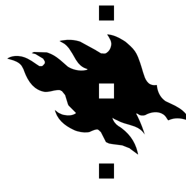
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS C
REPORT C-2004-61

A Survey of Application Level Intrusion Detection

Lea Viljanen



UNIVERSITY OF HELSINKI
FINLAND

A Survey of Application Level Intrusion Detection

Lea Viljanen

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
?

Technical report, Series of Publications C, Report C-2004-61
Helsinki, December 2004, 32 pages

Abstract

Abstract

This survey presents research for intrusion detection at the application level. The current approaches to intrusion detection are discussed and the aspects of collection, analysis and reaction models are introduced. The need for intrusion detection at application level is discussed. The majority of the survey introduces, categorizes and explains the past and present research efforts on the application level.

This survey is part of Trust Based on Evidence (TuBE) project.

Computing Reviews (1998) Categories and Subject Descriptors:
K.6.5 [Management of Computing and Information Systems]: Security and Protection - Unauthorized access
A.1 Introductory and Survey
D.2.0 [Software Engineering]: General - Protection mechanisms

General Terms:
Security, Design, Algorithms

Additional Key Words and Phrases:
Intrusion detection

Contents

1 Introduction **1**
- 1.1 IDS Data Collection Model . . . 1
- 1.2 IDS Data Analysis Model . . . 2
- 1.3 IDS Reaction Model . . . 3

2 Problems in the Current Systems **5**

3 Data Collection Methods for Application IDS **7**
- 3.1 Host based monitoring . . . 7
  - 3.1.1 Process auditing mechanisms . . . 7
  - 3.1.2 Kernel mechanisms . . . 8
  - 3.1.3 Library and system call interposition . . . 9
- 3.2 Network based monitoring . . . 9
- 3.3 Direct monitoring . . . 9
- 3.4 Monitoring the execution environment . . . 10
  - 3.4.1 OS-level environments . . . 10
  - 3.4.2 Dynamic execution environments . . . 11
  - 3.4.3 Component level wrappers . . . 12

4 Data Analysis Approaches to Application IDS **13**
- 4.1 Misuse detection . . . 13
- 4.2 Anomaly detection in execution monitoring . . . 14
  - 4.2.1 Sequence similarity techniques . . . 16
  - 4.2.2 Call policy techniques . . . 18
  - 4.2.3 State automata techniques . . . 18
  - 4.2.4 Machine learning . . . 20
  - 4.2.5 Data mining . . . 21
  - 4.2.6 Statistical profiling . . . 21
- 4.3 Anomaly detection in input semantics . . . 22
- 4.4 Specification based AppIDS . . . 22
  - 4.4.1 Policy based systems . . . 22
  - 4.4.2 Static analysis . . . 24

5 Application IDS Reaction models **26**

6 Conclusion **27**

Chapter 1

Introduction

Organizations today rely increasingly on the Internet and the partnering and data exchange opportunities the network connectivity brings. However, the current Internet and networking environment is full of threats, both external and internal. Physical security around the organization premises are not enough since network based automated attacks and network-borne viruses have caused organizations and their published applications to be potential attack targets every minute of every day. Restricting the network access to a set of partner organizations limits the exposure, but does not remove the general threat, since a simple misjudgement from the partner organization personnel may cause a security incident that has ramifications for all the connected partners as well.

Additionally, applications are nowadays so complex that practically every piece of software has some amount of faults, or bugs. Some of the bugs are security related, i.e. their existence creates a security vulnerability that can be exploited to create a real security problem or incident.

When these two facts are combined, it can be concluded that we need tools and mechanisms to detect the actions of potentially malicious people and automated applications. These actions may try to exploit known and unknown software faults or system features to gain access or priviledges against the published or implicit system security policy.

An intrusion detection system (IDS) tries to detect when such attempt is made. It usually consists of a data collection agent that collects information about the system being observed. The collection agents are sometimes also called sensors. This collected data is then either stored or processed directly by the detector proper. The detector analyzes the data and presents conclusions to the responsible person in some usable format. In some cases the detector can also initiate countermeasures for the detected intrusion.

Intrusion detection systems can be classified in many ways into several categories depending on their characteristics. One way of doing this division is according to the data collection method and the data analysis mechanism. Furthermore, we can add a new category to the classification, namely the reaction mechanism. After the introduction and problem descriptions, this survey follows this three category classification.

1.1 IDS Data Collection Model

The following aspects can be identified in the data collection models of various intrusion detection systems:

- Where do the agents or sensors reside? Classically they have been either host based agents or network based agents, HIDS and NIDS respectively.

- Is the collecting centralized or distributed, i.e. is the data collected from a single point, such as centralized `syslog` or in various places, such as different network segments.

- Is the monitoring direct or indirect? This means whether the IDS can directly observe the target, including its internal state, or is it a separate component relying on target external interfaces for information.

Network based IDS (NIDS) is a listening device in the network, where it typically captures all network traffic and tries to detect protocol packets with malicious contents. It has a set of usually lexical rules detailing the definitions of malicious traffic. Examples of malicious traffic are possibly port scanning, too large ICMP packets (i.e ping-of-death) and known web server CGI exploits. Network based IDS also has directives what to do when such activity is detected.

Host based IDS (HIDS) works within a computer server and tries to detect misuse or anomalies that occur in the server but are not necessarily network based, usually by examining various log files and system level audit trails. Examples of such problems are brute-force password guessing, changing key system configuration files and accessing protected files.

The collecting can be done centralized or distributed. Centralized collection reads the information from a single point, such as central log file or from main network switch or the internet firewall. However, this approach has both scaling and coverage problems, therefore most current systems preferably deploy distributed collection, i.e. there is an agent in nearly every host and network segment.

The monitoring can be done indirectly or directly. Indirect monitoring is defined as "The observation of the monitored component through a separate mechanism or tool" while direct monitoring can access the component and its internal state directly [Zam01]. Examples of indirect monitoring are application log file reading, while direct monitoring is performed within an application.

These issues in the context of application level intrusion detection are discussed in Chapter 3.

1.2 IDS Data Analysis Model

Whether one deploys a HIDS or NIDS system, centralized or distributed collection, an analysis system is required to provide a sensible summary on the observed activities. One feature of the analysis model is the location of the analysis, whether the analysis is done in many places simultaneously) or centralized.

But the most differentiating feature of this is the analysis philosophy; misuse detection, anomaly detection or specification based analysis. This has been one of the basic factors according to which IDSes have been usually classified and by which we also classify application level efforts in Chapter 4.

