of libraries that depend on response functions in different object files which have not been installed yet. Currently, we only issue a warning message that enumerates the symbols that cannot be resolved. The problem has to be handled by the system administrator who has to perform the necessary installations manually. In the future, we plan an automatic support to perform this update automatically.

A sensor that scans its input for the occurrence of a certain event needs to be compiled against this event's corresponding `C struct`. Therefore, we cannot add new basic events types to sensor while the system is running. This is no real limitation, as the kinds of basic events usually do not change frequently. It is more important to be able to update and modify pattern descriptions which represent the actual attacks, an operation supported by *Quicksand*.

5.2 Event Synchronization

The use of patterns that can specify temporal relationships between events occurring at different hosts introduces the following challenge. The relative order of distributed events needs to be known locally at the node that has to determine whether certain events satisfy the time constraints of a distributed pattern.

Consider the case when a node A sends a message M_1 to another node B and the transmission of this message is defined as a send event of a pattern currently evaluated at node A. The *Quicksand* instance at node A detects M_1 and transmits a second message M_2 to B informing it that M_1 has been a relevant send event. When the *Quicksand* correlation unit at node B gets M_2, it has to consider all events that have occurred between the receipt of message M_1 and M_2 as relevant for the pattern. The only important question that remains is how B can determine when M_1 has reached it.

This task is easy when only one message has been exchanged between A and B but becomes more complicated when several are involved. Especially when dealing with real world networks, packets can get lost, delayed or duplicated. This prevents nodes from simply counting the number of received packets and using this number to exactly specify the packet at the sending host.

To solve this problem, each packet has to be uniquely identifiable. This enables the *Quicksand* instance at the sending host to uniquely refer to each packet that contains an interesting send event and the receiver to consider only those events for a pattern that have been monitored after the relevant packet has been received. In order to identify the packets, we use logical timestamps. Each host utilizes a logical clock to mark outgoing packets with logical timestamps. The sender can then refer to this logical timestamp when it detects that a packet contains the send event of a pattern. When the receiver later gets the message indicating that a certain received packet has carried a send event, it is easy to decide which local events have occurred before and after the reception of that packet (with a known logical time). We call the problem of including such an identification or a logical timestamp into each packet the *timestamp wrapping problem*.

Note that *all* packets that are sent between two hosts could potentially be associated with a send event and therefore have to be identified by a timestamp. That means that timestamps have to be transparently integrated into each packet even for applications that are completely unaware of the timestamp service. An example would be a user who utilizes `telnet` to connect to another host. Each typed character causes a packet to be sent to the remote machine that could be associated to a send event.

We have done a simple prototype implementation of such a timestamp wrapping service for *Quicksand* which is presented in detail in the following sections. This serves as a proof of concept that accurate synchronization is possible. For our tests, that involved different operating systems, a simpler approach has been chosen. We assume a maximum delay between the actual send event and the message from one *Quicksand* correlation engine to the other indicating that the send event has occurred. At the receiving node, the processing of local messages is simply delayed for that assumed maximum time span to get a correct temporal order between remote and local events. We are aware of the fact that

this might introduce incorrect temporal dependencies. When messages arrive too early, our system might report intrusions that have not taken place (false positives) while we would miss intrusions (false negatives) when messages take longer than our pre-defined time span. By using a reasonable long delay, false negatives can be practically prevented, especially in local networks. In addition, we have never experienced any false positive as the incidentally occurrence of events specified by a certain pattern during this short time span is very unlikely.

One problem with the presented approach is that it requires the modification of the network stack of the underlying operating system. As long as such a service is not implemented in most common OSs (especially `Microsoft Windows`), the approach can only be used for systems where the source code is available and can be modified. Therefore, we tested our approach with `Linux` and `Solaris` machines.

The timestamp service can also be used to implement logical clocks according to Lamport's 'happened-before' relation [57]. This allows causal ordering of events in distributed systems. As the order of events occurring at different nodes of a network is critical for many distributed applications [23], our transparent time service implementation could be utilized by numerous systems. Ranging from our application of interest, an intrusion detection system to network management tools, it could also be applied to replication services such as `ISIS` [11].

The traditional alternative to logically ordered events are totally ordered events using physical clocks. When the physical clocks of all involved hosts are synchronized with an acceptable accuracy, each packet can be simply assigned a physical timestamp. This enables every node to determine the relative order of events. The needed clock synchronization could be achieved by using `GPS` timers or via the `network time protocol` (`NTP`). However, when we are considering a large heterogenous network where the application of `GPS` for each computer is too expensive and an `NTP` server may not always be at hand, our protocol may be a viable option.

5.2.1 Design Issues

We did not invent a generic method or algorithm to provide a synchronization service but merely developed an actual instance (implementation) of an existing solution (i.e. logical timestamps). Therefore, we had to make several design decisions for our implementation which are discussed below.

These design decisions were guided by four main requirements which our solution should fulfill.

1. *Minimal Overhead* - We want to impose as little overhead as possible on the participating computer systems. That means that the additional bandwidth and computing resources consumed should be minimized. As each sent packet has to contain an identification, it is especially important to keep it short to prevent unacceptable additional network load.

2. *Compatibility* - The solution should be compatible to existing network protocols. It makes no sense to create one's own protocol that encapsulates or substitutes established standard message formats. Existing nodes without timestamp support should still be able to communicate with nodes marking their packets.
3. *Fault Tolerance* - Real networks may fail and one has to deal with lost, delayed or duplicated packets in current datagram (i.e. `IP`) based networks. We need to include enough redundancy to cope with a reasonable amount of network faults. Obviously, this requirement conflicts with the aim to achieve minimal overhead.
4. *Sufficient Granularity* - It might not be necessary for an application to uniquely identify each sent message. One could also group a few messages together and assign an unique identifier to the whole set. This would lead to less overhead, but also negatively affects accuracy. To be safe (and not to miss actual causal relations) one has to assume that all packets of the group have been sent at the same time as the first one. Therefore, some incorrect causal relationships might be introduced at the receiving host. Our solution has to provide sufficient granularity for a broad range of applications and optimally identifies each single packet.

To reach these partially contradicting goals, we had to make choices for a suitable protocol layer, for the exact format of the identifier and its integration into packets as well as decide between absolute and relative timestamps. For each point, a number of alternatives are described in the following subsections and the rationales for our decisions are provided.

Choice of Protocol Layer

The first design issue dealt with the choice of the appropriate protocol layer (according to the `OSI` reference model). We had to choose a layer whose messages should be marked and considered the data link (more specific `Ethernet`), the network (`IP`) and the transport layer (`UDP`, `TCP`). The data link layer does not seem appropriate because it only provides a mean for higher layer packets to hop between directly connected computers on their route to the target. As our identifiers are only relevant between the actual endpoints of the communication and the sender cannot influence the fragmentation performed by the data link layer between itself and the receiver, it is not necessary to distinguish (and mark) packets at this level.

A look at the transport layer reveals that two different protocols, `UDP` and `TCP`, are primarily used in most networks and have to be supported. While `TCP` uses its own 32 bit sequence number to tag packets, `UDP` carries no such information. One would have to add additional information to a `UDP` packet which increases bandwidth usage.

Additionally, `TCP` is not a packet oriented protocol but provides a reliable data stream. Such a stream does not provide enough granularity. When a session between two computers is established, many messages may be exchanged over such a channel. It is not enough to record the begin of the session as events may occur later and cause messages to be sent over the channel which are related to the occurrence of events at the receiving node.

The payload field of the application layer would be a natural choice when one would only like to uniquely mark (timestamp) the packets of the appropriate application itself. Our service, however, has to tag all sent packets for arbitrary applications. When the identification would be included into the payload of an unaware program and the target host does not support the timestamp service, the remote application would not be able to deal with the modified packets. Therefore, the payload field of the application layer does not suit our needs.

It seems natural to consider each `IP` packet as a message unit and create an unambiguous relation between the dispatch and the receipt of each `IP` packet. Whenever a node monitors interesting events as a result of a received `IP` packet, it can easily determine later (after being informed by the transmitting node that this packet has contained the send event) which local events have occurred prior and after the receipt of that packet.

In the following discussion, the terms `IP` packet and message are used interchangeably. The `IPv4` packet header (`RFC 791`) is shown below.

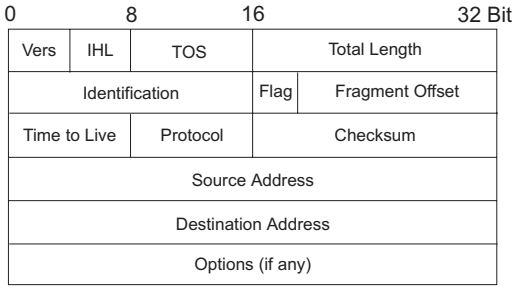

Figure 5.4: `IPv4` Header

Timestamp Format

After the decision to mark `IP` packets had been made, it was necessary to choose a concrete timestamp format and its integration into the `IP` packets. Two possibilities were considered.

First, one could use the information available in the header itself. The first obvious choice is the 16 bit identification field that is used to reassemble fragmented packets at the target. It is simply incremented by one each time a packet is sent. Additionally, we considered the length, fragment offset and checksum field. Although the length field is another 16 bits long, most packets are rather short (to still fit into the longest `Ethernet` frame). Therefore, one would observe many identical values which is unsuitable for a unique identification. The checksum field has to be recalculated after each hop when the time-to-live field is decremented and cannot be used either. One could use the 13 bits of the fragment offset when the DF (don't fragment) bit is simultaneously set. This causes packets to be dropped instead of fragmented when they do not fit into a single data link layer packet and therefore the fragment offset remains untouched. Unfortunately, the compatibility would be effected negatively as packets can not be sent over networks with a small MTU (maximum transferable unit) size.

Second, IP offers the possibility to include up to 40 bytes in an optional section. The timestamp could be added to the header inside the option field but this would use up network bandwidth proportional to the size of the identifier.

We chose not to abuse or modify the header information and only use the 2 byte identifier field. Additionally, we do not extend the size of the packets in any way by adding options to the header or data to the payload field.

Absolute vs. Relative Timestamp

An important decision had to be made about the design of the timestamp itself. It is necessary to uniquely identify a packet over a long period of time, optimally over the whole lifetime of the node which sends messages. Two options are available to achieve this goal.

One uses a single counter (i.e. absolute timestamp) for the complete lifetime of the node that is incremented each time a packet is sent. The counter is saved in non-volatile memory and restored on boot up so it is necessary to provide enough room to cover all packets sent by that node. In order to prevent an overflow of the counter, at least 6 bytes are needed for a heavy accessed server on a Fast Ethernet (and maybe more when faster networks emerge - e.g. Gigabit Ethernet). When the size of the IP header, which is 20 bytes (see Figure 5.4), is compared to the size of the timestamp, the bandwidth overhead is significant. Especially when one takes into account that the new version of IP (IPv6 - RFC 2460) reduces the non-address part of the header from 12 to 8 bytes.

The other option uses a shorter counter that uniquely identifies packets in a given context. The context is established implicitly or by special synchronization packets. These synchronization packets are acknowledged messages which negotiate a common, agreed context between the sending and receiving node. This allows us to use shorter timestamps for intermediate packets. A shorter timestamp can be integrated into the IP header more easily and has the advantage of reducing the bandwidth overhead. On the contrary, the synchronization might cause additional packets to be sent and it is not possible to rely on implicit synchronization alone because of potential network faults.

A shorter timestamp causes less overhead for each individual packet but makes synchronization necessary more often. We claim that an overhead of 6 to 8 bytes per packet is not acceptable and use a relative timestamp of 2 bytes. This allows us an easy integration into the IP header but demands some work to maintain a consistent context state between nodes. The way to achieve such a common context is described in the next section.

5.2.2 Protocol Specification

All nodes that are attached to the network (e.g. computers on the Internet) can be divided into two sets. One set is called *active* and consists of all nodes that support the dispatch and receipt of timestamps while members of the other, called *passive*, have no notion of logical time. All active nodes send marked packets all the time and do not adjust their messages according to the receiver. That implies that marked packets have to be understood by

passive nodes to guarantee compatibility to existing network protocols. Although active nodes can easily communicate with passive nodes and vice versa, they have to store some state information when talking to active nodes. Therefore, it is necessary for each active node to know whether its communication partner is active or not.

Setup Phase

The knowledge of all active nodes within a network can be disseminated in two ways. One uses a (potential hierarchical and/or distributed) name service where each active node can register itself and query for other active nodes. This would be a database that stores a list of `IP` addresses that support the timestamp service. While this could make sense for isolated regions with a dedicated name server, the approach is not applicable for many interconnected installations. For a global name service comparable to `DNS`, the administrative overhead is surely unacceptable. The other way, which we have implemented, uses a setup phase between two nodes before they communicate for the first time ever. The sending computer transmits a setup request to a well-known port at the receiving machine and waits for a response. Meanwhile, the process that queued the original message for sending is suspended. When no setup reply is received, the node is considered to be passive, otherwise it is definitely an active node and a context can be established. The results of such setup requests are stored in a cache and passive computers are asked again (by sending additional setup requests) after some time. When the reply to a setup message is lost, the sender considers the receiving machine to be passive and establishes no context. The receiving machine (that has established a context) notices the problem when the next setup request arrives. In that case, the context is reset and another reply transmitted.

Timestamp Protocol

In order to uniquely mark all packets that are transmitted to each receiver, the sender has to store a logical timestamp for every target. In our design, a logical timestamp is 8 bytes long and consists of two parts, a 6 byte context and a 2 byte identifier. For each packet which is sent to a node, the corresponding timestamp has to be updated. A different logical timestamp is used between each pair of sender/receiver nodes (when both are active nodes). Notice that a pair of nodes that exchanges messages in both directions actually uses two timestamps.

As described in Section 5.2.1, each packet effectively carries only the 2 byte identifier (stored in the identification field of the `IP` header as a relative timestamp) while the 6 byte context is established by different means (as described below). The 2^{16} (0 to 65535) possible identifiers defined by the 2 byte identification field are divided into three groups, called *low* and *high* and *neutral* as shown in Figure 5.5. The low group ranges from the numbers 0 to 1024 while the high group ranges from 2^{15} to $2^{15} + 1024$ (32768 to 33792). The remaining numbers belong to the neutral group.

The timestamp is initialized only when two active nodes establish a connection for the first time. Even when one of both temporary becomes passive or unavailable, the

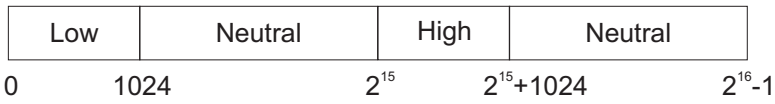

Figure 5.5: Identifier Group Division

context and identifier values are saved and restored the next time the other node starts its timestamp service (i.e. gets active) again. That means that timestamp values have to survive crashes and need to be kept (or mirrored) in non-volatile memory. When context information is lost due to a crash, the appropriate communication partner may use setup packets to bring its partner into a consistent state again. The connection between two nodes mentioned above is different from a `TCP` connection (connection oriented communication). It describes any information that is passed from a host to another one (e.g. `UDP` packet and `TCP` syn packet).

The context at the sender is increased by one every time the used identifiers change their group membership from neutral to low or from neutral to high. Whenever a sender marks a message with an identifier from the low group after identifiers from the high group have previously been used, the context counter is incremented. The same is true for the dispatch of the first high group identifier after previous low group ones. That means that the context counter is incremented twice during a single run-through of the identification counter. The way the protocol works for the sender is shown in Figure 5.6 below. Whenever a packet is ready to be sent, a state transition is made and the actions at the target state are executed. The action `record` describes the insertion of the current identifier value into the identification field of the `IP` header and the storage of the send event in an appropriate database.