Misuse detection relies on patterns (usually called signatures) of known attacks and effective handling of large amounts of network traffic or log files from which to detect these patterns. While the signatures are attack or attack-class specific, and thus their applicability to unforeseen situations is limited, some new attack types can be detected if they fall in the similar class than some previous attacks and the signature has been written generically enough.

Anomaly detection tries to differentiate between anomalous and normal behaviour of the system and alert on all unexpected behaviour since it may be an attack. The problem of this method lies on the question how to effectively define system normal behaviour, how to compare the current situation to this normalized ideal and how to minimize the false positive rate (i.e. normal behaviour flagged as intrusions) while keeping the false negative rate (i.e. missed intrusions) close to zero. On the positive side, this method is not tied to any specific attack or attack type and can

potentially alert to new and previously unseen attacks. This technique is especially valuable in new situations where attacks and attack types are not known.

Specification based analysis approaches the problem by more or less formally by defining the correct behaviour of the program [KRL97] or network protocol [SGF+02] and reacting to any deviations. The key problem in this approach is the initial generation of such a formal model, which needs to be done for each security critical system.

Each of these philosophies have different techniques suited to the task such as statistical analysis, syntactical pattern matching or expert systems. For example, the classic IDES system [Den87] uses rule based pattern matching in analyzing the audit logs. The basic detection principles have been very well classified by Axelsson in his survey of intrusion detection systems [Axe00].

1.3 IDS Reaction Model

One newer aspect in the IDS classification is the IDS reaction model, i.e. what the system does when it finds out a potential problem. The reaction model can be immediate or delayed. Depending on the organization and its capabilities sometimes it is quite enough to know a penetration has occurred some time in the past, while in some cases it is quite imperative to be able to instantly do something to the suspicious activity.

Traditionally IDS components have been interested in just generating and sending alarms immediately so that people can deal with the intrusion. This is a sensible approach in situations where people are available 24x7x365. But in some organizations it can take hours or even days for a system administrator to be able to react to the alarm. Therefore, some organizations would actually be better off with a delayed reaction IDS.

The delayed reaction has given rise to various expert systems that analyse massive amounts of network traffic and system logs with for example data-mining [LS98] or expert system techniques [LSC97]. These enables them to create better overviews and search for larger patterns in the attacks. This can be done for example during the night and when people show up to work in the morning, they are given a report on the night's activities. Various types of delayed analysis systems have also been utilized in commercial products.

Another aspect is the mechanism of the reaction, active vs. passive. Passive reactions, such as alarms, do not affect the observed system in any way. Active reactions, such as closing the attacking network connection, actively protect the system, thus gaining the name Intrusion Protection Systems or Intrusion Prevention Systems (IPS).

The different variants are illustrated in a simple matrix form in Figure 1.1. The lower left hand quadrant, i.e. active but delayed measures, is currently somewhat open. In this quadrant would fit measures that for example adjust suspect peer authorization, reputation or trust values after a security incident has occurred. These measures could also be taken as part of the immediate reaction, but the type of these measures is such that their effect is visible usually after the incident has occurred.

The surveyed application IDS systems with some reactive component are discussed in Chapter 5.

Reaction time \ Reaction type	Active	Passive
Immediate	***IPS***	***Current mgmt consoles***
Delayed	**--**	***Log analysis systems***

Figure 1.1: The IDS reaction matrix

Chapter 2

Problems in the Current Systems

Consider a scenario where a local workstation user is sending all outgoing e-mail to a company main mail server, which then resends these e-mails to the real destination. This is a common setup in organizations where the outgoing connections of a simple workstation are restricted.

What happens if the workstation is infected with a virus, worm or trojan that is either propagating by sending out a mass mailing of itself or sending out some other undesired content (i.e spam) to all addressess in the addressbook. These e-mail borne viruses have been very common in recent years and the trojan horse programs installed by the viruses have been used to send spam.

Note that the infection can occur even if an antivirus program is deployed at the workstation. The time between the detection of a new virus and the saturation point of a network enabled virus is diminishing so fast that the antivirus companies have a hard time developing and especially globally distributing the recognition signatures and countermeasures in time. The same is true for network IDS systems that use pattern matching so that the signature patterns need to be distributed.

So, we have an infected machine on the network that starts spewing out massive amounts of e-mail (either with viral or spam payload). This is not detected by the NIDS, since sending e-mail is by its rules quite acceptable, nor is it detected by the HIDS at the mail server, since it also considers receiving e-mail from a trusted host in the network quite normal. So the mail server happily accepts and starts redistributing the harmful content around the world.

Another scenario could be an extranet server that is offering some application via a defined Web Services interface. It probably has some form of access control and authorization to determine which systems are allowed to connect and what various contacting clients can do. But what happens if the client system is penetrated by a malicious hacker? The client penetration may not be at all visible to the extranet server, since the authentication and the authorization information is usually stored at the server and possibly even unencrypted on the system disk. Thus the attacker can send syntactically correct but semantically malicious information to the extranet server undetected by access control and authorization mechanisms, NIDS or HIDS.

Some additional detection approaches are required if we aim to limit the damage of these type of situations to the minimum. Especially some level of application awareness is required of the intrusion detection systems.

Application level intrusion detection (AppIDS) detects intrusions by collecting information about the monitored application system, analyzing this information by evaluating relations, and taking some action if a relation result is judged to be anomalous [Sie99].

The difference between an application IDS and a host based IDS is the fact that AppIDS is application specific; it examines one application or application class only. HIDS usually detects problems within one host system, especially on operating system level but it can include application specific components as well. Similarly, the difference between AppIDS and NIDS is the fact

that NIDS uses only network based collection methods. While a NIDS can also include application specific rules, it is concentrating on the network traffic in general, not one specific application.

Defined like this, application IDS is not a new idea. Research on various forms of application execution monitoring has been done in the 1990's, for example in [FHSL96]. The name *application intrusion detection* was coined later by Sielken in [Sie99] and its benefits were enumerated more closely by Diego Zamboni [Zam01]. The application level intrusion detection has several good characteristics over NIDS and HIDS that make it a desirable enhancement to the IDS palette:

- Depth of defence. This is a basic tenet in the security field, if one system (such as a firewall) develops a problem, we must have other mechanisms in place to detect and protect against intrusions.
- The application awareness in NIDS and HIDS can be restricted. This means that the network and host based systems do not necessarily need to be enhanced to recognize application level problems, this can be delegated to an application level system.
- It can technically monitor applications directly, i.e. there can be sensors observing the application internal state directly. However, all AppIDS methods do not utilize this possibility.
- The application can see data which a network or even host IDS sensor can not. The key example is traffic that is encrypted at the application level (examples are SSH, SSL). This NIDS blindness to encrypted traffic is a concern in some organizations. Terminating the encrypted connection in the perimeter may not be an option for all environments.
- The IDS reaction can be more fine-tuned according to the service capabilities. Depending on the intrusion type we might not want to break the connection altogether, but for example restrict the connection or data rate (i.e. tarpitting). Other application type countermeasures can be changing the source authorization parameters (i.e what actions are permitted and what data is available).