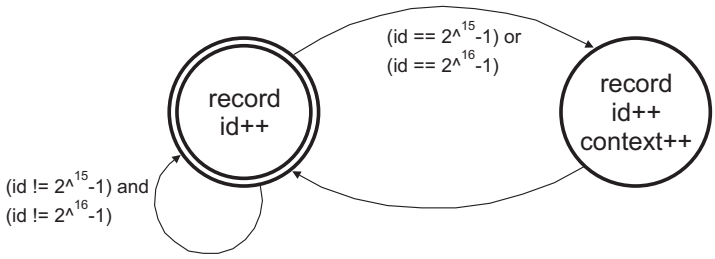

Figure 5.6: Protocol at Sending Node

While it is easy to assign the correct context counter value to each packet at the sender, the situation is not as trivial for the receiver. The update of the context counter at the receiver can be done implicitly or explicitly by special synchronization (sync) packets. An implicit update can be done as soon as a low group timestamp arrives after at least one high group timestamp has been received previously and vice versa. The explicit variant is the sending of synchronization packets. Before the sender transmits its first message with a low group timestamp after high group identifiers have previously been used or vice versa, a special packet including the new context value is sent. This packet is retransmit-

ted periodically until it has been acknowledged by the receiving host. When a number of retransmissions fail but the host is still available, it is assumed that the timestamp service has failed or is no longer active and the host becomes passive. While the synchronization process is running, all new messages to the receiving host are held back. The delay introduced by explicit synchronization is usually (i.e. without packet loss) the time it takes a packet to get to the remote machine and back. This simply triples the time it takes for the regular packet to arrive at the remote host - a time that should be acceptable for almost all applications. Even an interactive computer game can cope with several lost packets (i.e. a small time span where no packets arrive), and under normal circumstances the explicit synchronization will not take longer.

The way the protocol operates at the receiving node is shown in Figure 5.7 below. Whenever a packet is received, the identification field of its `IP` header is extracted and a state transition made. Then the actions of the target state are performed. The action `record` describes the assignment of the correct context value to the packet and the storage of the receive event in an appropriate database.

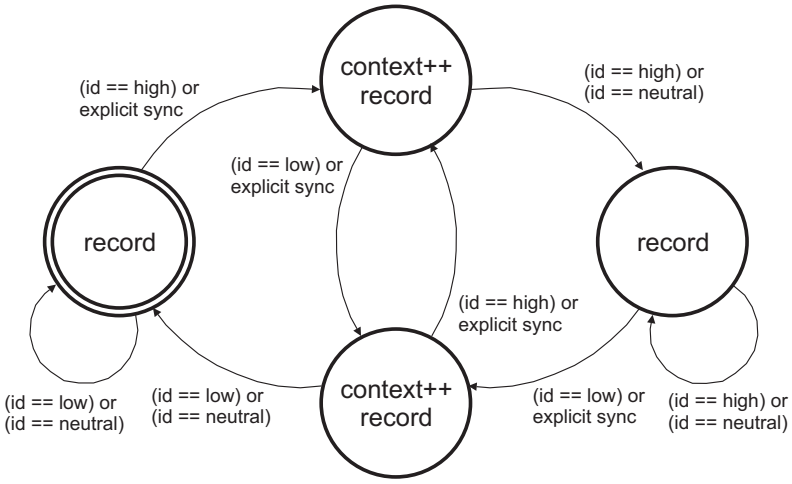

Figure 5.7: Protocol at Receiving Node

Timestamps are assigned to messages at the receiving node by appending the correct context value to the packet's identification value in the following way. When a low or high group packet is received, the current context value is used. When a neutral packet with an identification field less than 2^{15} arrives, the largest even number that is less or equal to the current context counter value is used as its context value (e.g. 6 for a context value of 6 and 7). Packets with an identification field greater than 2^{15} get the largest odd number that is less or equal to the current context value (e.g. 7 for a context value of 7 and 8). This assures that delayed neutral packets are still assigned consistent context values.

Assumptions

In order for the implementation to work correctly, the following assumptions must hold. We assume that the communication subsystem may drop, delay or duplicate packets.

- No more than 1024 consecutive packets may be dropped by the communication subsystem. Otherwise all low or high group packets might get lost and an implicit synchronization fails. When synchronization packets are used every time the context changes, this assumption can be dropped. Usually, communication lines are very reliable and the drop of such are large number of packets is very unlikely.
- A packet with timestamp n must arrive earlier than all packets with timestamps greater than $n + (2^{15} - 1024)$ or not arrive at all. This is necessary for neutral group packets to be assigned to the correct context. Additionally, a low group packet that is delayed until later high group packets are received causes an illegal increment of the context counter. The same is obviously true for delayed high group packets.

  While the incorrect increment of the context can be prevented by always sending synchronization packets, the incorrect assignment of packets to wrong contexts cannot. Unfortunately, this problem is always existent when utilizing relative counters and allowing arbitrary delays of packets.

  The sending of a synchronization packet can help when packets are regularly delayed. As new packets are held back during the synchronization process, the time it may take old packets to arrive is extended by at least the round trip time needed for the sync messages. This reduces the likelihood of incorrect assigned packets. When packets are sent via `Ethernet` on a LAN the order of packets is preserved. Only when packets are sent over several hops and routers choose different paths to the same destination, packets may overtake previous ones. However, it is extremely unlikely that some packets are sent over a very slow line while a couple of thousand others are later routed over a fast link and arrive earlier (although it might happen in theory).

As stated above, the loss of packets during transmission causes no problems as long as it occurs infrequent enough or sync packets are used. The duplication of packets poses no problems either as the receiver simply records the arrival of two or more packets with the same timestamp in its event log. The pattern matching process itself will have to deal with such duplicates but as the same packet has been received multiple times it has to be recorded that often. The most severe problem are delayed packets which are assigned to a wrong context or which cause premature (i.e. illegal) context increments at the receiver.

The following paragraph deals with consequences of inconsistent context updates or incorrectly assigned packets. Whenever a packet is assigned to a wrong context (assuming the context values are consistent) it gets a later timestamp than it actually should. Nevertheless, the correlation process at the sending node notifies the receiver about the interesting event using the correct, earlier time. When the pattern matching algorithm at the receiver eventually uses that earlier timestamp to find all events that have happened

after that point in time, it considers events that have occurred before the actual packet has been received.

In our area of application (i.e. intrusion detection), the consequence could be a false positive, the raise of an unjustified alarm. Nevertheless, the occurrence of an incorrect warning is much less severe than the opposite, the miss of an actual intrusion. Therefore, a rare incorrect assignment of a packet to a later context can be tolerated. As packets are never assigned earlier timestamps, it is impossible to miss existing happened-before relations and therefore a detectable intrusion pattern cannot remain unnoticed.

Inconsistent updates of the context counter may occur in two different ways. On one hand, the loss of all low or high group packets results in a smaller context counter value at the receiver. When such an inconsistency is detected during the exchange of sync packets, the context parts of all events received since the last successful synchronization are increased accordingly and the context value is adjusted correctly. This causes all affected messages to be potentially dated later than they have actually been sent. This might result in the detection of non-existing relations as some (or even all events) are dated later than they actually have occurred. But as stated previously, to be on the safe side, events are assumed to have happened at the latest possible point in time.

On the other hand, a delayed low or high group packet might result in a larger context counter value at the receiving node. In such a situation, the packets might only have been dated later. As stated above, this situation can be considered acceptable and so only the context value is adjusted.

Adaptive Synchronization

In order to decrease the network load, we use a technique that we call *adaptive synchronization*. This approach starts by having the sending host transmitting synchronization packets after every second context change. When the connection between both nodes is reliable, the sending host notices that all synchronization packets are immediately acknowledged and both nodes have a consistent view of the current context. This allows us to gradually relax the dispatching of sync packets and rely upon implicit synchronization. Whenever a sync packet needs retransmission or the context values do not match, the time between explicit synchronization is immediately reduced.

Outlook on future IP versions

Another question is the applicability of our approach to the new version of `IP`, namely `IPv6`. As stated before, the size of the non-address part of the `IPv6` header has been reduced to 12 bytes. The field for identifying a packet in case of fragmentation was moved to the optional header part. Nevertheless, a 20 bit flow label field was introduced which is intended to uniquely mark a sequence of packets from a source to one or multiple targets. It seems natural to use this field as a new, slightly extended identification field. This would allow us to reduce the number of necessary explicit synchronization packets by a factor of 16.

5.2.3 Implementation

Our implementation is developed in C for a Linux 2.2 kernel. We need to be able to do the following things.

First, our module has to send and receive synchronization messages (realized as UDP packets). This can easily be done with normal privileges in user mode by using the network (socket) interface.

Second, we must be able to inspect outgoing as well as incoming IP packets. This is necessary to read and set the identification field of the IP packet headers of messages that are exchanged with other active hosts. Reading could either be done by using a packet filter interface such as BPF (BSD packet filter) as used by libpcap or by writing a kernel module that uses the firewall interface. The writing of IP header fields requires a kernel module with direct access to the packet or a kernel patch.

Third, packets have to be delayed during the synchronization process. Whenever a process attempts to send a packet to a target host that is currently involved in exchanging sync packets, it is put to sleep. This can be realized directly inside the kernel (network code) or by a kernel firewall module.

The implementation is realized as a combination of a daemon process and a kernel module using the firewall support (compiled into the kernel). The kernel module is responsible for updating and reading the identification fields while the daemon process runs in user space and performs the synchronization. The communication between both parts is done via system calls.

5.2.4 Evaluation

Performance

This subsection deals with the performance impact of our logical timestamp implementation. We generate additional overhead at two places.

One is the dispatch and receipt of sync packets. This happens at most once every 2^{15} packets and when adaptive synchronization is used considerably less often. The size of a UDP packet including all header information (20 bytes for IP, 8 bytes for UDP) and payload (maximal 12 bytes in our implementation) is limited to 40 bytes. When two packets are exchanged, the resulting proportional increase in bandwidth usage is 0.002 bytes per packet. Each regular IP packet is sent without any additional overhead and the network bandwidth remains unaffected.

The other is the processing overhead at the communication partners. Whenever a packet should be sent to an active host, the appropriate identifier has to be loaded and written into the IP header. A hash table is used to map IP address of target hosts to their current context and identifier values. Additionally, the dispatch and the receipt of a packet has to be stored in the event database for later analysis. Compared with the normal processing of incoming and outgoing packets, this effort is nevertheless negligible.

Storage Considerations

It is necessary for a host to store the logical timestamps (8 bytes) for each active host that it is communicating with. The timestamps have to be saved between reboots and it is necessary to have an entry for every active node that host has exchanged messages with. Parts of this database are held on disk and only entries that are currently used are loaded into kernel memory space into a hash table. Even when assuming that a host contacts a few thousand active nodes, the size of the data remains relatively small (a few kilobytes) which seems acceptable for current systems.

A more severe problem is the storage of the event data itself. The granularity of using `IP` packets is very fine and a node that sends and receives packets at full network speed can create thousands of events (i.e. packets) every second. It is obvious that not every packet should be recorded in an event log. We aggregate all event information produced by packets between interesting external events (events not caused by dispatch or receipt of an `IP` packet) into intervals. That means that only a single entry in the event database has to be present for all packets that are received between two external events.

5.2.5 Applications

Every application that needs to relate event data from different distributed nodes might profit from our approach. We assume that events at a node B can be influenced by events from node A only when a message has been sent from node A to B after the relevant event occurred at node A and before the relevant event took place at node B. Current systems that use such assumptions and provide event synchronization usually gather event data at a central location from different nodes. This simplifies the ordering of processing of events as information can immediately be timestamped on arrival and is available locally. Most systems that correlate events from different sources to deduce additional information about the state of the network follow this approach. This ranges from network management tools such as `HP Open View` [85] to intrusion detection systems (refer to Chapter 3). When one intends to do distributed analysis of data, it is hard to determine at a receiver node which events have happened at the sender before it dispatched a certain message, especially after a longer period of time when clock drift effects or resets of physical clocks are issues. One possibility is the introduction of a service that synchronizes the physical clocks with an acceptable accuracy and use physical timestamps to mark the sending and receipt of events. This is a variant that seems less than desirable in large, heterogenous network environments and again introduces a central service into the system. The other variant uniquely marks packets and allows a sender to refer to such an identifier to inform the receiver which packet is interesting. Our proposed logical timestamp implementation provides such a service.

5.3 Summary

The last two chapters explain in detail our proposed correlation framework. While the previous chapter discusses the pattern specification with the help of our declarative *At-*

tack Specification Language and the resulting distributed search algorithm, this chapter concentrates on the practical aspects of the system.

We present *Quicksand*, the implementation of our intrusion detection system that consists of the distributed search algorithm, probes to gather event data and components to set up and manage a system installation.

Quicksand provides the core detection algorithm together with a communication framework that enables nodes to exchange messages needed for the correlation process in a secure manner. It includes local sensors to obtain data from the host's operating system and the network. A control component is available to translate pattern specifications into a representation usable by the search algorithm and to distribute those representations to all hosts. *Quicksand* offers an interface to extend its functionality with new sensors and response modules. One such module that realizes the adaptive behavior which is needed for *Network Alertness* is the anomaly based network sensor that is presented in Chapter 7.

An important prerequisite for distributed pattern detection is the knowledge of the causal order of events that take place at different hosts in a network. This chapter presents an efficient protocol that allows the synchronization of interesting occurrences at different nodes. This is done by including logical timestamp information into the header of the underlying communication protocol (`IP` in our case).

Although the expressiveness of our pattern language is very powerful, not all desirable scenarios can be specified. Intrusions that do not manifest themselves as patterns of related events cannot be described directly. The following chapter introduces a complementary mechanism that allows to efficiently deal with such situations.

Chapter 6

Centralized High-Speed Event Correlation

Beware of dissipating your powers; strive constantly to concentrate them.

– Johann Wolfgang von Goethe

The distributed correlation framework introduced in the two previous chapters provides a scalable and fault tolerant mechanism to detect **related** events on different hosts. The patterns that can be defined and found by this approach describe scenarios of related activity that targets a single node. This single node is used as the root node of the tree-shaped pattern where the detection algorithm terminates. Send events are needed to establish links between interesting occurrence at distributed hosts.

But there are other scenarios that do not follow the assumption that the relationship between distributed events can be monitored as send events. Malicious activities can manifest themselves at different nodes without any communication that takes place between those spots. Important members of this class of intrusive behavior are port scans and doorknob rattling attacks initiated from an attacker in the outside network. In these scenarios, many connection attempts are made from a single source outside to many nodes inside the protected installation, but there is no traffic between these inside hosts themselves. That prevents our correlation framework to model and detect such reconnaissance activity.

A reasonable way to combat that deficit is the deployment of a single sensor [54] at the bridge between the inside and the outside network. Activity that occurs completely inside the network is either visible as connections between hosts or as sequences of events that occur only locally at a single machine. Such behavior can be modeled by our correlation framework. When the assumption is made that all traffic from the outside has to pass the bridge where our sensor is installed, it is possible to monitor all attacks from the outside that are aimed against multiple hosts inside the protected zone at this point. This assumption is reasonable because almost all sites protect their network with a firewall which is exactly the bridge needed for our sensor deployment.

The sensor at the bridge is realized as a network based probe that scans the traffic between the inside and the outside network. It is based on NetSTAT [102, 103] which

is part of the STAT tool suite [101]. The problem is that the detector is installed at a high-speed link that carries the complete traffic between the protected network and the outside. To be able to find port scans or doorknob rattling attacks, it is not sufficient to treat each packet independently. The sensor has to perform *stateful inspection* of the traffic it monitors. In this case, the probe has to maintain information about attacks in progress (e.g. in the case of multi-step attacks) or it has to perform application-level analysis of the packet contents. These tasks are resource intensive and, in a single-node setup, may seriously interfere with the basic task of retrieving packets from the wire.

Current network based IDSs are barely capable of real-time traffic analysis on saturated `Fast Ethernet` links (100 Mbps) [38]. Analysis tools that can deal with higher throughput are unable to maintain state between different steps of an attack or they are limited to the analysis of packet headers. As network technology presses forward, `Gigabit Ethernet` (with 1000 Mbps) has become the de-facto standard for large network installations. In order to protect such installations, a novel approach for network based intrusion detection is necessary to manage the ever-increasing data volume.

The centralized sensor is a complementary approach that can be deployed in addition to *Quicksand* to cover attack scenarios that cannot be modeled in our *Attack Specification Language*. Because *Quicksand* is intended to be utilized for large network installations, the centralized probe has to cope with high-speed links.

6.1 Traffic Slicing

To solve the problem of managing the massive data volumes on the network link, we propose a partitioning approach to network security analysis that supports in-depth, stateful intrusion detection. The approach is centered around a *slicing* mechanism that divides the overall network traffic into subsets of manageable size. These subsets (or slices) are then sent to a number of distributed sensors. The traffic partitioning is done so that a single slice contains all the evidence necessary to detect a specific attack, making sensor-to-sensor interactions unnecessary. This approach has often been advocated by the high-performance research community as a way to distribute the service load across many nodes. In contrast to the case for standard load balancing, the division (or slicing) of the traffic for intrusion detection has to be performed in a way that guarantees the detection of all the threat scenarios considered. If a random division of traffic is used, sensors may not receive sufficient data to detect an intrusion, because different parts of the manifestation of an attack may have been assigned to different slices. Therefore, when an attack scenario consists of a number of steps, the slicing mechanism must assure that all of the packets that could trigger those steps are sent to the sensor configured to detect that specific attack. Our design allows for meaningful slicing of the network traffic into portions of manageable size.

After a discussion of recent developments in this area in Section 6.2 the system design is presented. The following Section 6.4 describes the architecture of our prototype tool and Section 8.2 presents the results of the first quantitative evaluation of our implementation.

6.2 State-of-the-Art

The possibility of performing network based intrusion detection on high-speed links (e.g. on OC-192 links) has been the focus of much debate in the intrusion detection community recently. A common position is to state that high-speed network based intrusion detection is not practical because of the technical difficulties encountered in keeping pace with the increasing network speed and the more widespread use of encrypted traffic. Other researchers (including the author of this dissertation) have advocated an approach where highly distributed network based sensors are located at the periphery of computer networks; the idea being that the traffic load is more manageable there.

Even though both of these approaches above have their strengths, the previous section introduced scenarios where analysis of network traffic on a single high-speed link still represents a fundamental need (e.g. port scan detection). The commercial world attempted to respond to this need and a number of vendors now claim to have sensors that can operate on high-speed ATM or Gigabit Ethernet links. For example, ISS [43] offers NetICE Gigabit Sentry, a system that is designed to monitor traffic on high-speed network connections. The company advertises the system as being capable of performing protocol reassembly and analysis for several application-level protocols (e.g. HTTP, SMTP, POP) to identify malicious activities. The tool claims to be the 'first network-IDS that can handle full Gigabit speeds'. However, the authors of the tool also state that Gigabit Sentry handles a full gigabit in lab conditions, but real-world performance will likely be less. '[...] Customers should expect at least 300 Mbps real-world performance, and probably more depending up the nature of their traffic. [...] Gigabit Sentry can only capture slightly more than 500,000-packets/second'. These comments show the actual difficulties of performing network based intrusion detection on high-speed links. Other IDS vendors such as Cisco [22] offer comparable products with similar features. Unfortunately, no experimental data gathered on real networks is presented. TopLayer Networks [96] introduces a switch that keeps track of application-level sessions. The network traffic is split with regard to these sessions and forwarded to several intrusion detection sensors. Packets that belong to the same session are sent through the same link. This allows sensors to detect multiple steps of an attack within a single session. Unfortunately, the correlation of information between different sessions is not supported. This could result in missed attacks when attacks are performed against multiple hosts (e.g. ping sweeps) or across multiple sessions.

Very few research papers have been published that deal with the problem of high-speed intrusion detection. Sekar et al. [83] describe an approach for high-performance analysis of network data, but unfortunately they do not provide experimental data based on live traffic analysis. Their claim of being able to perform real-time intrusion detection at 500 Mbps is based on the processing of offline traffic log files. This estimate is not indicative of the real effectiveness of the system when operating on live traffic.

6.3 System Design

The problem of intrusion detection analysis in high-speed networks can be effectively addressed only if a scalable solution with respect to increasing network speeds is available. Let us consider the traffic on the monitored network link as a bi-directional stream of link-layer frames (e.g. `Ethernet` frames). This stream contains too much data to be processed in real-time by a centralized entity and has to be divided into several smaller streams that are fed into a number of different, distributed sensors. Each sensor is only responsible for a subset of all detectable intrusion scenarios and can therefore manage to process the incoming volume in real-time. Nevertheless, the division into streams has to be done in a way that provides each sensor with enough information to detect exactly the same attacks that it would have witnessed when operating directly on the network link.

6.3.1 Requirements

The overall goal is to perform stateful intrusion detection analysis in high-speed networks. The approach presented in this chapter can be characterized by the following requirements.

- The system implements a misuse detection approach where *signatures* representing attack scenarios are matched against a stream of network events.
- Intrusion detection is performed by a set of sensors, each of which is responsible for the detection of a subset of the signatures.
- Each sensor is autonomous and does not interact with other sensors.
- The system partitions the analyzed event stream into slices of manageable size.
- Each traffic slice is analyzed by a subset of the intrusion detection sensors.
- The system guarantees that the partitioning of traffic maintains detection of all the specified attack scenarios. This implies that sensors, signatures, and traffic slices are configured so that each sensor has access to the traffic necessary to detect the signatures that have been assigned to it.
- Components can be added to the system to achieve higher throughput. More precisely, the approach should result in a scalable design where one can add components as needed to match increased network throughput.

6.3.2 System Architecture

The requirements listed in the previous section have been used as the basis for the design of a network based intrusion detection system. The system consists of a *network tap*, a *traffic scatterer*, a set of m *traffic slicers* $S_0, ..., S_{m-1}$, a *switch*, a set of n *stream reassemblers* $R_0, ..., R_{n-1}$, and a set of p *intrusion detection sensors* $I_0, ..., I_{p-1}$. A high-level illustration of the architecture is shown in Figure 6.1.

The network tap component monitors the traffic stream on a high-speed link. Its task is to extract the sequence F of link-layer frames $\langle f_0, f_1, ..., f_t \rangle$ that are observable on the wire

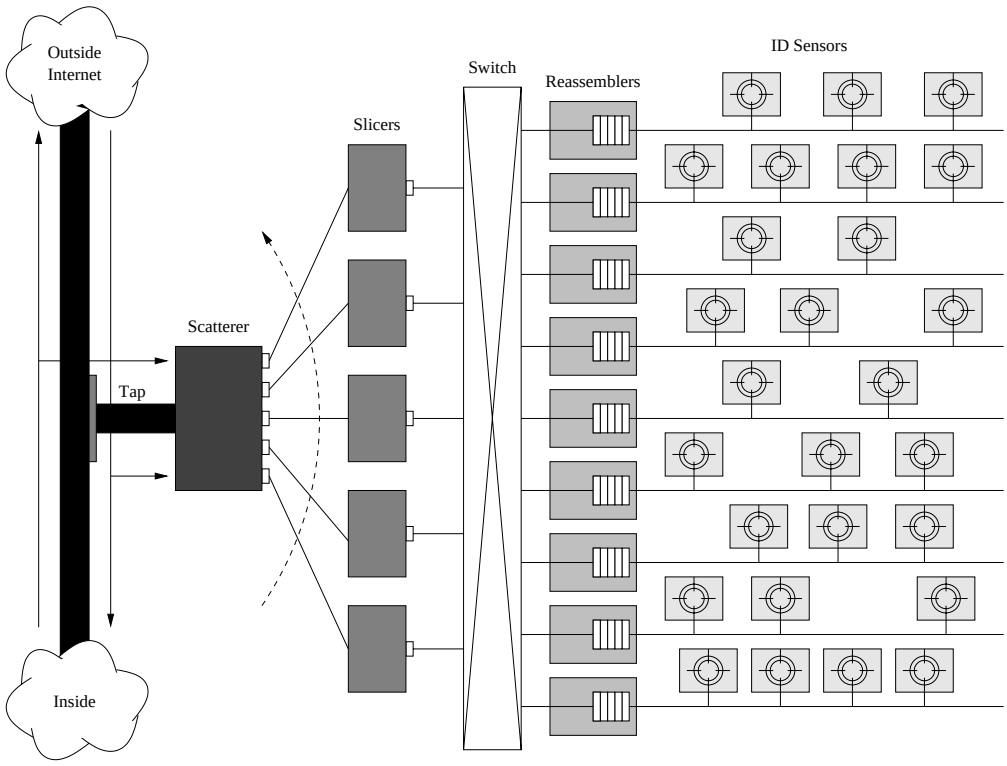

Figure 6.1: Architecture of High-Speed Intrusion Detection System

during a time period Δ. This sequence of frames is passed to the scatterer which partitions F into m sub-sequences $F_j : 0 \leq j < m$. Each F_j contains a (possibly empty) subset of the frame sequence F. Every frame f_i is an element of exactly one sub-sequence F_j and therefore $\cup_{j=0}^{j<m} F_j = F$. The scatterer can use any algorithm to partition F. Hereafter, it is assumed that the splitting algorithm simply cycles over the m sub-sequences in a round-robin fashion, assigning f_i to $F_{i \bmod(m)}$. As a result, each F_j contains an m-th of the total traffic.

Each sub-sequence F_j is transmitted to a different traffic slicer S_j. The task of the traffic slicers is to route the frames they receive to the sensors that may need them to detect an attack. This task is not performed by the scatterer, because frame routing may be complex, requiring a substantial amount of time, while the scatterer has to keep up with the high traffic throughput and can only perform very limited processing per frame.

The traffic slicers are connected to a switch component, which allows a slicer to send a frame to one or more of n outgoing channels C_i. The set of frames sent to a channel is denoted by FC_i. Each channel C_i is associated with a stream reassembler component R_i and a number of intrusion detection sensors. The set of sensors associated with channel C_i is denoted by IC_i. All the sensors that are associated with a channel are able to access all the packets sent on that channel. The original order of two packets could be lost if the two frames took different paths over distinct slicers to the same channel. Therefore, the reassemblers associated with each channel make sure that the packets appear on the

channel in the same order that they have appeared on the high-speed link. That is, each reassembler R_i must make sure that for each pair of frames $f_j, f_k \in FC_i$ it holds that (f_j before f_k) $\iff j < k$.

Each sensor component I_j is associated with q different attack scenarios $A_j = \{A_{j0}, .., A_{jq-1}\}$. Every attack scenario A_{jk} has an associated *event space* E_{jk}. The event space specifies which frames are candidates to be part of the manifestation of the attack. For example, consider an attack targeting a web server called `spider` within the network protected by the intrusion detection system. In this case, the event space for that attack is composed of all the `TCP` traffic that involves port 80 on host `spider`.

Event spaces are expressed as disjunctions of *clauses*, that is, $E_{jk} = c_{jk_0} \vee c_{jk_1} \vee ... \vee c_{jk_n}$, where each clause c_{jk} is an expression of the type xRy. x denotes a value derived from the frame f_i (e.g., a part of the frame header) while R specifies an arithmetic relation (e.g., =, !=, <). y can be a constant, the value of a variable or a value derived from the same frame. Clauses and event spaces may be derived automatically from the attack descriptions, for example, from signatures written in attack languages such as `Bro` [69], `Sutekh` [75], `STATL` [31] or `Snort` [80] as well as our *Event Description Language*.

6.3.3 Frame Routing

Event spaces are the basis for the definition of the filters used by the slicers to route frames to different channels. The filters are determined by composing the event spaces associated with all the scenarios that are 'active' on a specific channel. More precisely, the set of active scenarios is $AC_i = U_{j=0}^{j<u} A_j$ where A_j is the set of scenarios of $I_j \in IC_i$. The event space EC_i for a channel C_i is the disjunction of the event spaces of all active scenarios, which corresponds to the disjunction of all the clauses of all the active scenarios. The resulting overall expression is the filter that each slicer uses to determine if a frame has to be routed to that specific channel. Note that it is possible that a certain frame could be needed by more than one scenario. Therefore, it will be sent on more than one channel.

The configuration of the slicers as described above is static; that is, it is calculated offline before the system is started. The static approach suffers from the possibility that, depending on the type of traffic, a large percentage of the network packets could be forwarded to a single channel. This would result in the overloading of sensors attached to that channel. The static configuration also makes it impossible to predict the exact number of sensors that are necessary to deal with a `Gigabit` link. The load on each sensor depends on the scenarios used and the actual traffic. The minimum requirement for the slicers is that the capacity of their incoming and outgoing links must be at least equal to the bandwidth of the monitored link.

One way to prevent the overloading condition is to perform dynamic load balancing. This is done by reassigning scenarios to different channels at run-time. This variant obviously implies the need to reconfigure the filter mechanism at the traffic slicers and update the assignment of clauses to channels.

In addition to the reassignment of whole scenarios to different channels, it is also possible to split a single scenario into two or more *refined scenarios*. The idea is that each

refined scenario catches only a subset of the attacks that the original scenario covered, but each can be deployed on a different channel. Obviously, the union of attacks detectable by all refined scenarios has to cover exactly the same set of attacks as the original scenario did. This can be done by creating additional constraints on certain attributes of one or more basic events. Each constraint limits the number of attacks a refined scenario can detect. The constraints have to be chosen in a way such that every possible value for a certain attribute (of the original scenario) is allowed by the constraint of at least one refined scenario. Then, the set of all refined scenarios, which each cover only a subset of the attacks of the original one, are capable of detecting the same attacks as the original.

A simple way to partition a particular scenario is to include a constraint on the destination attribute of each basic event that represents a packet which is sent by the attacker. One has to partition the set of possible destinations such that each refined scenario only covers attacks against a certain range of hosts. When the union of these target host ranges covers all possible attack targets, the set of refined scenarios is capable of finding the same attacks as the original scenario. Such an approach is necessary when a single scenario causes too much traffic to be forwarded to a single channel.

In addition, obviously innocent or hostile frames could be filtered out before the scenario clauses are applied, thereby eliminating traffic that needs no further processing. This could be used, for instance, to prevent the system from being flooded by packets from distributed denial-of-service slaves that produce traffic with a unique, known signature.

6.4 Prototype Architecture

The initial set of experiments were primarily aimed at evaluating the effectiveness of the scatterer/slicer/ reassembler architecture. For these experiments, we deployed three traffic slicers ($m = 3$) and four stream reassemblers ($n = 4$) with one intrusion detection sensor per stream. The next section presents the details of the hardware and software used to realize the initial prototype while section 8.2 gives the details of the experiments we performed and presents the corresponding results.