There are also problems at the semantic level. Most intrusions today exploit known application vulnerabilities, which are in many cases weaknesses in mechanical input handling. Unchecked buffer bounds, misplaced nulls and format string problems etc are syntactical in their nature, i.e. the received value differs from the expected value in length, type or allowed characters. But current systems are not interested in the actual semantic sensibility of the received input, i.e. if the client suddendly places an order for 10 000 000 flight tickets. While inputing this should not cause immediate access to the operating system level (unless there is a buffer overflow and attached exploit code) or denial of service attack, it may still be a cause of concern because of fraud possibilities. Application level mechanisms can potentially analyze this type of problems.

Chapter 3

Data Collection Methods for Application IDS

Before we can actually analyze whether an intrusion has actually happened in an application, we must have access to the application behaviour at a suitable level. This chapter discusses various options for monitoring the application and collecting the application behaviour data for analysis.

However, not all reviewed work in the application IDS research domain were explicit in their data collection methods (for example [JL01]) and some were just developing new analysis methods to old collection mechanisms (for example [WD01, GJM02, WFP99]). The works detailing new analysis methods are discussed later in Chapter 4.

The reviewed research papers were generally not discussing the distributed vs. centralized aspect of their collection method. However, since application level data collection can add information to the overall intrusion detection system, these methods can be seen as extra sensors providing depth of defence in a larger integrated intrusion detection system.

3.1 Host based monitoring

There are several mechanisms within a host, typically at the operating system level, that can be used to monitor an application. Some of these could be seen as host-based intrusion detection data collection but since they have insight to the application behaviour, we have included them in this survey.

3.1.1 Process auditing mechanisms

The process auditing subsystem of the operating system is of use to application level intrusion detection especially when it comes to extracting system call or other process behaviour data for execution monitoring analysis. Process auditing subsystems do not have insight into the application internal behaviour, but are able to monitor how the application acts towards the operating system.

Early work used auditing subsystems to extract security events from the operating system logs. For example VAX/VMS has a very advanced auditing facility that has been used for user activity tracing in an intrusion detection setting [TCL90]. In this particular case the audit events analyzed were user commands, so this is not really in the application intrusion detection domain.

A more widely used system is Solaris SunSHIELD Basic Security Module (BSM), which is a part of the Sun Microsystems' Solaris operating system. It provides the security features defined

as C2 in the Trusted Computer System Evaluation Criteria (TCSEC). The features provided by the BSM are the security auditing subsystem and a device allocation mechanism that provides the required object reuse characteristics for removable or assignable devices [Sun94]. BSM has been used in several AppIDS systems to be the source of system call information [KFL94, KRL97, End98, GSS99, GMS00, MG02].

Wespi et al also used an unnamed audit subsystem to collect audit events [WDD00]. The received audit records were not system calls, but on a more abstract level where the basic operation information was available (such as "file open"), but not the particular parameters.

Other operating system tools have also been used. Some Unix systems contain the user-level program `strace`, which will run a program given as an argument and list all the system calls it makes. Other such Unix programs also exist, such as `truss`. The `strace` program has been used to gather information in several application IDS systems [FHSL96, KH97, Mar00]. Additionally, the data analyzed by Lee et al ([LS98],[LSC97]) had been obtained with `strace`.

Since these user level programs exist, the Unix kernel offers primitives for system call trace data collection. Some research also used these underlying operating system constructs directly from user level for better performance [SBDB01], [FKFG03].

3.1.2 Kernel mechanisms

The process auditing mechanisms above used operating system services for reporting system call or audit data but worked at operating system user space. However, this reporting function can also reside at kernel level. For example, Sekar et al used a kernel level module to intercept system calls to support policy enforcement. Also the analysis mechanism resided in the kernel space [SBS99].

A kernel-level approach has also been used in behaviour data collection from software execution environments. In the work by Ko et al, the execution environment i.e. a software wrapper, functions in kernel space [KFBK00].

One application level mechanism utilizing kernel level possibilities is the waypoint technique. Waypoints are kernel-supported trustworthy control flow checkpoints in the code. Waypoints can actively report program control flow information in real time. Also security attributes can be attached to each waypoint or to a sequence of waypoints. Waypoints are set at each application function entrance and exit, also middle waypoints that reside in the function can be created. The waypoints are generated via static analysis [XDC04].

From the data collection point of view, waypoints have one key benefit over plain system call collection. Waypoints can give the intrusion detection system a better granularity than traditional system call based solution by the ability to create more waypoints than system calls [XDC04].

The key problem with this type of approach is the extensive kernel support required. The waypoint structures and policy enforcement systems must be located in the kernel to be secure from manipulation. While they are more secure from the system users, it is also riskier to augment stock kernels with extra modules. From the kernel point of view, any security or performance problem in the added IDS functionality may be reflected to the whole system. This analysis applies to monolithic kernel structure, exhibited in systems like Linux. The situation is different in microkernel systems, such as Windows NT and derivatives, where only minimal message passing and scheduling systems are present in the kernel proper. Other functionality is running in user space or between the outer user ring and inner kernel ring. The effects of embedding IDS functionality into a microkernel depend on the positioning of the intrusion detection modules in the microkernel protection system. This has not been a very well researched area.

Kernel-level approaches have also other problems. First of all, they are kernel specific and thus not portable. Secondly, the kernel systems can also cause performance interference, i.e. generic

overhead to operations that are not in need of an IDS mechanism. The system management is a problem as well, since reconfiguring a kernel-space system may require a system reboot. Also the kernel services have a very low level of abstraction, which causes the analysis phase having also low level of abstraction and deduction.

3.1.3 Library and system call interposition

One possibility of collecting information about application calls is library or system call interposition, which is the process of inserting a modified call between the application and the real target of the call.

The interposition can be done at library level, for example in Linux and Solaris environments the LD_PRELOAD environment variable is able to insert libraries that are searched before original libraries. This technique can be used to track the library calls an application makes.

The system built by Kuperman and Spafford used this library interposition to detect unsafe Unix programming practices, such as overruns of static buffers in `sprintf` calls. The penalty of interposition was about 3% for non-interposed calls and about 57% for interposed and logged system calls. Most of that overhead was due to the logging mechanism [KS99].