The prototype is composed of a number of hosts responsible for the analysis of the traffic carried by a `Gigabit` link.

The `Gigabit` link is realized as a direct connection (crossover cable) between two machines equipped with `Intel Xeon` 1.7 GHz processors, 512 MB RAM and 64-bit PCI `3Com 996-T Gigabit Ethernet` cards running `Linux 2.4.2` (`Red Hat 7.1`). One of the two machines simulates the network tap and is responsible for creating the network traffic (via `tcpreplay` [98]). The other machine acts as the traffic scatterer and is equipped with three additional 100 Mbps `3Com 905C-TX Ethernet` cards.

The scatterer functionality itself is realized as a kernel module attached to the `Linux` kernel bridge interface. The bridge interface provides a hook that allows the kernel to inspect the incoming frames before they are forwarded to the network layer (e.g. the `IP` stack). The scatterer module intercepts frames coming from the `Gigabit` interface and immediately forwards them to one of the outgoing links through the corresponding `Fast`

Ethernet card. The links are selected in a round-robin fashion. The scatterer also attaches a sequence number to each packet, which is later used by the reassemblers. In order to overcome the problem of splitting Ethernet frames with a length close to the maximum transferable unit (MTU), the sequence number has to be integrated into the Ethernet frame without increasing its size. To leave the data portion untouched, we decided to modify the Ethernet header. We also aimed to limit the modifications of the Ethernet frame to a minimum in order to be able to reuse existing hardware (such as network interface cards and network drivers). Therefore, the MTU had to remain unchanged. For this reason, we decided to use the six-byte Ethernet source address field for sequence numbers. As a result, before the traffic scatterer forwards a frame, it writes the current sequence number into the source address field and increments it.

The experimental setup demonstrates that the partitioning of traffic is possible and that it allows for the detailed analysis of higher traffic volume (including defragmentation, stream reassembly, and content analysis). Because we only use three traffic slicers (with an aggregated bandwidth of 300 Mbps), sustained incoming traffic of 1 Gbps would overload our experimental setup. However, the introduction of additional traffic slicers would allow us to handle higher traffic inputs.

The traffic slicers (Intel Pentium 4 1.5 GHz, 256 MB RAM, 3Com 905C-TX Fast Ethernet cards running Linux 2.4.2 - Redhat 7.1) have the NIC of the link that connects them to the traffic scatterer set to promiscuous mode in order to receive all incoming frames. The data portion of each incoming frame is matched against the clauses stored for each channel. Whenever a clause for a channel is satisfied, a copy of the frame is forwarded to that channel. Note that this could (and usually does) increase the total number of frames that have to be processed by the intrusion detection sensors. Nevertheless, a sufficiently large number of sensors combined with sophisticated partitioning enable one to keep the amount of traffic at each sensor low enough to handle. In our test setup, the partitioning (i.e. the clauses) was determined as follows. Similar to Snort [81], we distinguished between an inside network and an outside network, representing the range of IP addresses of the protected network and its complement, respectively. The protected network address range is divided according to the existing class C subnetworks. The network addresses are then grouped into four sets, each of which is assigned to a different channel. This partitioning allows the system to detect both attacks involving a single host and attacks spanning a subnetwork. As explained in Section 6.3.3, more sophisticated schemes are possible by analyzing additional information in the packet headers or even by examining the frame payload.

Once the filters have been configured, the frames have to be routed to the various channels. As in the case for the transmission between the scatterer and the traffic slicers, we want to prevent frames from being split when sent to the channels. This makes it necessary to include the destination address information of the intended channel in the Ethernet frame itself without increasing its size and without modifying the payload. To do this, we use the link layer destination address. Therefore, the destination address is rewritten with values 00:00:00:00:00:01, 00:00:00:00:00:02, etc. depending on the destination channel. There were two reasons for using a generic link number instead of the

actual `Ethernet` addresses as the target address for sensors. First, a number of sensors may be deployed on each channel, processing portions of the traffic in parallel. Since each sensor has to receive all packets on the channel where it is attached, selecting the `Ethernet` address of a single sensor is not beneficial. Second, whenever the NIC of a sensor has to be replaced, the new `Ethernet` address would have to be updated at each traffic slicer. In order to save this overhead, each traffic slicer simply writes the channel number into the target address field of outgoing frames.

The actual frame routing is performed by a switch (a `Cisco Catalyst 3500XL`) that connects traffic slicers with reassemblers. The MAC address-port table of the switch holds the static associations between the channel numbers (i.e. the target `Ethernet` addresses set by the traffic slicers) and the corresponding outgoing ports. In general, backplanes of switches have very high bandwidth compared to `Ethernet` links, so they are not likely to be overloaded by traffic generated by the scatterer.

In our setup, the stream reassemblers are located at each sensor node (using the same equipment as the traffic slicers), and they provide the intrusion detection sensors with a temporally sorted sequence of frames by using the encapsulated sequence numbers. The reassembly procedure has been integrated into `libpcap` so that every sensor that utilizes these routines to capture packets can be run unmodified. For each frame, we assume that no other frame with a smaller sequence number can arrive after a certain time span (currently 500 ms). This means that when an out-of-order packet is received, it is temporarily stored in a queue until either the missing packets are received and the correctly-ordered batch of packets is passed to the application or the reassembler decides that some packets have been lost because a timeout expired and the packet is passed without further delay. Therefore, each received packet is passed to the sensors with a worst case delay that is equal to the timeout value. The timeout parameter has to be large enough to prevent the situation where packets with smaller sequence numbers arrive after subsequent frames have already been processed but small enough so that the reaction lag of the system is within acceptable limits. Since the processing and transmission of frames is usually very fast and no retransmission or acknowledgments are utilized, one can expect frames to arrive at each reassembler in the correct order most of the time. In principle, this allows one to safely choose a very short time span. We expect to have no problems in reducing the current timeout value, but at the moment we have no experimental evaluation of the effect of different timeout values on the effectiveness of intrusion detection.

The network cards of the nodes would normally be receiving traffic at rates close to their maximum capacity. If administrative connections, such as dynamically setting clauses, reporting alarms, or performing maintenance work were to go through the same interfaces, these connections could potentially suffer from packet loss and long delays. To overcome this problem, each machine is connected to a second dedicated network that provides a safe medium to perform the tasks mentioned above. An additional communication channel decoupled from the input path has the additional benefit of increasing the resiliency of the system against denial-of-service attacks. That is, alarms and reconfiguration commands still reach all intended receivers since they do not have to compete against the flood of incoming packets for network access.

6.5 Summary

This chapter presents an alternative approach to perform intrusion detection and event correlation at a single, high-speed link. This enables our intrusion detection system to cope with attack scenarios (such as port scans) that include distributed occurrences which do not manifest themselves as connection patterns between nodes of the network.

As the main goal of our system is the applicability for large enterprise networks, it is important to design the centralized approach in a scalable way that can deal with traffic on very high speed links. This is realized by a slicing mechanism that partitions the network traffic into portions that are manageable by single sensors. The partitioning is done in a way that makes sure that every probe receives all the information it requires to detect its assigned scenarios.

The performance of the prototype, which is presented in this chapter, is evaluated in Section 8.2.

Chapter 7

Adaptive Sensors

> Adapt or perish, now as ever, is nature's inexorable imperative.
>
> – H. G. Wells

This chapter describes the second cornerstone of *Network Alertness*, the design of an adaptive anomaly sensor [53].

As explained in Section 3.5, almost all network based anomaly detectors work with underlying traffic models and monitor the flow of packets, i.e. bursts or complete sessions. The source and destination `IP` addresses and ports are used to determine parameters such as the number of total connection arrivals in a certain period of time, the inter-arrival time between packets or the number of packets to/from a certain machine. These parameters can be used to detect port scans or denial-of-service attempts. No current sensor is based on a sufficiently powerful application model that would allow analysis of single packets. We propose a novel approach that includes the packet payload as well. To circumvent the problem of encrypted payload, the sensors have to be installed at the same host as the service (or the application) that is receiving the packets. This allows the host to decrypt the payload before it is analyzed by our probes.

To be able to react selectively to packets sent form a certain source, it is necessary to be able to modify the threshold that separates normal from malicious behavior for every single packet. This includes the analysis of the packet's payload. The following sections introduce the design and implementation of our proposed network based sensor that is capable of performing anomaly based detection on the contents of packets.

The extension of the detection from protocol headers to the payload has the additional benefit of having the possibility to find Remote-to-Local attacks (R2L). The term Remote-to-Local attack has been coined during the `DARPA` sponsored `MIT Lincoln Labs` intrusion detection evaluation [56] and specifies intrusion attempts from remote users with the aim of getting unauthorized local access to the target host (typically with `root` privileges). Such attacks usually exploit a vulnerability of a service at the target machine. This is done by sending invalid input which causes a buffer overflow or an input validation error in the code running the service. The attacker sends one (or a few) carefully crafted packets including

shell-code which is executed at the remote machine on behalf of the attacker to elevate his privileges. As the intruder only has to send very few packets (most of the time a single one is sufficient), it is nearly impossible for systems that use traffic models to detect such anomalies. Our sensor combines the capability of adapting to specific attack sources with the potential to detect an important class of intrusions usually missed by network based anomaly systems.

The table below shows the results of the top three intrusion detection systems for the four different intrusion classes used in the DARPA evaluation.

Detection Rates	Scans	DOS	U2R	R2L
Known Attacks	92%	80%	64%	80%
New Attacks	88%	22%	66%	8%

Table 7.1: DARPA Intrusion Detection Evaluation Results

The figure shows that less than 10% of new R2L intrusion attempts have been detected. Known attacks describe intrusions that were used during the preparation phase of the evaluation and were known to the developers of the participating systems. New attacks describe intrusions that have been added for the actual evaluation. Therefore, the numbers for the new attacks are more significant in determining the quality of IDSs. While probes or DOS attacks provide the attacker with additional information or degrade the performance of target machines, only R2L and U2R (User-to-Root)[1] intrusions actually compromise a machine. Therefore, it is important to reliably detect such attacks.

7.1 Sensor Design

The idea of our adaptive anomaly detection is to extend current simple application models from considering only packet header information at the network and transport layer (i.e. TCP flags in an application context) to include the application payload as well. Unfortunately, the payload of IP packets observed at a network usually varies dramatically. When the entirety of all IP packets is considered, one can usually deduce only very little information that can be used for statistical reasoning. Therefore, we cannot process the payload of packets without some knowledge of the application that created them. This makes it necessary to partition the network traffic and independently analyze packets sent by different applications. By concentrating only on one type of traffic, statistical data with lesser variance can be collected. This allows our probe to establish a notion of 'normal traffic' for each service. Our sensors operate in a service specific setup where different measurements for different application protocols are applied.

[1] User-to-Root attacks specify intrusions where a *local user* unauthorized elevates his privileges (often to obtain root permissions). Such attacks are not directly visible on the network.

At first glance, the vast number of different protocols seems to make our approach undesirable. Nevertheless, one should consider that it is not necessary to change the basic detection code for each new service. We use a generic backend that is responsible for the actual statistical anomaly detection. Service specific front-ends extract data from the network and transform it into a format suitable for the back-end. Additionally, only a few services need to be publicly accessible. By monitoring `HTTP`, `DNS`, `SMTP`, `IMAP`, `POP` and `FTP` traffic, most of the protocols that have to be available for use by anonymous clients from the Internet are covered. The first prototype that we have developed is currently able to check `DNS` traffic.

As stated above, anomaly detection systems detect intrusions by comparing observed behavior to expected behavior using certain metrics that are defined by the underlying model. The expected behavior (called *profile*) has to be defined by the user or can be automatically deduced during the *training period*. Because manual creation of expected behavior is cumbersome and error prone, most systems extract profiles from training data. It is important to point out the difference between approaches that require classified data during the training period [59] (i.e. data samples that are marked explicitly as normal or malicious) to build their models and those that do not [74]. Systems that are based on classified samples extract features that basically allow the data to be clustered into a normal and a malicious group. New data is then analyzed according to these selected features and mapped into one of the sets (obviously raising an alarm when it is classified as malicious). This approach has the drawback that classified data is rarely available in new environments where the IDS has to be deployed. Therefore, these systems have to utilize general models extracted from existing sample data. In changing environments, such designs may produce many false alarms or miss actual intrusion attempts. We follow the second approach that uses observed, unclassified traffic from the place where the system is installed to form a model of 'normal' behavior. Any traffic that deviates from that model is considered hostile.

Our module uses an initial, user definable training period during which it reads packets from the network. As stated above, this data is split into service specific traffic and forms the input to build our profiles. After that initial phase, our sensor switches to detection mode in which the new traffic is compared to our application model to detect anomalies. When the environment changes dramatically resulting in too many false alarms, it is simple to update the application model by rerunning the training phase on the changed traffic.

The following section describes the features of network traffic that we use to build our profile and the metrics that is used to determine the deviation of actual, observed traffic.

7.2 Packet Processing

The task of the packet processing step is to read the network traffic and extract suitable input data for the following statistical processing step. It is implemented in a single module called *Packet Processing Unit (PPU)*. Our anomaly detection is based on the analysis of the payload that is passed as input to the different network services. This implies that

we cannot directly operate on the packet level itself as an attacker can easily distribute his malicious payload over several datagrams. Tools such as `fragrouter` [87] help to split and send data in several `IP` fragments or `TCP` packets. Therefore the statistical processing receives a *service request* as the basis for its analysis.

A service request is the user supplied data which is sent over the network to a certain service to perform a single task on behalf of that user. Usually, a network service is implemented as a daemon operating in an interactive mode that waits for incoming connections on a well-known port. In order to utilize its services, one has to open a connection to the daemon port and provide input data. The input data basically consist of the exact type of the desired service and additional parameters. The daemon parses the supplied information, processes it and returns the results. It then terminates the connection or awaits further input. We call the user supplied input sent to the daemon in such a single transaction a service request. Depending on the service and its exact type, requests can have different formats and layouts.

For a `HTTP` request to a web server, the type of service can be `GET`, `HEAD` or `POST` and contains parameters such as the URL or the type of browser the user runs. In case of `DNS`, a service request is usually a single packet that contains the `DNS` name which should be resolved or an `IP` address that needs to be mapped to a `DNS` name.

The PPU has to extract service requests from the stream of packets on the wire to pass them to the statistical processing step. The packet processing takes place in two stages. The first one is service independent and performs generic `IP` and `TCP` stream reassembling. It passes complete `UDP` packets or acknowledged segments of a `TCP` stream to the second stage. This second stage needs service specific knowledge and has to extract single service requests from its input. It therefore requires basic understanding of the layout of requests although in most cases, only very limited information is necessary. The end of a request can usually be determined by watching for an end-of-request character or character sequence (e.g. two `CTRL-LF` characters in a `HTTP` request) or by checking a length field in the request header (e.g. the number of fixed size `resource records` in a `DNS` query). The type of service is also easily determined by a string (e.g. `GET` in case of `HTTP`) or a header value (e.g. query type and class in case of `DNS`). The assignment of a type to a certain request is not enforced by the service protocol itself. It is possible to assign different types to requests that are considered equal by the service protocol (and by the daemon implementing the service). In the case of a `HTTP` request, the fact that it is a `GET` request may be used to perform a more detailed analysis on the URL (e.g. one can assign different types to requests that invoke `cgi`-programs or `php`-scripts). The complete service request together with its type then enters the statistical processing phase.

7.3 Statistical Processing

The statistical processing step is realized by a module called *Statistical Processing Unit (SPU)*. The SPU is only considered with requests and their types. As stated above, the statistical properties of requests for different services can be very diverse. But even different

types of requests for a single service can vary significantly. In case of `HTTP` traffic, standard `GET` requests look very similar but a `POST` request that transmits lots of data keyed in by a user into a `HTML` form may be different. Therefore, requests are divided into several groups where each group contains requests of types with similar statistical properties. The requests of each group are then analyzed independently. Currently, the division of requests has to be done manually by the developer who adds a new service (protocol) to the IDS. We plan to develop a metrics that allows our sensor to automatically find similar properties of different request types after the initial setup period and group them accordingly. Nevertheless, most requests for a certain service are very similar and a reasonable starting point is to divide all requests into groups according to the associated network service.

The following properties of a request are used to determine its anomaly score.

1. Type of Request
2. Length of Request
3. Payload Distribution

The anomaly score is compared to a threshold that can be manually set by the security administrator. It should initially be set to a value where no more than ten false alarms are reported per day[2]. In Section 8.3 we show that this level allows our sensor to detect a significant majority of attacks. While the system is operating and hostile patterns emerge, *Quicksand* response components alter the threshold to be more sensitive to service requests from suspicious sources.

The anomaly score of a service request is the weighted sum of the three scores computed for each of the three properties enumerated above. A number of constant factors have been introduced into the formulas that are discussed in the next few sections. Most of them are used to force the scores calculated for each of the three properties above into the same order of magnitude and could be changed (scaled appropriately) without affecting our sensor. A few, which have been determined empirically, are needed to reflect our considerations accordingly.

7.3.1 Type of Request

Including the type of request as a property in calculating the anomaly score has the following rationale. Often, exploits that are based on buffer overflows or input validation errors use a feature of a network service that is rarely or infrequently requested by users. Basic services are usually well understood and have been used extensively over a long period of time. An extra feature that has been added for a very specific purpose or very recently is often less understood and has not been exposed to such a large number of input data (and test cases). Therefore, it is more likely that the implementation of these features contains security flaws that can be exploited. The recent, well-known attacks against the

[2]Ten false alarms per day are considered to produce an acceptable low noise level that allows a system to be used in a production environment.

`bind` implementation of the domain name service (`DNS`) are not targeted against the standard name translation routines but against the code that handles `NXT` (`next record`) or `TSIG` (`transaction signature`) [16, 17] queries. When only half of all Remote-to-Local exploits abuse infrequently used features, the probability that such a request contains malicious payload is much higher than that of a regular one.

We therefore assign higher anomaly scores to requests that are of types that occur less frequent. The anomaly score (AS) is calculated as follows

$$AS_{type} = -\log_2(p[typ])$$

`p[typ]` is given as the probability that a certain request is of type `typ`. This probability is equal to the relative frequency that a request with type `typ` has occurred during the training period. In order to prevent too high anomaly scores (or even infinite values) for request types that occur very infrequent (or not at all during the training period) the probability of each request type is set to be at least $3.05 * 10^{-5}$ (yielding a maximal anomaly score of 15.0).

7.3.2 Length of Request

The length of a request can be a good indicator for the correctness of its content. Usually, a service request consists of some protocol specific information and user (or user program) supplied input. The length of the protocol information does not vary much between requests of a certain type. The user supplied input mostly consists of a few, short strings (e.g. a domain name or URL) in human readable form and does not cause much variation in the total length either. The situation looks different when requests carry input to overflow a buffer in the target service. It is necessary to ship the shell-code itself (which has a typical length of a few hundred bytes) and additional padding that depends on the length of the buffer which is targeted. Instead of a short URL or a simple domain name, the user supplied part contains several hundred bytes. This obviously increases the total length of the request. The anomaly score is calculated by using the mean (μ) and the standard deviation (σ) of the lengths of the requests that have been monitored during the training period. The following formula is used for a request with the length l. The anomaly score grows exponentially as the request length increases. In order to tolerate a reasonable amount of deviation of the length values, we use 1.5 as the base and multiply σ with a constant factor of 2.5.

$$AS_{len} = 1.5^{\frac{(l-\mu)}{2.5*\sigma}}$$

This formula has the property that it assigns anomaly scores greater than 1.5 (the base of the exponential function) only to requests that are longer than the average. This is consistent with our assumption that malicious payload increases the total length. The maximum value of AS_{len} is limited to 15.0.

7.3.3 Payload Distribution

The biggest advantage in extending the application model to consider the payload of requests is the possibility to analyze it for the occurrence of abnormal content. As we do not intend to do signature based analysis we have to build a model of a 'normal' payload. Our model is based on the observation that requests mainly contain printable characters and human readable strings. For example, a `HTTP` request consists of several plain text lines and `DNS` queries contain domain names as strings. This implies that a large percentage of characters in such requests are drawn from a small subset of all 256 possibilities (`ASCII` values for letters, numbers and a few special characters). Like in English text, those characters are not uniformly distributed but occur with different frequencies. Obviously, we cannot expect that the frequency distribution is identical to a standard text. Even the frequency of a certain character (e.g. the frequency of letter `'e'`) varies tremendously between requests. Nevertheless, there are similarities between the character frequencies in different service queries. These become apparent when the relative frequencies of all possible 256 characters are sorted in descending order (obviously, many will be 0 for a typical request). Our payload analysis is only based on the frequency values themselves and it does not matter whether the character with the most occurrences is an `'e'` or a `'.'`. We call the sorted, relative character frequencies of a request its *character distribution*.

Consider the text string '`Aaaza`' with the corresponding `ASCII` byte values '65 97 97 122 97'. The left diagram of Figure 7.1 shows the absolute occurrences of the bytes that are contained in the string above. The right diagram displays the sorted, relative frequencies (i.e. character distribution) which have been calculated from the absolute values represented as a *histogram*.

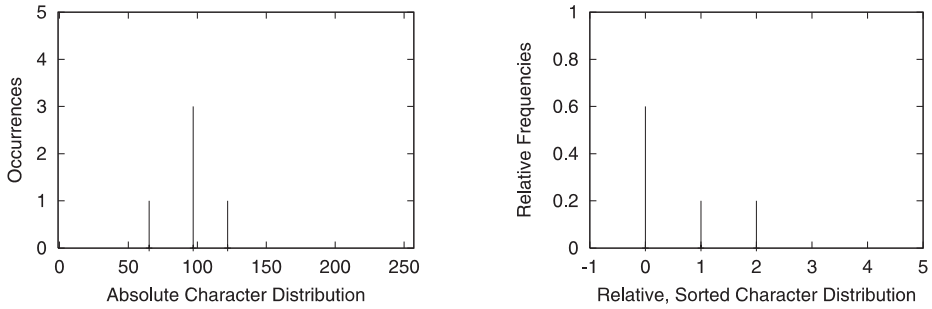

Figure 7.1: Character Distributions

For the payload of a regular request, one can expect that the relative frequencies slowly decrease in value when one moves in the direction of the positive x-axis. In case of abnormal payload the frequencies can drop extremely fast (because of a peak caused by a very high frequency of a single character) or barely (in case of a nearly uniform character distribution).

The character distribution of a perfect normal packet is called *payload distribution (PD)*. The payload distribution is a discrete distribution with

$\mathcal{PD} : \mathfrak{D} \mapsto \mathfrak{P}$ with $\mathfrak{D} = \{n \in \mathcal{N} | 0 \leq n \leq 255\}$ and $\mathfrak{P} = \{p \in \Re | 0 \leq p \leq 1\}$

The relative frequency of the character that occurs n-most often (0-most denoting the maximum) is given as $\mathcal{PD}(n)$. When the histogram in Figure 7.1 is interpreted as a payload distribution then $\mathcal{PD}(0) = 0.6$ and $\mathcal{PD}(1) = 0.2$.

The payload distribution is calculated during the training period. In this period, the SPU stores the character distributions of all received requests. The payload distribution is then approximated as the mean of all character distributions. This is done by setting $\mathcal{PD}(n)$ to the mean of the frequencies for the n-most frequent character of all requests. As we have operated on relative frequencies that sum up to 1.0, the means will do so as well (making the payload distribution well-defined).

For each request received in detection mode, we assume that the character distribution is a sample drawn from the payload distribution. We use a statistical test to determine the likelihood that the sample is really derived from the payload distribution. For a normal request the test should yield a high confidence in the correctness of this hypothesis while it should be rejected for malicious payload. We use a variant of the `Pearson` χ^2-test as our 'goodness-of-fit' test.

For our intended statistical calculations, it is not necessary to operate on all values of $\mathcal{PD}$ directly. Instead, it is enough to consider a small number of intervals. Therefore, we divide the domain of $\mathcal{PD}$ into a total of six segments (as shown in the table below).

Segment	0	1	2	3	4	5
x-Values	0	1-3	4-6	7-11	12-15	16-255

Table 7.2: $\mathcal{PD}$-Intervals for χ^2-Test

The expected relative frequency of characters in a segment can be easily determined by adding the values of $\mathcal{PD}$ for the corresponding x-values. As the relative frequencies are sorted in descending order we expect the values of $\mathcal{PD}(n)$ to be more significant for our anomaly score when n is small. This fact is clearly reflected in the division of $\mathcal{PD}$'s domain.

When a new request arrives, the absolute number of occurrences for each character is determined. Afterwards, these values are sorted in descending order and combined according to the table above (aggregating values that belong to the same segment). The χ^2-test is utilized to calculate the probability that the given sample (derived from this request) has been drawn form the payload distribution. The standard test requires the following steps to be performed.

1. *Calculate the observed and expected frequencies* - The observed values O_i (one for each segment) are already given and the expected number of occurrences E_i are calculated by multiplying the relative frequencies for each of the six segments with the length of the request.
2. *Compute the χ^2-value* as $\chi^2 = \sum_{i=0}^{i<6} \frac{(O_i - E_i)^2}{E_i}$

3. *Determine the degrees of freedom and obtain the significance* - The degrees of freedom for the χ^2-test are identical to the number of addends in the formula above minus 1. This yields 5 in our case. The actual probability that the sample is derived from the payload distribution (i.e. significance of the sample) is read from a pre-defined table using the χ^2-value as index.

The χ^2-values themselves increase as the likelihood that the sample stems from the payload distribution decreases. Therefore, it is not necessary to first perform the table lookup in step 3 and then transform the probability back into an anomaly score. The χ^2-value can be used directly for the computation of the score. As stated above, the test operates on absolute values. This results in higher absolute χ^2-values for longer packets than for short ones even though they have the same relative deviation from the payload distribution. As the packet length is already factored into our anomaly score, we divide the χ^2-value by the payload length `l` to get a length independent result. This result is then multiplied by a constant factor of 15.0 (the maximum used for both other properties) to get scaled to the correct order of magnitude. The maximum is set to 20.0 (in contrast to 15.0 to reflect the higher importance of this property).

$AS_{pd} = \chi^2 * \frac{15}{l}$

This approach is very efficient. In the following Section 8.3, we show by means of experimental data that this method is able to distinguish between normal and malicious requests for certain applications. It has the additional advantage compared to signature based systems that it cannot be fooled by some well known attempts to hide the attacker's payload. Signature based systems often contain rules that cause an alarm when long sequences of 0x90 bytes (`nop` operation of `Intel x86` [41] based architectures) are detected in a packet. As a consequence, attackers substitute such sequences with assembler instructions that act similar (e.g. `add rA, rA, 0` - which adds 0 to the value in register A and stores the result back to A). This prevents signature based systems from detecting such attacks. In our case, such sequences still cause a distortion of the request's character distribution and result in high anomaly scores. In addition, characters in malicious payload are often `XOR`'ed with constants or shifted by a fix value (`ROT-13 code`). Such evasion attempts do not change the resulting character distribution and the anomaly score remains the same.

7.4 Anomaly Score

The anomaly score is a value that specifies the extent of the deviation of the received request from the expected values specified by the profile. It is a compound value derived from the factors that have been described above and is calculated as follows.

$AS = 0.3 * AS_{type} + 0.3 * AS_{len} + 0.4 * AS_{pd}$

The anomaly score for each request can be in a range from 0 to 17.00 (when each addend contributes to the sum with its maximum). The payload anomaly score has slightly more weight to reflect its importance. This score is compared to a threshold that can be chosen

by the security administrator. The default threshold is computed during the training phase and set to a value that would cause 10 false alarms per day when the sensor receives the training data itself as input. A lower threshold means that is more likely that attacks are detected with the disadvantage of an increasing number of false alarms. The limit should be set to the lowest value possible provided that the number of false alarms is manageable. This decision depends on the type of traffic that is seen on the network and a policy which decides how many false alarms are considered acceptable. The evaluation presented in Section 8.3 shows that our prototype module managed to detect all our attacks by setting the threshold to a value that produced significantly less than 10 false alarms per day during our experiments.

7.5 Summary

This chapter presents an intrusion detection sensor that uses statistical anomaly detection to find Remote-to-Local attacks targeted at essential network services. We use a service based approach that separates statistical data for requests to different services to improve our detection capability. In contrast to other systems which mainly rely on information in the network and transport layer headers (`TCP/IP`) to perform their analysis, we propose an extended application model that includes the payload as well. This allows to adapt the detection to single packets sent from specific sources instead of requiring the aggregation of packets over whole sessions.

The sensor is integrated as a plug-in into the *Quicksand* intrusion detection framework and enables correlating nodes to change the anomaly threshold when analyzing traffic from suspicious senders. The knowledge of emerging hostile patterns that is gained by the distributed detection process is utilized to modify the data gathering itself. This feedback mechanism allows nodes to tighten the analysis of packets from potentially malicious sources and realizes the envisioned behavior of *Network Alertness*.

Chapter 8

Evaluation

> The farther the experiment is from theory the closer it is to the Nobel Prize.
>
> – Frederic Joliot-Curie "A Random Walk in Science"

The aim of this chapter is to determine whether our proposed intrusion detection framework fulfills the requirements of *Network Alertness*. We evaluate the properties of the distributed correlation process, the centralized high speed detection prototype and the adaptive sensor module.

The focus of *Network Alertness* is on a system that is deployable in large networks, therefore the detection mechanisms must be scalable and fault tolerant to a large extent. Both, theoretical considerations as well as experimental data, help to support our claim that the designed framework meets these requirements. The evaluation of the sensor module concentrates on its ability to analyze the payload of single packets and the efficiency of the detection scheme.

8.1 Distributed Correlation Approach

This section evaluates the

- scalability and
- fault tolerance

properties of the proposed distributed detection processes. This makes it necessary to define the evaluation criteria that we use to quantify these properties.

We measure *fault tolerance* as the percentage of nodes of the complete network which have their events correlated after a single machine running parts of the IDS (sensor or correlator) fails or is taken out. This indicates the percentage of distributed patterns that can still be detected. When a node failure partitions the set of hosts into several subsets where events are still related within each of these subsets, the highest percentage among

all of them is chosen. When a correlator that is responsible only for a subset of all nodes fails, the remaining system may still perform event correlation on a reduced set of hosts. The fault tolerance measures exactly that fraction of nodes.

The *scalability* of distributed intrusion detection systems is characterized by two values. One indicates the total network traffic between all nodes (total traffic) while the other measures the maximum network traffic at a single node (peak traffic).

We compare our completely decentralized system (distributed approach) to a design that deploys sensors at every host and centrally collects their data (centralized approach) and to one that introduces several layers of processing nodes (hierarchical approach) on top of the sensors which forward data that might be part of a larger attack scenario to upper level sensors. An example of a centralized system is NSTAT (see Section 3.1.2), while Emerald and AAfID (Sections 3.2.2 and 3.2.3) follow a hierarchical approach.