Call interposition can also be used at kernel-level. Bowen et al used a loadable Linux kernel module to insert an interposition layer into the kernel and rewrote the kernel system call table to execute first an interceptor and then the call. If any function besides interception is required, it can be specified in a module compiled from a behavioral monitoring specification language (BMSL) description [BCS+00].

The kernel-level method described above is able to track the application behaviour towards the kernel only, whereas library interposition is able to collect data on the library call level. Static analysis can achieve similar goals to library interposition, but it can not give us information on what happens in an application at run-time.

3.2 Network based monitoring

Some application IDS systems do use network based data collection, since they use the data available from the network for intrusion analysis. This assumes that the application is distributed in a way that network data is meaningful for analysis. This approach is not discussed in detail, since this is more in the domain of network based IDS.

Some interesting research exists, though. For example Raz et al captured data in a dynamic data feed for semantic anomaly analysis [RKS02]. Also CORBA remote calls have been intercepted so that the remote call execution patterns could be analyzed similarly to system call patterns [SMS99]. These approaches show that network traffic interception can be utilized in the application layer as well.

3.3 Direct monitoring

The most straightforward way of collecting data for misuse or anomaly analysis in an application is to write detection and collection hooks into the application code itself. This is direct monitoring, i.e. the method can access the component and its internal state directly.

Diego Zamboni did code internal misuse detection in his research by modifying existing software to detect well known attacks against it. The solution described added a series of detection

code agents to the `sendmail` application code against attacks collected from Common Vulnerabilities and Exposures (CVE) database [Zam01]. He also added some generic agents, for example against port scanning. Different techniques, such as character matching heuristics or stateful/stateless detection, were used in the agent implementation.

Similar approach was also used by Sielken, who first did a risk analysis of potential application fraud points and then wrote additional code to detect anomalies in client behaviour [Sie99].

Elbaum and Munson instrumented an application, a Linux kernel in their case, with code hooks that provide information on how the application modules call each other [EM99]. This module profile information was then used for anomaly detection.

The hard part of the code-internal method in general is the extensive programming and programming related activities it requires. While the actual number of code lines generated is not significant, modifying an existing system means in practice that a significant effort is spent in understanding the code before being able to make meaningful modifications [Zam01].

The actual agent or sensor can also be in a separate module, which has been linked into an application via a suitable interface [AL01]. This method enhances the flexibility of the agent implementation and diminishes the time needed to understand the code base. However, the used interface limits the agent's view of the application and also places restrictions on how the agent can affect the application. Therefore, the degree of directness in the agent model is also diminished depeding on the interface characteristics.

3.4 Monitoring the execution environment

A very distinctive class of IDS data collection methods includes systems where the application execution happens in whole or in part in a pre-configured container or other safe execution environment. The executed application is monitored by this container and the container is able to collect behaviour data and potentially also analyze unexpected behaviour. There are several variants of this basic idea.

3.4.1 OS-level environments

There are various methods of restricting the application visible environment for security reasons. One of the most successfull is Unix kernel `chroot` system call and the associated control program, where a subdirectory of the file system hierarchy is created to resemble the full Unix file system, and the application execution is restricted to this subdirectory and the services and components visible from there. However, `chroot` can not be considered a real application IDS, since it just prevents applications accessing the real OS file system and it has not been designed to alert on any undesired activity. There are other such execution restriction systems as well, such as the BSD `jail` but since they are somewhat off-focus we do not explore them in more detail.

Since no widely used operating system offers a ready made system focused on application level intrusion detection, there are ways to augment the systems with monitored execution environments or application security wrappers. One such effort is the Generic Software Wrapper system, which is a loadable Unix kernel level module (LKM) that associates wrapper instances with application processes [FBF99]. Generic software wrappers have also been used by Ko et al in their work [KFBK00].

Wrappers are defined in the Wrapper Definition Language (WDL) that masks the operating system details from the wrappers and also defines the functionality the wrappers have. One key point of this system is the fact that the WDL characterization of the low level kernel API contains some semantic information on the system calls, for example a tag labeling a call parameter or

return value as a file descriptor. This enables both some level of OS independence and also simple semantic reasoning in the wrapper action rules.

Wrappers intercept some or all system calls the applications make and take action based on the WDL definition. The wrapper can either deny the system call or augment or transform the call. Augmenting means adding functionality to the call, such as encrypting or performing intrusion detection analysis on the call data. Transforming, on the other hand, is replacing the call with one or more other system calls or other events [FBF99].

Another example of this kind is *Systrace*, an environment which supports fine grained process confinement, intrusion detection, auditing and privilege elevation [Pro03]. There the system calls are recorded by a kernel-level mechanism and the system call is checked against the recorded normal policy in a user space policy daemon. The approach takes also the system call arguments into account. *Systrace* is currently available for Linux, Mac OS X, NetBSD, and OpenBSD.

One positive point of this OS level approach is the fact that it works is practically application independent. It is also easy to deploy even for older or third-party applications. However, since it is an OS level tool, it does not necessarily have any deep insight into the actual workings of the application. Therefore, these systems are doing indirect monitoring. A suitable system may not be available to the preferred operating system either.

3.4.2 Dynamic execution environments

Dynamic execution environments (DEE) are systems where compilation, profiling and optimization are dynamic, such as Sun Java and Microsoft .NET Common Language Infrastructure [IF02]. Dynamic execution environments are a subgroup of virtual execution environments (VEE), more often called virtual machines.

As an example of a dynamic execution environment, programs written in the Java programming language are run in Java Virtual Machine (Java VM, JVM). Rather than running directly on the native operating system, the program is interpreted by the JVM and the resulting bytecode is run within the JVM on top of the native operating system. The JVM restricts the application by creating a sandbox, where a security policy dictates what the application is allowed to do.

One IDS technique utilizing the Java virtual machine is dynamic sandboxing. It consists of two activities: sandbox generation and sandbox execution. In the first, a dynamic sandbox profile is constructed by running the program with an instrumented JVM and all observed method calls are written to the profile. Later, in the execution phase proper, observed method calls are compared to this profile and anomalies cause an error [IF02].

The difference between JVM's own sandboxing and this dynamic sandboxing is similar to the difference between Unix `chroot` jail and a real IDS. Both the Java sandbox and the Unix `chroot` are able to just deny the activity outside their limits, but dynamic sandboxing can be classified as a real intrusion detection system, since it is able to detect malicious activity.

There are also systems where only the intrusion detection data collection agent is built into a JVM [SKV03]. To monitor any suspicious activity performed by applications running in the JVM (JikesRVM in this case) the virtual machine was extended with an event logging system. The auditing facility consists of an event driver, an event queue, and an event logger. The event driver adds thread-level execution events to the event queue. The logger processes events that are contained in the queue and writes them to an external log . This external log can then be processed by a suitable intrusion data analysis system (STAT in this case). In the analysis model this system is more related to host based intrusion detection systems than other application level systems.

3.4.3 Component level wrappers

Component based application development has rapidly become popular in recent years. There are sound reasons for it, especially regarding cutting development costs with code reuse and isolating faults with modularization.

Application wrapping for intrusion detection can also be done on the application component level instead of the level of application as a whole. Herrmann and Krumm created an architecture where an adapter is generated for each Java Bean and the bean is wrapped by the adapter. An adapter generator introspects the bean for its interface actions and creates the related adapter based on this examination. This approach means that each component is protected by the adapter and all component interaction via the component interface goes through the adapter so that the interaction can be observed [HK01]. This is direct monitoring in the sense that the application interaction between the beans can be observed, but on the other hand this method does not have visibility in the inner behaviour of the beans.

Chapter 4

Data Analysis Approaches to Application IDS

Since application level IDS is a relatively new concept and there are several data collection methods, there are several algorithms and methods for analyzing the collected data. Generally analysis approaches are divided as follows:

1. Misuse detection
2. Anomaly detection
3. Specification based intrusion detection

In this document, survey of the anomaly detection techniques has been divided into two subchapters: anomaly detection in execution monitoring and anomaly detection in input semantics. This division is purely a device for managing the readability and structure of this document, conceptually these two are in the same anomaly detection domain.

4.1 Misuse detection

Misuse detection has not been a very widely researched topic in the application IDS arena. The methodology map is shown in Figure 4.1.

A trivial approach to misuse detection is to make modifications in the code to detect known vulnerabilities [Zam01]. More refined method is to use known attacks to train a neural network, a multi-level perceptron (MLP), to recognize them and also similar attacks from a stream of audit events [End98]. A software wrapper technique can also be used for misuse detection if the attack signatures are encoded into the wrapper data structures [KFBK00].

Kuperman and Spafford used library interposition to detect unsafe Unix programming practices, such as unbounded data insertion into static buffers, race conditions and unsafe system calls (such as `system()` and `execvp()`) [KS99]. Since this technique searches for vulnerabilities facilitated by the used platform and tools instead of the application characteristics, it is application independent.

This lack of research in the misuse domain is not very surprising, since misuse detection uses known vulnerabilities either for searching or training. This means each new discovered vulnerability must be encoded in the application intrusion system. If the detection method is source code based, as in the trivial case, this would mean recompiling and reinstalling the application every

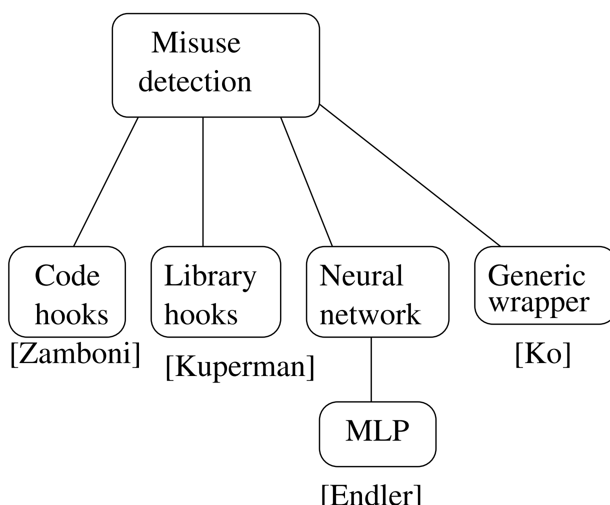


Figure 4.1: The misuse analysis methodology map

time a new type of vulnerability is found. Also wrapper based techniques suffer from this update problem, although to a lesser extent, since the update must be done on the wrapper side, the application can remain as it is.

Machine learning could make life with new vulnerabilities much easier, if we could be sure that a new attack would be covered by the learning generated by the old training data. At minimum, an IDS system employing machine learning would need to be tested every time a new vulnerability is observed, and if the old training does not cover the new problem, retraining must be done.

4.2 Anomaly detection in execution monitoring

One key form of application execution monitoring is application call analysis. By the word *call* we mean both system calls and application internal procedure or method calls. Also "calls" to software modules are considered. While the techniques of extracting information for different call types are different (see Chapter 3), algorithms for analysing this type of information are similar. Thus we do not separate these types here but discuss them jointly. The analysis method map for anomaly detection is shown in Figure 4.2. The picture also includes research done in the area of anomaly detection in input semantics, which is discussed in more detail in Chapter 4.3.

System calls seem to be at a suitable granularity level for detecting malicious behaviour in applications. This suitability is based on the assumption that the exploit must interact with the operating system to create a real intrusion. While this is true in the case of buffer overflows and shell code exploits, it leaves problems caused by the application data semantics unexplored.

The seminal paper describing this idea came from Forrest et al in 1996 [FHSL96],[FHS97], and it has generated a multitude of different techniques and variants for either obtaining or analyzing the system call information.

Their basic idea is to parallel computer defense with human immunology. This analogy needs the definition of system "self", which by their original idea is obtained from the program system call profile. This idea of self can be used for anomaly detection by collecting traces of system calls from programs in a secure training phase and creating a normal profile of how the process runs. This normal database is then used to monitor the running processes and flag anomalies when system calls occur in abnormal positions.