8.1.1 Theoretical Considerations

For our theoretical discussion, we assume a network with n hosts and the occurrence of $n * e$ interesting events during a time interval of length Δ. The interval Δ also specifies when messages 'time out' and are removed from the detection process. While the number of events in the whole system is assumed to be proportional to the number of nodes, the number of events at each single host may not exceed a certain threshold τ. This is reasonable as it allows a certain variance of the distribution of events within the system (i.e. modeling local hot spots such as web or file servers in very large networks) without allowing a single node from having to deal with arbitrary many events as the number of nodes grows larger.

The coverage of a network after a single node failure is given below for the different approaches. We assume that the hierarchical system uses $l = \lfloor \log_m((m-1) * n) \rfloor$ layers with m^k nodes (k ... 0 to $l-1$) in each layer, where m specifies the number of children for each node.

System	Type of Node	Coverage
Centralized	Sensor	$\frac{n-1}{n}$
	Correlator	0
Hierarchical	Sensor	$\frac{n-1}{n}$
	Correlator (at Layer k)	$n - \frac{m^{l-k}-1}{m-1}$
	Root (Layer 0)	$\frac{n-1}{m}$
Decentralized	Node	$\frac{n-1}{n}$

Table 8.1: Fault Tolerance Properties

The loss of a node at layer k in the hierarchical model stops correlation of the complete subtree with $\frac{m^{l-k}-1}{m-1}$ nodes. When the root is lost, each subtree with $\frac{n-1}{m}$ nodes can still

do correlation. Not surprisingly, this shows that centralized and hierarchical system are more vulnerable especially to the loss of important nodes (i.e. nodes in top layers or the root itself) than our completely distributed design. We perform correlation only at nodes where the relevant events are actually observable, therefore a loss of some hosts cannot influence the detection capability of the remaining system.

The theoretical scalability values of our system depend on assumptions about the used patterns and the number of send events to different targets during the time interval Δ.

As explained in Section 4.2.3, all messages at the send node have to be copied to each new target of a send event. This results in message traffic which is proportional to the number of send events to different targets during Δ. The average number of send events at a single node during Δ is indicated as ω. Depending on the used patterns, different amounts of messages have to be copied over send event links. In the optimal case, only one message instance, representing the send event itself, has to be transmitted. When the attack scenario contains dynamic constraints between events that are separated by one or more send event link, additional messages have to be moved to the target host as well. The situation worsens when a message has to be moved over several consecutive send links as it gets copied to each target at every step (yielding potential exponential growth of the number of messages). Therefore, the depth δ of a pattern (defined as the maximal number of consecutive send links a message has to traverse) is an important factor to determine scalability of our system. Usually, not all event patterns define or use variables and messages created from those events do not need to be forwarded. We denote υ (with $\upsilon < 1$) as the fraction of pattern descriptions of an attack scenario that actually do define or use variables and result in messages that might need to be transferred over the net. When e events occur at a single node, on average only $e * \upsilon$ of them need to be transmitted over send links.

As each message only contains part of the data of the complete event object we save bandwidth in comparison to systems that have to send the whole event itself (because they do not know which information is important at the higher levels). The ratio between the event object size and the message size (including id and timestamp) is written as r.

The explanation (and notation) given above allows us to formulate the estimate of the total network traffic as

$$\frac{n * (e * \upsilon) * \omega^{\delta}}{r} \tag{8.1}$$

As each node equally participates in the detection process, the peak traffic is equal to the expected average traffic at a single node which results in

$$\frac{n * (e * \upsilon) * \omega^{\delta}}{n * r} = \frac{(e * \upsilon) * \omega^{\delta}}{r} \tag{8.2}$$

Although Equation 8.2 shows the potential for an exponential explosion of the needed network traffic, the next section will show that δ which only depends on the used patterns is usually very small for our area of application (e.g. none of our attack scenarios had a δ greater than 2). Note that the peak traffic does not depend on the total number

of nodes in the system which means that under our assumptions the system is perfectly scalable. Although the situation may be different in practice, the formula indicates good scalability properties for our design. Additionally, the factors υ and r help to keep the total bandwidth utilization reasonably low.

Table 8.2 shows the total and peak traffic values for a centralized and a hierarchical solution. We assume that each hierarchy layer is capable of reducing the events it forwards to a higher level node by a constant factor c.

System	Total Traffic	Peak Traffic
Centralized	$e * n$	$e * n$
Hierarchical	$e * n * \sum_{i=1}^{i<l} c^i$	$e * n * c^{l-1}$
Decentralized	$\frac{n*(e*\upsilon)*\omega^\delta}{r}$	$\frac{(e*\upsilon)*\omega^\delta}{r}$

Table 8.2: Scalability Properties

The total and peak traffic values for the centralized solution reflect the fact that all event data is sent to a single location. The traffic in the hierarchical system is created by nodes that forward data up to higher layers. As a fraction (determined by c) of the event data is forwarded over several levels the total traffic consists of the sum of the traffic volumes between each layer. The peak traffic occurs at the root node ($i = l - 1$). Although it is significantly smaller than in the centralized case, it still depends on the number of nodes in the system.

In both the decentralized and the hierarchical system, the total traffic volume increases when compared to a centralized design. Nevertheless, the peak traffic indicates that they scale much better than a centralized one.

8.1.2 Experimental Results

Our intrusion detection system is designed to provide a scalable solution for enterprise sized networks. Unfortunately, we do not have the resources to perform scalability tests on such a scale. A simple simulation does not seem reasonable because it would, based on our assumptions, exactly deliver the results we have derived theoretically. Therefore, we decided to perform an actual experiment but had to restrict ourselves to our department's network. We ran processes executing our detection algorithm on the web server, the `DNS` server, our firewall and six additional hosts. These machines are running `Linux 2.2.14` and `SunOS 5.5.1` on different `Pentium II`, `Athlon` and `Sparc` hosts. The idea is to gather experimental data that can be compared to values that we would expect given our theoretical considerations.

We use our anomaly sensor described in the previous Chapter 7 and the `Snort` based network sniffer to collect data from the network. We provided EDL definitions for interesting network packets (i.e. `TCP`, `UDP`, `IP` and `Ethernet`) as well as for `Snort` alerts. These are the basic building blocks for attack scenarios written in our *Attack Specification Language*.

We specified 16 different distributed patterns that aim to detect distributed signatures and anomalies with the following properties.

Property	Average	Maximum	Minimum
Pattern Depth (δ)	1.19	2.00	1.00
Fraction of Events with Variables (υ)	0.83	1.00	0.50

Table 8.3: Properties of Distributed Patterns

The given numbers in Table 8.4 are based on a week of real data collected in our network during which we processed 16374 events. We used a time interval (Δ) of 24 hours.

Property	Average	Maximum	Minimum
Events per Δ	2340	3818	1732
Send Event Targets for Single Node (during Δ)	1.62	5	0
Total Traffic (in Messages)	3922	7536	3159
Peak Traffic (in Messages)	1011	2722	744

Table 8.4: Message Traffic

An analysis of the traffic showed that our system was exposed to a number of real intrusion attempts where one was even successful (exploited a hole in our `FTP`-server). This supports our assumption that we used real traffic including actual attacks for our evaluation. As expected, our used patterns did not result in a message explosion and the total number of messages never exceeded twice the number of actual events. When one also considers that only relevant attributes (mainly 2 or 4 bytes) instead of the whole event have to be transmitted, the used bandwidth is comparable to a centralized system. The unexpected high peak traffic values resulted from many scans that included port 80 which the firewall reported to the web server. In our setup, a high fraction of the messages concentrated on two machines (web server, `DNS` server) while regular nodes transmitted fewer messages. However, an increase of nodes in our local network would not raise the load at these machines significantly (as the port scan messages were caused by machines on the Internet anyway) while producing more total traffic inside the network. In such a case (which also applies to large enterprise intranets), we expect that the ratio between the messages at these servers and messages at regular nodes decreases.

8.2 Centralized High-Speed Approach

The goal of the set of experiments described in this section is to get a preliminary evaluation of the effectiveness of our centralized sensor implementation. The general assumption is

that we are interested in in-depth, stateful, application-level analysis of high-speed traffic. For this reason, we chose `Snort` as our 'reference' sensor and we enabled reassembling and defragmentation.

8.2.1 Experimental Results

To run our experiments we used traffic produced by `MIT Lincoln Labs` as part of the `DARPA` 1999 IDS evaluation [56]. More precisely, we used the data from Tuesday of the fifth week. The traffic log was injected on the `Gigabit` link using `tcpreplay` [98]. To achieve high speed traffic we had to artificially 'speed up' the test traffic which has been recorded on a `Fast Ethernet` (an option supported by `tcpreplay`). We assumed that this would not affect the correctness of our experiment. We also assumed that the `LL/MIT` traffic is a reasonable approximation of real-world traffic. This assumption has often been debated, but we assume that for the extent of the tests below, this is reasonable.

The first experiment was to run `Snort` on the traffic log containing the test data. The results of this 'offline' run were 11,213 detections in 10 seconds with an offline throughput of 261 Mbps. The rule set used included 961 rules.

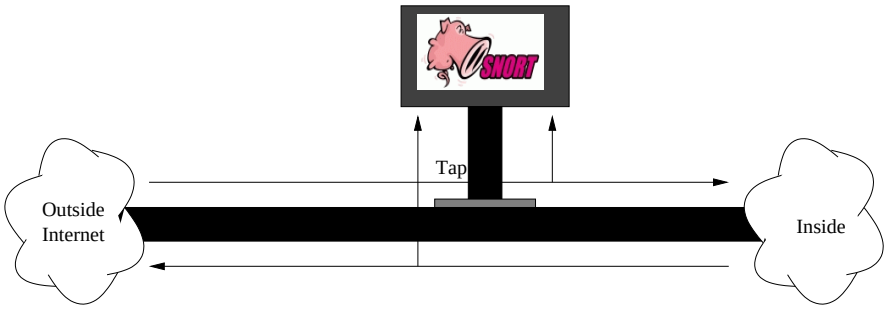

Figure 8.1: Single-Node Snort Setup

The second experiment was to run `Snort` on a single node monitor. The setup is shown in Figure 8.1. In practice, `Snort` is run on the scatterer host and it reads directly from the network card. We measured the decrease in effectiveness of the detection when the traffic rate increases[1]. The rule set used included only the 18 rules that actually fired on the test data obtained during the first test. Figure 8.2 shows the results of this experiment. The reduced performance is due to packet loss, which becomes substantial at approximately 150 Mbps. This experiment identifies the saturation point of this setup.

The third experiment was to run `Snort` in the simple setup of Figure 8.1 with a constant traffic rate of 100 Mbps and an increasing number of signatures. The experiment starts with only the eighteen signatures that are needed to achieve complete detection for the given data. The plot in Figure 8.3 shows how the performance decreases as more signatures

[1]The limit of 200 Mbps in the graphs is the maximum amount of traffic that `tcpreplay` is able to generate.

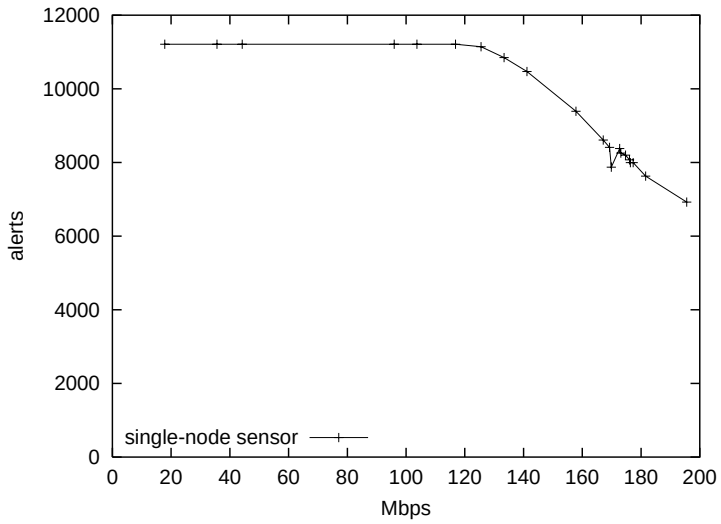

Figure 8.2: Single-Host Detection Rate for increasing Traffic Levels

are added to the sensor. This experiment demonstrates that such a setup is limited by the number of signatures that can be used to analyze the traffic stream.

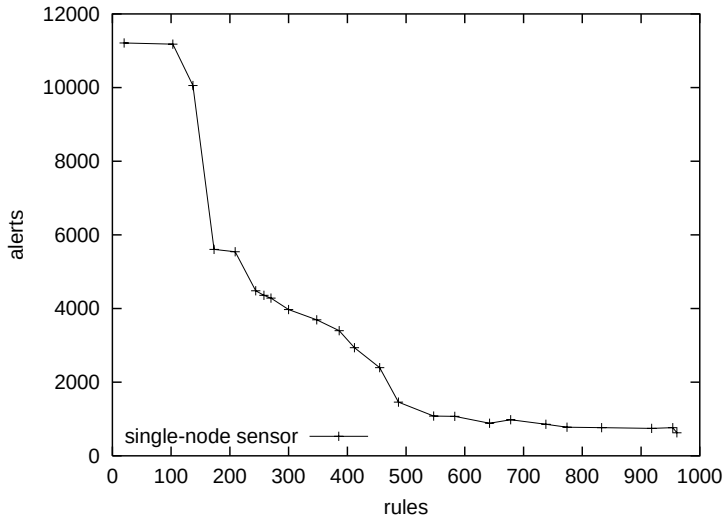

Figure 8.3: Single-Host Detection Rate for increasing Number of Signatures

The fourth and fifth experiments repeated the previous two experiments, but now by using `Snort` sensors in our proposed architecture. Figure 8.4 and 8.5 present the results of these experiments.

The performance of the single node experiments are included for comparison. The drop in detection rate at high speeds by the distributed sensor which can be seen in Figure 8.4 is caused by packet loss in the scatterer. The network cards currently used for the output traffic are not able to handle more than about 170 Mbps. The experimental results show

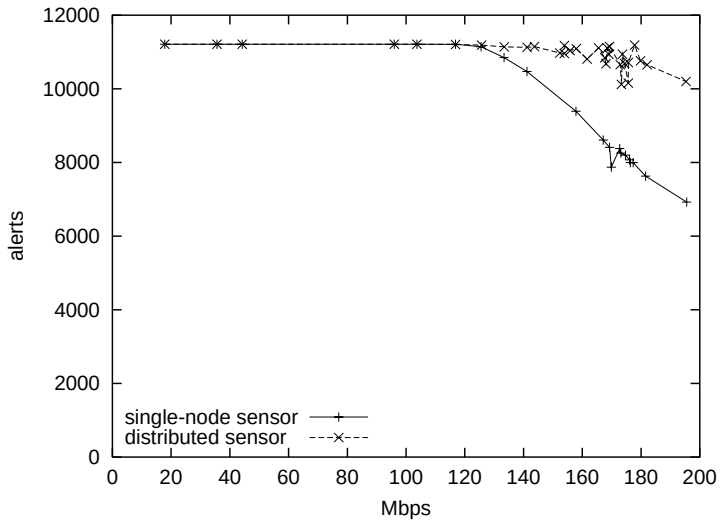

Figure 8.4: Distributed Detection Rate for increasing Traffic Levels

that the proposed architecture has increased throughput and is much less sensitive to the number of signatures used.

8.3 Adaptive Sensors

This section evaluates the prototype of our adaptive sensor that can process and analyze DNS requests. In contrast to HTTP that sends requests in plain text, the DNS protocol uses a simple compression mechanism to shorten the length of requests by substituting substrings of domain names with pointers to previous occurrences of these strings. This could have a negative effect on our assumption regarding the character distribution. Below, we show the results of our service specific anomaly detection sensor that has been installed on the DNS server of our department.

The service independent part of the packet processing unit (PPU) has been realized with Snort [81] because it already has the ability to reassemble IP and TCP traffic. It offers an interface that allows developers to use custom modules as plug-ins. We realized the service dependent part of our PPU as such a plug-in that is inserted directly after Snort's IP/TCP reassembly stage. This allows us to operate on completely reassembled UDP packets or acknowledged chunks of TCP streams.

Usually, DNS uses a single UDP packet to transmit a request to the server. In such a case, the complete payload of the packet can simply be passed to the statistical processing unit (SPU). In case of a request that is transmitted over a TCP stream, the PPU connects subsequent TCP stream chunks which it receives from the TCP reassembler. The simple header of the DNS request is parsed to determine the amount of data that is transmitted. When the complete request has been observed, it is passed to the SPU.

It is not entirely obvious how the type of a DNS request should be determined. In order

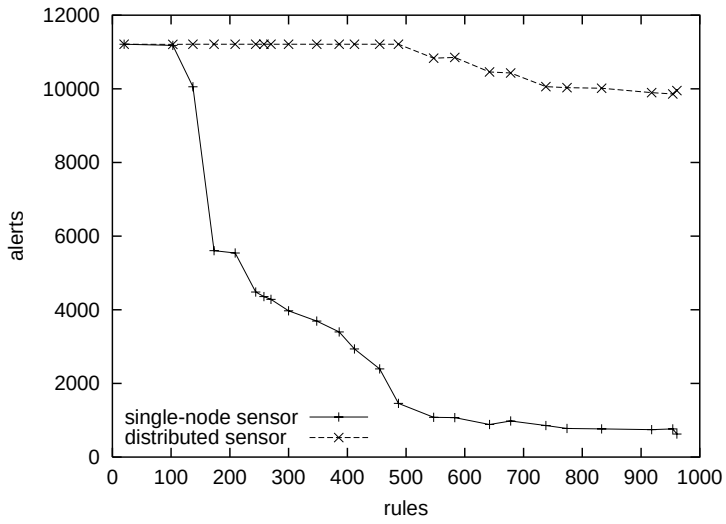

Figure 8.5: Distributed Detection Rate for increasing Number of Signatures

to save overhead, each request contains a number of records (called resource records) that usually have different types. Nevertheless, each DNS request contains a distinguished first resource record (called question) that specifies the desired operation. We use the type of that question resource record to calculate the type dependent anomaly score.

Several tables show the application model that was built during a training period of 24 hours. A total of 75463 requests with average length (μ) 39.572 and variance (σ) 31.915 have been processed.

Type	Explanation	Occurrences	relative Freq.
PTR	Reverse DNS Query	57306	0.7594
A	DNS Query	15963	0.2115
ANY	Request all Records	1167	0.0155
AAAA	IPv6 Query	599	0.0079
MX	Mail Exchange Query	317	0.0042
SOA	Zone of Authority	111	0.0015
Total		75463	1.0000

Table 8.5: Request Type Distribution

Table 8.5 above shows the absolute and relative occurrences of requests with respect to their type.

Table 8.6 below shows the expected character frequencies for all six segments as determined by the payload distribution. The first 32 values of the payload distribution are displayed in the left diagram of Figure 8.6.

After the initial training period, we used the resulting application model to test our ID sensor on the department's DNS server for 10 days. During this time, our module processed

Segment	0	1	2	3	4	5
Expected Freq.	0.117	0.257	0.185	0.199	0.117	0.1253

Table 8.6: Expected Request Character Frequencies

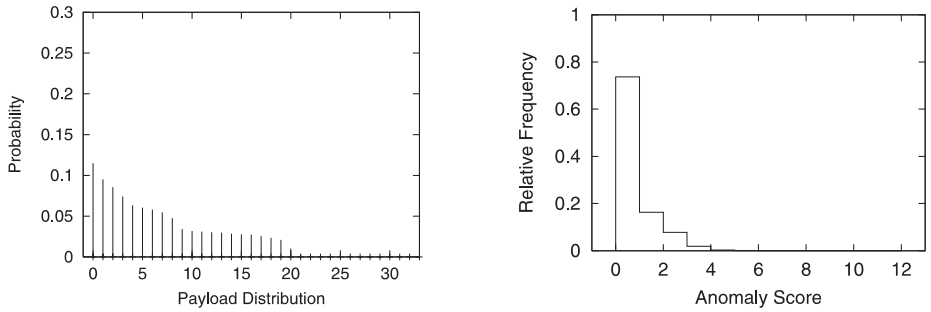

Figure 8.6: Payload and Anomaly Score Distribution

688388 `DNS` requests. Table 8.7 shows the absolute and relative number of requests with respect to their anomaly score. In addition, the relative numbers are visualized in the right diagram of Figure 8.6.

Score	Absolute	Relative	Score	Absolute	Relative
$[0,1[$	507041	0.73653	$[6,7[$	76	0.00011
$[1,2[$	112319	0.16316	$[7,8[$	14	0.00002
$[2,3[$	53294	0.07742	$[8,9[$	6	0.00001
$[3,4[$	13136	0.01902	$[9,10[$	0	0.00000
$[4,5[$	2240	0.00325	$[10,11[$	0	0.00000
$[5,6[$	262	0.00038	$[11,12[$	0	0.00000

Table 8.7: Absolute and Relative Distribution of Anomaly Scores

We have increased the default threshold of 6.97 to 7.0 and received only 20 false alarms during the complete test period (an average of only 2 per day). The false alarms have been exclusively caused by very short requests that contained less than 20 characters. As the χ^2-test is inaccurate for very small samples their A_{pd} score was the maximum possible. Additionally, they have been of types that only occur infrequently. Our sensor could be modified to put lesser weight on the χ^2-value for small samples to reflect this inaccuracy. In that case, the threshold of 7.0 might have resulted in no false alarms at all.

The payload of all requests that have been received during the test period were dumped into a file with a final size of 67.2 MB. We measured the additional load that our traffic analysis inflicted on the `DNS` server by running our module offline on that dump file. The average runtime (after executing the program ten times) on a `Pentium 4` (1.4 GHz - 512 MB RAM - `Linux 2.4.2`) to process the complete file was 41.3 seconds. As 10 days

worth of data could be analyzed in under a minute the additional load on the DNS server was negligible.

After the encouraging false alarm rate was explored, we attacked our DNS server with several actual exploits (obviously we use the latest, patched version of bind). Our tests included five DNS exploits listed in arachNIDS [6], a well known exploit database, and additionally the famous but now outdated NXT ('ADM rocks') exploit. The arachNIDS site lists a total of 8 offending signatures of potential attacks against port 53 (used by DNS). Three of them describe regular requests (e.g. requesting a zone transfer) that might be used by an attacker to get information. They are not included in our tests because they can be used in unmodified form by authorized clients to perform legal requests and therefore should not raise an alarm.

Table 8.8 lists properties and anomaly scores (AS) for all six test cases. All exploits use requests of types that have not occurred during our training phase and received the maximum A_{typ} scores.

Test Case	Length	A_{typ}	A_{len}	A_{pd}	AS
T1 - IQuery	27	15.0	1.07	14.48	10.61
T2 - Tsig LSD	509	15.0	10.86	8.99	11.36
T3 - Tsig OWN	546	15.0	13.11	16.36	14.98
T4 - Infoleak	24	15.0	1.08	13.74	10.32
T5 - Tsig Lucy	533	15.0	12.27	9.43	11.95
T6 - Nxt	557	15.0	13.87	20.00	16.66

Table 8.8: Anomaly Scores of Attacks

Table 8.9 shows for each exploit request the actual and expected character frequencies for all six segments that have been used to calculate A_{pd}. Figure 8.7 shows typical character distributions of two exploits. The left diagram shows a distribution with a very large slope while the distribution in the right one is too uniform.

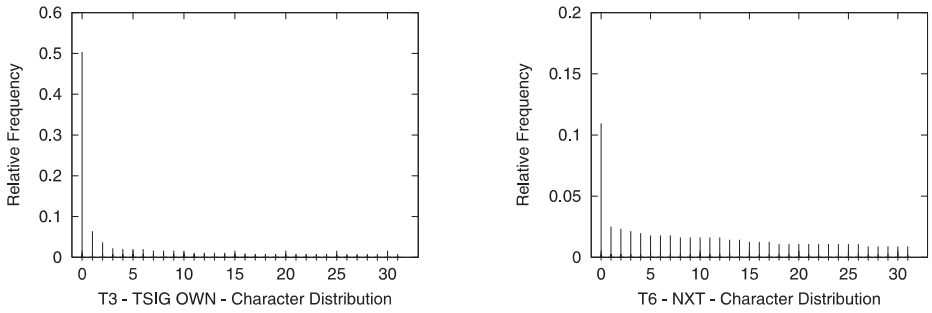

Figure 8.7: Exploit Character Distributions

All malicious requests have anomaly scores greater than the threshold and have been correctly identified as attacks. Notice that in addition to that, all intrusions have anomaly

Test Case	Seg. 0	Seg. 1	Seg. 2	Seg. 3	Seg. 4	Seg. 5
IQuery	13	7	3	4	0	0
	3	7	6	5	3	3
Tsig LSD	180	105	33	35	18	138
	59	130	93	102	60	65
Tsig OWN	275	67	33	40	23	108
	64	139	101	108	65	69
Infoleak	11	8	3	2	0	0
	3	6	4	5	3	3
Tsig Lucy	121	69	39	53	36	215
	62	135	100	106	63	67
Nxt	61	39	31	46	32	348
	64	143	102	112	66	70

Table 8.9: Expected and Actual Character Distribution of Attacks

scores that are larger than those of every regular request received during normal operation. This indicates that the threshold could be further raised without missing any attacks.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

> After all is said and done, a lot more will be said than done.
>
> – Unknown

This dissertation introduces *Network Alertness*, a novel concept for cooperating intrusion detection sensors that correlate event data to find and adaptively react to distributed attacks. Nodes exchange information of suspicious activity and can become aware of attack scenarios that affect multiple hosts in parallel; a problem that has been recently coined the 'Great Challenge' in intrusion detection by the DARPA [25]. The techniques necessary to implement our vision and contribute to the solution of this challenge have been presented and evaluated.

Network Alertness is based on a distributed framework that enables nodes to collaborate in a peer-to-peer fashion to identify emerging hostile patterns. These patterns can be described in an *Attack Specification Language* which utilizes a declarative semantics instead of an operational one. This helps domain experts to express intrusion scenarios in a more natural way. In order to prevent an explosion of the number of messages, the specification language had to be restricted. The consequential decentralized algorithm to find events that satisfy such patterns was implemented and exhibits superior scalability and fault tolerant properties when compared to current solutions. This is achieved by restricting the detection to only those hosts that witness actual parts of the attack. We abandon nodes with a dedicated task of correlating events as used in traditional centralized or hierarchical approaches because they limit scalability and are vulnerable to faults or attacks. The correlation framework, together with an interface to allow deployment and management of our collaborating sensors, was implemented and named *Quicksand*. We describe the system architecture of *Quicksand* and introduce a mechanism to integrate third-party products into our design.

An important problem in distributed systems is the synchronization of events, especially when temporal relationships between occurrences at different machines need to be

considered. As this is the case with the distributed patterns that can be specified in our framework, we have designed a universal `IP` based approach to efficiently exchange timestamp information in message based systems. As the network traffic between nodes can be very high, the primary design goal was the minimization of the introduced overhead. We achieved this by adding no additional information to each regular packet and perform explicit synchronization in an adaptive manner as rarely as possible.

As mentioned above, restrictions to our pattern specification language have been necessary to prevent an unreasonable amount of messages that nodes would have to exchange without them. Although most interesting scenarios can be easily described, it is not possible to express port scans (or similar reconnaissance activity) from a machine outside the protected installation against unrelated hosts that are located inside. To circumvent this drawback, a scalable but centralized detector is introduced that is deployed at the border between the inside and the outside networks. This system supports stateful, in-depth analysis of network traffic on high-speed links and covers those scenarios that our distributed analysis cannot handle. Special care was taken to design this sensor according to the same requirements used for the peer-to-peer approach; that is a high degree of scalability and fault tolerance. Therefore, a centralized tap located at the link partitions and forwards the data to a set of distributed probes where it is analyzed in detail.

In addition to the correlation framework which allows detection of attack scenarios, the intrusion detection system also has to react to the evolution of hostile patterns. The most important factor after an emerging threat has been identified is to prevent further damage. The first step is the identification of hostile packets that are sent by an attacker as the scenario progresses. This makes it necessary to selectively adapt the probes to traffic that originates from suspicious sources. We provide a network based anomaly sensor that extends the current simple models to analyze application data as well. A profile of a 'normal' client request to an Internet service is created and compared to every incoming query in order to determine its anomaly score. This capability of determining a rating for individual requests allows the modification (or adaptation) of the threshold depending on its source. When a request is considered to originate from a malicious party, the analysis can be much tighter.

9.2 Future Work

> One of the symptoms of an approaching nervous breakdown is the belief that one's work is terribly important.
>
> – Bertrand Russell

Although the presented components provide a coherent framework to correlate events and react to suspicious intrusion scenarios, there is always room for improvement.

Adaptive sensors are only a first step towards an effective and automatic response model. Currently, *Quicksand* notifies an administrator or may run a script when an intrusion is identified. No active counter-measures that interrupt connections or terminate

processes are integrated. It is also thinkable to launch a counter attack against a potential adversary. This could include simple reconnaissance to learn more about the intruder or even involve denial-of-service counterstrikes. It would also be beneficial to create a model that balances a threat which the system is exposed to against the expected result of the planned response. Especially for e-commerce sites, it is harmful when an IDS always terminates connections and blocks further access from people when only benign threats are detected, or worse, as a result of a false alarm.

Another point is the automation of the following two tasks that currently have to be done manually.

One problem involves the automatic installation of shared libraries in *Quicksand* probes. As explained in Section 5.1.3, unresolved symbols in shared libraries that contain code of new scenarios or responses result in a warning message. This requires the system administrator to manually provide the necessary additional libraries. Similar to an approach presented in [104], the missing libraries could be determined and loaded automatically.

The other manual task is the clustering of request types into classes (or groups) with similar properties (see Section 7.3) for the adaptive sensors. This could be automated for new protocols. Although minimal human intervention is required to define the request types, the determination of types with analogical structures could be done automatically by applying information-theoretical or statistical mechanisms.

Bibliography

[1] Advanced Encryption Standard. National Institute of Standards and Technology, US Department of Commerce, FIPS 197, 2001.

[2] Aglets. http://www.trl.ibm.co.jp/aglets/.

[3] Edward Amoroso. *Intrusion Detection - An Introduction to Internet Surveillance, Correlation, Trace Back, and Response.* Intrusion.Net Books, New Jersey, USA, 1999.

[4] D. Anderson, T. Frivold, A. Tamaru, and A. Valdes. *Next Generation Intrusion Detection Expert System (NIDES).* SRI International, 1994.

[5] J. P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P. Anderson Co., Box 42, Fort Washington, February 1980.

[6] arachNIDS: advanced reference archive of current heuristics for Network Intrusion Detection Systems. http://www.whitehats.com/ids, 2001.