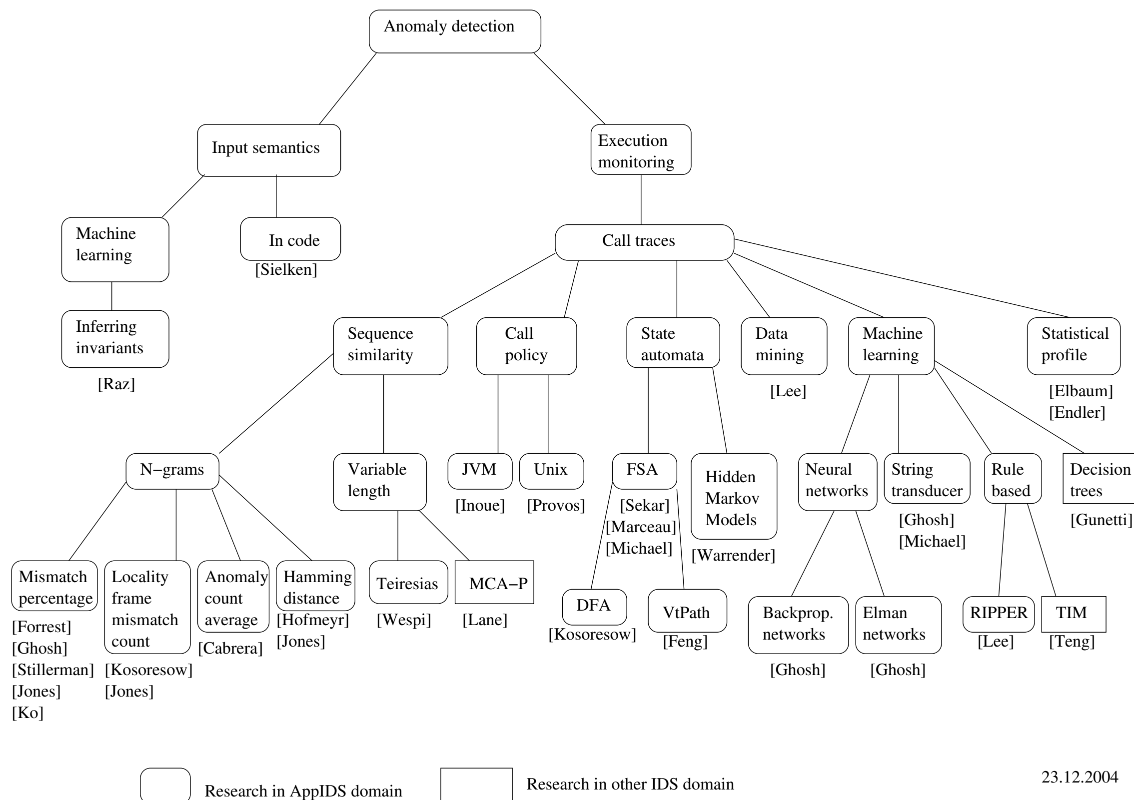


Figure 4.2: The anomaly analysis methodology map

This overall method is called *stide*, sequence time-delay embedding, and it contains four parts: selecting the feature set (system calls in this case), collecting normal data, extracting features from the data and detecting anomalies by comparing the features of current situation to the normal data.

There are two general problems in the system-call based anomaly detection; mimicry attacks and impossible paths [WD01].

Mimicry attack is a form of attack where the exploit is coded in a way that it resembles the normal run of application as much as possible, for example using only normally occurring system calls. However, not all attacks can be coded this way. This attack method would require extensive knowledge on what is normal for this application and also what kind of application model or detection threshold target is using [KH97], [WD01]. There are also two types of mimicry attacks: local and global. A global mimicry attack combines the legal system calls of multiple functions, while a local mimicry attack uses the legal system calls of only the running function [XDC04].

The impossible path problem manifests in techniques where the model includes all possible control paths, i.e. it has also those that no valid run-time behaviour uses. An example of this type is the return-into-others impossible path, where a function is called from location A, but the control is returned to point B [XDC04]. The main concern with impossible paths is that the model can classify such an occurring impossible path as normal and thus miss an attack. Not all system call methods exhibit this problem, typical examples of this problem are in the state transition research [WD01],[SBDB01].

4.2.1 Sequence similarity techniques

The original research by Forrest et al used simple sequence enumeration in data collection and feature extraction from the system calls. The system call sequences were of fixed size, *N*-grams, and they used a technique called look-ahead pairs for determining mismatches. The actual measure of anomaly in the original research was a *mismatch count* as a percentage of total number of possible mismatches in the look-ahead sequence. When tested with real exploits against `sendmail` application, the mismatch counts were between 0.3-5.3%. The described algorithm can be implemented in *O(N)* time, where *N* is the system call sequence length [FHSL96].

The key finding was that this was a good method of differentiating between processes and for detecting problems where an unusual system call sequence in an attack is used. However, this can not be used to detect all exploits, for example race conditions or resource stealing.

An observation made by Kosoresow et al around the same time was that all the anomalies observed exhibited behaviour where the system call mismatches between normal and intrusion traces are executed in bursts, i.e. they were temporally close. This allows for an anomaly detection technique where just the number of mismatched system calls within a certain system call sequence length is used as a detection rule [KH97].

This type of a simple numeric measure for an anomaly gives rise to a new form of a mimicry attack. In this evasion method the idea would be not to clump the exploit system calls together, but to pad the exploit code with normal sequences of system calls, thus fooling the detection threshold. As with basic mimicry attacks, this would require knowing specific details of the IDS analysis method, in this case the detection threshold.

Forrest, Hofmeyr et al refined their work later. The key modification was in the definition of an anomaly, i.e. sequence similarity. They noted that since anomalies occur temporally together, they can used the idea of *Hamming distance* for defining how much a sequence differs from the normal sequence. This method is more complex to computer, however. To detect an intrusion, at least one of the sequences generated by the intrusion must be classified as anomalous. They measure the strength of the anomaly by computing minimal Hamming distance and it was assumed that the higher the minimal Hamming distance, the more likely it is that the sequence was actually generated by an intrusion. Therefore the *signal of the anomaly* is the maximum value of the minimal Hamming distances in all the sequences. To make this measure applicable to variable length sequences, this signal value was used normalized over the sequence length value. This normalized value was used as the preferred measure of anomaly [HFS98]. So, if the normalized anomaly signal value is 0.5, it means that at some point half of the trace has been different from the original.

In the more detailed tests, it was found that after a certain length (6), the length of the sequences had very little effect in detection as measured by the normalized signal of the anomaly, so they used a fixed length of 10. In real exploit tests, the normalized signal of an anomaly was between 0.2-0.7 and mismatch percentages 1-38%. In summary, they were able to detect all the tested abnormal behaviors, including successful intrusions, failed intrusion attempts, and unusual error conditions [HFS98]. This of course depended on the correct alert threshold for the signal of the anomaly.

The original *N*-gram method of Forrest et al has been implemented as a special *seq-id* wrapper in the software wrapper IDS of Ko et al [KFBK00]. The wrapper system works in kernel space and is able to intercept and analyze system call sequences. The *seq-id* wrapper causes 5-7% penalty to the execution performance. It can also be used together with a misuse detection or specification based wrapper, but then the wrapper penalties get added.

This method has also been implemented when analysing anomalies in a distributed application

utilizing CORBA [SMS99]. There the observed sequences were not system calls, but CORBA remote method invocation sequences.

Wespi et al have augmented and modified this technique. First of all, they analyzed audit events, not system calls. Secondly, their feature extraction has been done using variable-length sequences and the Teiresias algorithm, which was initially developed for discovering rigid patterns in unaligned biological sequences. The more abstract audit events could be used instead of system calls, since the system call arguments, which were not available in the audit trail, were not used for analysis. This variable-length method has improved the false positive rate [WDD00]. Variable-length sequences were also used by Marceau, but she constructed a finite state automaton from them and the method is thus discussed later [Mar00].

Cabrera et al turned the feature extraction process of this method upside down by using a string matching classifier to extract data to separate *anomaly dictionaries*, i.e. system call sequences that are not in the normal dataset [CLM01]. These anomaly dictionaries correspond to the idea of self for the anomalies instead of self of the monitored system. The incoming system call sequence was considered normal if the anomaly count in the sequence was zero against all anomaly dictionaries. The anomaly count was defined as the average of the relative anomaly counts in the set. The relative anomaly count is the number of anomalous sequences found in the trace, divided by the total number of sequences in the trace. The key problem of this approach is that all new anomalies are classified as one of the old or as normal. This can be solved by a hybrid approach where the anomaly count is first checked against a threshold value. If the anomaly count value was above threshold, the sequence was probably an attack even if it would not be in the anomaly dictionary. It was observed that the String Matching Classifier performs better for longer sequences, while the anomaly count detector performs better for shorter sequences. It was also shown that a hybrid detector, combining both schemes at their best choice of sequence lengths leads to substantial improvement of detection accuracy.

Jones et al also used sequence enumeration techniques, but the sequences were of calls to C-language libraries instead of system calls [JL01]. They utilized three different analysis methods: the original mismatch percentage from Forrest et al, the normalized anomaly signal from Hofmeyr et al and locality frames from Kosoresow et al. In two real tests against Apache web server, the mismatch percentages were 1.5 and 3.9 and the corresponding anomaly signal strengths 0.7 and 0.8. They also observed that the intrusion detection result is stable if the sequence length is >6. Their tests were done using a sequence length of 10. They also made successfull tests with two buffer overflows, one trojan program and one memory depletion denial of service exploit. Additionally, they created a a trojan code attack against `mSQL` and were able to detect that better with library calls than system calls (signal strengths 0.8 vs 0.7, max. locality frame count 20 vs 7). They presume it is because system calls have too little variety in `mSQL`, it is doing basically I/O functions.

Ghosh et al compared in two papers five different algorithms for analysing the system call traces. In their analysis they compared the simple *N*-gram similarity measure technique from Forrest et al [FHSL96] with neural network methods (a backpropagation network and an Elman network) [GSS99]. Later they compared two other machine learning methods, string transducers and a state tester, to the Elman network [GMS00]. The results show that though the similarity matching approach worked fairly well, the performance can be significantly improved (particularly in reducing the false positive rate) by using neural networks [GSS99].

The sequence enumeration and similarity techniques have also been used for analyzing anomalies in user command line behaviour [LB97]. Here the simple equality matching of system calls does not work very well, since the user can produce slightly different versions of the same basic commands. Therefore similarity matching techniques were used to score sequences of commands

for closeness. The closeness measure was MCA-P, match count with adjacency and polynomial bound. While user input has quite different characteristics from system calls, similarity matching techniques may be an option for systems which take into account not only the system call, but also the parameters of the call. This is not quite in the domain of application intrusion detection, but is included here to illustrate other methods used to solve a similar task.

4.2.2 Call policy techniques

One potential approach to execution monitoring is to observe not the call sequences, but to create a list of calls the application makes when run normally, and then in subsequent runs limit the allowed calls to this known good set (i.e. profile).

Call policies have been explored in an operating system setting by Niels Provos in his *Systrace* system. There the system calls an application makes are recorded during a training time thus creating a security policy of the allowed system calls [Pro03]. This procedure is not altogether automated: if the application uses for example random filenames, the policy must be edited to allow that particular form of non-determinism.

The system call arguments are also considered here. Before making a policy decision, the system call and its arguments are translated into a system independent human-readable format. The policy language operates on that translation and does not need to be aware of system call specific semantics [Pro03].

This has been done also in dynamic Java sandboxing which for a given program consists of two activities: sandbox generation and sandbox execution. In the first, a sandbox profile is constructed by running the program with an instrumented JVM. During this training session, profiling information is recorded to the sandbox profile. The sandbox is initially empty and grows during the training run by accumulating records for each unique behavior, i.e. method calls. Because nothing is added that is not observed, each sandbox is customized to a given program and context in which it is executed. During sandbox execution, behavior (i.e. a method call) that is not in the profile is considered anomalous [IF02].

The JVM prototype looks only method invocations and their signatures, although the authors note that also memory behaviour, method arguments, patterns of methods, whole program paths or even other building blocks than methods could be used for anomaly detection [IF02].

This call policy approach assumes that the application run from which the profile was created did not include any attacks or exploits. Additionally, it assumes that any deviations from the profile are potential attacks. To lower the count of false positives, this requires that all variations of the valid application run-time behaviour must be present in the profile. Call policies are a sensible approach in a method or procedure call setting, where the potential number of calls is much greater than the number of system calls and thus the sequence enumeration technique detailed before may not be viable.

This method is somewhat similar to the policy-specification based techniques discussed later in Chapter 4.4, but with the difference that the list of allowed calls is not specified by an administrator but extracted from a known good application run.

4.2.3 State automata techniques

In the normal system call anomaly detection, the near real time speed of the detection is an issue. Therefore the system call traces need to be reduced for quicker processing. For that Kosoresow et al noted that the applications were deterministic in the sense that long sequences of system calls were repeated. This is also intuitive; if the application forks when it gets a connection, this forking

sequence is always repeated. Therefore deterministic finite automaton (DFA) with macros can be used to build patterns of the various system call sequences.

However, creating the exact DFA of a process will be a problem. First, because every possible ordering of system calls would have to be made, the DFA would be very large. This is simply because the system call order can depend on the input data. And second, if the trace length is a function of data the application is processing, every variation in the data length should be included. If this is not done correctly this could lead to false positive alarms. Creating the macros for the DFA is also time consuming and potentially NP-hard [KH97]. For all these reasons deploying this approach to real systems would be difficult.

The simplest work construcing a state automaton was done by Marceau. There variable length *N*-grams of the system call sequences were used to create a suffix tree and an algorithm for creating a finite state automaton (FSA) from it was presented. The resulting "self database" was smaller than the sequence database and thus the detector can be more effective at run-time. The algorithm was adjusted so that it does not create false positives. However, the resulting system was less sensitive to anomalies than the original *N*-gram work [Mar00]. The reason for this may be that the automaton was intentionally weakened by similarity compression so that a specific value for *N* would not be needed, but a general upper limit would suffice.

Various other state automata systems have also been researched. Michael and Ghosh used the *N*-grams from Forrest et al, to create a simple finite state machine. They compared the original work with the state machine and another technique, a string transducer. The results showed that the original *N*-gram work performed best if the detection rate was compared with the false positive rate. However, it had much longer training period than the new methods, i.e. with a limited set of training data the new methods had much better false positive rates. However, none of the techniques was able to detect more than 94% of the intrusions. This was due to limitations in the input data [MG02].