[7] M. Asaka, A. Taguchi, and S. Goto. The Implementation of IDA: An Intrusion Detection Agent System. In *11th FIRST Conference*, June 1999.

[8] Rebecca Bace. *Intrusion Detection.* Macmillan Technical Publishing, Indianapolis, USA, 2000.

[9] J. S. Balasubramaniyan, J. O. Garcia-Fernandez, D. Isacoff, E. Spafford, and D. Zamboni. An Architecture for Intrusion Detection using Autonomous Agents. In *14th IEEE Computer Security Applications Conference*, December 1998.

[10] S. M. Bellovin. Packets found on an Internet. Technical Report 3, AT&T, 1993.

[11] K.P. Birman and T.A. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. Computer Systems*, 5(1):47–76, 1987.

[12] GNU bison: A general-purpose parser generator. http://www.gnu.org/software/bison/bison.html.

[13] M. Bykova, S. Ostermann, and B. Tjaden. Detecting Network Intrusions via a statistical Analysis of Network Packet Characteristics. In *33rd Southeastern Symposium on System Theory*, 2001.

[14] J.B.D. Caberera, B. Ravichandran, and R.K. Mehra. Statistical Traffic Modeling for Network Intrusion Detection. In *8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2000.

[15] National Computer Security Center. Department of Defense Trusted Computer System Evaluation Criteria. Orange Book 5200.28-std, DOD, December 1985.

[16] CERT Advisory CA-1999-14 Multiple vulnerabilities in BIND. http://www.cert.org/advisories/CA-1999-14.html, 1999.

[17] CERT Advisory CA-2001-02 Multiple vulnerabilities in BIND. http://www.cert.org/advisories/CA-2001-02.html, 2001.

[18] CERT Advisory CA-2001-13 Buffer Overflow in IIS Indexing Service. http://www.cert.org/advisories/CA-2001-12.html, 2001.

[19] CERT Advisory CA-2001-23 Continued Threat of the Code Red Worm. http://www.cert.org/advisories/CA-2001-23.html, 2001.

[20] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security.* Addison-Wesley, Reading, Massachusetts, USA, 1994.

[21] Common Intrusion Detection Framework. http://www.gidos.org.

[22] CISCO. CISCO Intrusion Detection System. Technical Information, Nov 2001.

[23] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems - Concepts and Design.* Addison-Wesley, Harlow, England, 2nd edition, 1996.

[24] M. Crosbie and E. Spafford. Defending a Computer System using Autonomous Agents. In *18th National Information Systems Security Conference*, October 1995.

[25] Defense Advanced Research Projects Agency. http://www.darpa.mil, 1998.

[26] Data Encryption Standard. National Bureau of Standards, US Department of Commerce, FIPS 46-3, 1977.

[27] J. D. de Queiroz, L. F. R. da Costa Carmo, and L. Pirmez. Micael: An Autonomous Mobile Agent System to Protect New Generation Networked Applications. In *2nd Annual Workshop on Recent Advances in Intrusion Detection*, September 1999.

[28] H. Debar, M. Dacier, and A. Wespi. Towards a Taxonomy of Intrusion Detection Systems. *Computer Networks*, 1999.

[29] D. Denning. An Intrusion-Detection Model. In *IEEE Symposium on Security and Privacy*, pages 118–131, Oakland, USA, 1986.

[30] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[31] S. T. Eckmann, G. Vigna, and R. A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. In *ACM Workshop on Intrusion Detection Systems*, Athens, Greece, November 2000.

[32] L. Eschenauer. Imsafe. http://imsafe.sourceforge.net, 2001.

[33] GNU flex: A fast lexical analyser generator. http://www.gnu.org/software/flex/flex.html.

[34] S. Forrest, S. A. Hofmeyr, A. Somayaj, and T. A. Longstaff. A sense of self for Unix processes. In *IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[35] K. Fox, Henning R., J. Reed, and R. Simonian. A Neural Network Approach Towards Intrusion Detection. In *National Computer Security Conference*, 1990.

[36] T. D. Garvey and T. F. Lunt. Model based Intrusion Detection. In *14th National Computer Security Conference*, October 1991.

[37] A. Ghosh and A. Schwartzbard. A Study in Using Neural Networks for Anomaly and Misuse Detection. In *USENIX Security Symposium*, 1999.

[38] NSS Group. Intrusion Detection and Vulnerability Assessment. Technical report, NSS, Oakwood House, Wennington, Cambridgeshire, UK, 2000.

[39] J. Hochberg, K. Jackson, C. Stallins, J. F. McClary, D. DuBois, and J. Ford. NADIR: An automated System for detecting Network Intrusion and Misuse. *Computer and Security*, 12(3):235–248, May 1993.

[40] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State Transition Analysis: A Rule-Based Intrusion Detection System. *IEEE Transactions on Software Engineering*, 21(3), March 1995.

[41] Intel. *IA-32 Intel Architecture Software Developers Manual Volume 1-3*, 2002. http://developer.intel.com/design/Pentium4/manuals/.

[42] IP Security Protocol. http://www.ietf.org/html.charters/ipsec-charter.html, 2002.

[43] ISS. BlackICE Sentry GBit. http://www.networkice.com/products/sentry_gigabit, November 2001.

[44] Peter Jackson. *Introduction to Expert Systems.* Addison-Wesley, Reading, Massachusetts, USA, 1986.

[45] M. Jazayeri and W. Lugmayer. Gypsy: A component-based mobile agent system. In *8th Euromicro Workshop on Parallel and Distributed Processing (PDP 2000)*, Rhodos, Greece, January 2000.

[46] C. Kahn, P. Porras, S. Staniford, and B. Tung. A Common Intrusion Detection Framework. *Journal of Computer Security*, July 1998.

[47] R. A. Kemmerer. A Model-based Real-time Network Intrusion Detection System. Technical report, Computer Science Dep., University of California Santa Barbara, November, 1997.

[48] J. Kohl, B. Neuman, and T. T'so. The Evolution of the Kerberos Authentication System. *Distributed Open Systems*, pages 78 – 94, 1994.

[49] C. Krügel and T. Toth. Applying Mobile Agent Technology to Intrusion Detection. In *ICSE Workshop on Software Engineering and Mobility*, May 2001.

[50] C. Krügel and T. Toth. Distributed Event Correlation for Intrusion Detection. In *Symposium on Network and Distributed Systems Security (NDSS)*. Internet Society, February 2002.

[51] C. Krügel, T. Toth, and C. Kerer. Decentralized Event Correlation for Intrusion Detection. In *International Conference on Information Security and Cryptology (ICISC)*. Lecture Notes in Computer Science, Springer, December 2001.

[52] C. Krügel, T. Toth, and E. Kirda. Sparta - A Security Policy Reinforcement Tool for Large Networks. In *IFIP Conference on Advances in Network and Distributed Systems Security*. Kluwer Academic Publishers, November 2001.

[53] C. Krügel, T. Toth, and E. Kirda. Service Specific Anomaly Detection for Network Intrusion Detection. In *Symposium on Applied Computing (SAC)*. ACM Scientific Press, March 2002.

[54] C. Krügel, F. Valeur, G. Vigna, and R. A. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *IEEE Symposium on Security and Privacy*. IEEE CS Press, May 2002.

[55] S. Kumar and E. Spafford. A Pattern-Matching Model for Instrusion Detection. In *National Computer Security Conference*, October 1994.

[56] MIT Lincoln Lab. The 1998 DARPA Intrusion Detection Evaluation. http://ideval.ll.mit.edu/1998_index.html, 1998.

[57] L. Lamport. Time, Clocks and the Ordering of Events in a distributed System. *Comms. ACM*, 21(7):558–65, 1978.

[58] B. Landreth. *Out of the Inner Circle, A Hacker's Guide to Compuer Security.* Microsoft Press, Bellevue, Washington, USA, 1985.

[59] W. Lee, S. Stolfo, and K. Mok. A Data Mining Framework for Building Intrusion Detection Models. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[60] U. Lindquist and E. Jonsson. How to Systematically Classify Intrusions. In *IEEE Symposium on Security and Privacy*, Oakland, USA, 1997.

[61] U. Lindquist and P. A. Porras. Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST). In *IEEE Symposium on Security and Privacy*, Oakland, USA, May 1997.

[62] T. F. Lunt. A Survey of Intrusion Detection Techniques. *Computer and Security*, 12(4):405–418, 1993.

[63] T. F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P. G. Neumann, and C. Jalali. IDES: A Progress Report. In *Sixth Annual Computer Security Applications Conference*, Tucson, Arizona, USA, December 1990.

[64] B. Mukherjee, T. Heberlein, and K. Levitt. Network Intrusion Detection. *IEEE Network*, 8(3):26–41, May/June 1994.

[65] P. G. Neumann and P. A. Porras. Experience with EMERALD to Date. In *1st USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 73–80, Santa Clara, California, USA, April 1999.

[66] Steven Northcutt. *Network Intrusion Detection - An Analyst's handbook.* New Riders, Indianapolis, USA, 1999.

[67] OpenSSH: Free SSH tool suite. http://www.openssh.org.

[68] OpenSSL: The Open Source toolkit for SSL/TLS. http://www.openssl.org.

[69] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *7th USENIX Security Symposium*, San Antonio, TX, USA, January 1998.

[70] V. Paxson and S. Floyd. Why We Don't Know How To Simulate The Internet. In *Winter Simulation Conference*, December 1997.

[71] J. Pearl. *Probabilistic Reasoning in Intelligent Systems.* Morgan-Kaufman, 1988.

[72] P. A. Porras and R. A. Kemmerer. Penetration State Transition Analysis A Rule-Based Intrusion Detection Approach. In *Eighth Annual Computer Security Applications Conference*, San Antonio, Texas, Dec 1992.

[73] P. A. Porras and P. G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *20th NIS Security Conference*, October 1997.

[74] P. A. Porras and A. Valdes. Live Traffic Analysis of TCP/IP Gateways. In *Internet Society's Networks and Distributed Systems Security Symposium*, March 1998.

[75] J. Pouzol and M. Ducassé. From Declarative Signatures to Misuse IDS. In *Recent Advances in Intrusion Detection (RAID)*, volume 2212 of *LNCS*, pages 1–21, Davis, CA, October 2001. Springer-Verlag.

[76] M. J. Ranum. Intrusion Detection and Network Forensics. In *M1 Tutorial - USENIX Security 2000*, Denver, Colorado, USA, August 2000.

[77] R. L. Rivest. The MD5 message-digest algorithm. Technical report, Internet Request for Comments (RFC) 1321, 1992.

[78] R. L. Rivest. The RC4 encryption algorithm. Technical report, RSA Data Security Inc., 1992.

[79] R. L. Rivest, A. Shamir, and L. A. Adleman. A Method for obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[80] M. Roesch. *Writing Snort Rules: How To write Snort rules and keep your sanity.* http://www.snort.org.

[81] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *USENIX Lisa 99*, 1999.

[82] Bruce Schneier. *Applied Cryptography.* John Wiley & Sons, Inc., New York, USA, 2nd edition, 1996.

[83] R. Sekar, V. Guang, S. Verma, and T. Shanbhag. A High-performance Network Intrusion Detection System. In *6th ACM Conference on Computer and Communications Security*, November 1999.

[84] J. Shavlik, M. Shavlik, and M. Fahland. Evaluating Software Sensors for Actively Profiling Windows 2000 Computer Users. In *Recent Advances in Intrusion Detection (RAID)*, 2001.

[85] K. R. Sheers. HP OpenView event correlation. *Hewlett-Packard Journal*, October 1996.

[86] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur. DIDS (Distributed Intrusion Detection System) - Motivation, Architecture and an early Prototype. In *14th National Security Conference*, pages 167–176, October 1991.

[87] D. Song. Fragrouter. http://www.monkey.org/~dugsong/, 2000.

[88] D. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and ssh timing attacks. In *10th USENIX Security Symposium*, 2001.

[89] Stanford research institut. http://www.csl.sri.com.

[90] William Stallings. *Network Security Essentials - Applications and Standards*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 2000.

[91] S. Staniford, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS - A Graph Based Intrusion Detection System For Large Networks. In *20th National Information Systems Security Conference*, volume 1, pages 361–370, October 1996.

[92] S. Staniford, J. A. Hoagland, and J. M. , McAlerney. Practical Automated Detection of Stealthy Portscans. In *IDS Workshop of the 7th Computer and Communications Security Conference*, Athens, 2000.

[93] J. Steiner, C. Neuman, and J. Schiller. Kerberos: An Authentication Service for open Network Systems. In *USENIX Winter Technical Conference*, 1988.

[94] A. Stubblefield, J. Ioannidis, and A. Rubin. Using the Fluhrer, Mantin, and Shamir attack to break WEP. In *Symposium on Network and Distributed Systems Security (NDSS)*. Internet Society, February 2002.

[95] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems - Principles and Paradigms*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 2002.

[96] Toplayer networks. http://www.toplayer.com, November 2001.

[97] Trusted Solaris 8. http://wwws.sun.com/software/solaris/trustedsolaris/, 2002.

[98] M. Undy. Tcpreplay. Software Package, May 1999.

[99] A. Valdes and K. Skinner. Adaptive, Model-based Monitoring for Cyber Attack Detection. In *Recent Advances in Intrusion Detection (RAID 2000)*, number 1907 in Lecture Notes in Computer Science, pages 80–92, Toulouse, France, October 2000. Springer-Verlag.

[100] V. Venema. tcpwrapper. http://wzv.win.tue.nl.

[101] G. Vigna, S. Eckmann, and R. A. Kemmerer. The STAT Tool Suite. In *DISCEX 2000*, Hilton Head, South Carolina, January 2000. IEEE Computer Society Press.

[102] G. Vigna and R. A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. In *14th Annual Computer Security Applications Conference*, December 1998.

[103] G. Vigna and R. A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999.

[104] G. Vigna, R. A. Kemmerer, and P. Blix. Designing a Highly Configurable Web of Sensors. In *Recent Advances in Intrusion Detection (RAID)*, Davis, CA, October 2001.

[105] G. B. White, E. A. Fisch, and U. W. Pooch. Cooperating Security Managers: A peer-based intrusion detection system. *IEEE Network*, pages 20–23, January/February 1996.

[106] Public-Key Infrastructure X.509. The Internet Engineering Task Force, 2002.

Curriculum Vitae

Christopher Jan Krügel

Born: 17.01.1976, Vienna, Austria

Education

2000 - 2002	Ph.D. studies of computer science at Technical University Vienna
Summer 2001	Research visit at University of California, Santa Barbara (Reliable Software Group)
Spring 2000	Graduation as Master of Science (Computer Science) at Technical University Vienna (with highest distinction)
Summer 1998	Summer session at University of California, Berkeley
1995 - 2000	Master studies of computer science at Technical University Vienna
1994-1995	Military service (compulsory)
Spring 1994	Graduation from high school (with highest distinction)
1986-1994	High school at Bundesrealgymnasium 22, Vienna, Austria
1982-1986	Elementary school at Volksschule 22, Vienna, Austria

Work Experience

since 03/2002	Assistant Professor at the Distributed Systems Group (TU Vienna)
2000 - 2002	Research Assistant at the Distributed Systems Group (TU Vienna)
1997, 1999	Internship at Siemens Austria (Program and System Development)

Awards

2000	EIB Scientific Award for an excellent project in the field of home and building automation
1998, 1999	Award for excellent performance as a student

Research Interests

- Intrusion Detection
- Network Security
- Distributed Systems
- Operating Systems

Publications

http://www.infosys.tuwien.ac.at/Staff/chris