Sekar et al have also explored the use of finite-state machines to aid in the system call sequence analysis [SBDB01]. In their system they augmented the system call information with the program counter (PC), i.e. sequence-giving information, which allowed them to form the state machine automatically.

The actual analysis goes so that if an anomaly in the FSA occurs, it is not immediately flagged as an intrusion, but an anomaly counter is increased. If sufficient number of anomalies are observed during a time frame, an intrusion alert is generated. The anomaly count is decreased as time passes so isolated anomalies do not generate alerts. The system generated less false positives than the *N*-gram method used by Forrest et al. What is also interesting to note that the space and runtime overhead of the algorithm itself is low. However, the overhead for user-level system call interception used here was considerable, 100-250%.

Feng et al enhanced the FSA model by creating the *VtPath* model, in which the use of program counter is augmented and also call stack information is utilized for anomaly detection [FKFG03]. In addition to the system call information and the program counter, the return address is obtained from the call stack into a virtual call stack list, which acts as a history of unreturned function calls. Virtual path between two system calls is the ordered list of non-common return addresses in the call stacks of the two system calls. It is an abstraction of the execution between two system calls.

All found virtual paths and call return addresses are stored during the training phase. In the detection phase each system call causes the virtual call stack being stored. Depending on the contents of the stack, return address and the virtual path between the current call and the previous call, the system can find out various anomalies, such as buffer overflows. The key benefit of the the *VtPath* technique is that it does not suffer from the impossible path problem. In other comparison, the false positive rate is similar to the FSA method, when they corrected some FSA

implementation mistakes Sekar et al made. And since the system call interception was similar to the FSA research, the runtime overhead was still considerable.

Warrender et al compared various techniques in analyzing system call traces, one of the methods was a probablistic automaton called Hidden Markov Model (HMM) [WFP99]. HMM states are unobservable conditions in the system. The model attaches a probability to each state transition and to each type of output in the states. By this probabilistic modeling they can adjust their model over time and also easily represent branches in the system logic. However, the number of states and their probability must be given when the HMM is created, and therefore the training process of a HMM is very long, in the order of hours or days. On the positive side the detection accuracy was very high and the false positive rate was very low.

Also Wagner et Dean and Giffin et al have used state machines in modeling the application for intrusion detection [WD01], [GJM02]. However, their works form a state machine from a program specification, the source or binary code, so we discuss their work with other specification based intrusion detection in Chapter 4.4.

4.2.4 Machine learning

Several machine learning algorithms have been applied to application level intrusion detection as well.

As discussed previously, Ghosh et al compared three different algorithms for detecting the anomalies in the system call sequences: *N*-gram equality matching, backpropagation network and Elman recurrent network. The two latter systems are neural networks [GSS99]. Later, Elman networks were compared to string transducers and state testers [GMS00].

The results show that though the equality matching approach worked fairly well, the performance can be significantly improved (particularly in reducing the false positive rate) by using Elman networks. Elman networks were able to detect 77.3% of all intrusions with no false positives. Further, the Elman nets were able to detect 100.0% of all intrusions with significantly fewer false positives than either of the other two systems [GSS99]. Later, in user-to-root exploit tests, Elman networks achieved 100% detection of attacks very quickly at a false alarm rate of close to 3 per day. With remote-to-local attacks the results were not so good, 30% attacks detected at a rate of approximately 10 false alarms per day [GMS00].

The problem with Elman networks is the training time, it is in the order of thousands of minutes. The detection performance of string transducers were comparable to Elman networks, but the training phase is orders of magnitude smaller, in the range of tens of minutes [GMS00].

Comparing string trancducers and state testers to Forrest et al's *N*-gram matching resulted in *N*-grams achieving slightly better performance. However, the *N*-gram method was much slower to learn, which is to say that it required a great deal more training data to achieve false positive rates comparable to string transducers and state testers [MG02].

Various rule-based systems have also been used. The earliest work used a time-based inductive engine (TIM) to infer rules from VAX/VMS user audit records [TCL90]. This was more of a successfull feasibility study than an actual method performance evaluation. Later Lee, Stolfo and Chan used rule-based machine learning to determine anomalies from the system call traces [LSC97]. Their method was to apply RIPPER rule generation system to the traces to generate rules.

The rules generated from RIPPER can classify whether the trace is "normal" or "abnormal" compared to the rules. But it does not as such tell whether the trace indicates an intrusion or not. Therefore some post-processing was done to determine whether the abnormal traces really were intrusions. These post-processing heuristics try to weed out occasional prediction errors, which

differ from intrusions by the length of the anomaly.

The key finding was that the rules for normal behaviour generated by RIPPER rule generator were well suited to anomaly detection. Additionally the best method for anomaly detection is to classify normal behaviour in RIPPER rules, classification of abnormal behaviour works best for misuse detection [LSC97].

Compared to the work in Forrest et al [FHSL96], this method is both faster and the rules have lower storage requirements. Because there were only 200-280 rules generated by RIPPER and only 100-140 rules on the average had to be checked for classification, authors state that this is potentially doable in a real-time environment. No practical tests have been performed, however.

As an example of applying machine learning to other IDS data, Gunetti and Russo used inductive learning in the form of relational decision trees to classify users into different categories based on their Unix command-line command usage patterns [GR99]. They used a tool called ReliC and achieved about 90% user recognition rate, which is comparable to other similar research.

4.2.5 Data mining

Lee and Stolfo, who previously investigated using machine learning for anomaly detection in system call traces, also extended their work to use data mining techniques [LS98]. The data mining approach relates to the construction of the base classifiers and how to identify meaningful features from the raw data stream. Data mining techniques, such as *association rules* and *frequent episodes* algorithms can be used to help in this feature extraction. An association rule algorithm can help find rules within one audit stream, while frequent episodes reasoning can help discover inter-audit patterns, i.e. patterns between audit data streams. And the latter is especially useful when looking at sequence information, such as system call traces.

Their aim was to make a multi-level model, where a set of base classifiers, such as the system call trace learner, would feed a meta learning task. The meta classifier would take input from all types of base classifiers (including network based mechanisms) and a learning algorithm would be applied for the overall anomaly analysis. This would be able to utilize information from many sources, not just application or network. However, in their work they have only very preliminary results from a *tcpdump* network packet trace but nothing from a system call trace.

4.2.6 Statistical profiling

Anomaly detection in execution monitoring can also be made by creating a statistical profile of the execution, for example a call histogram, and comparing the executing program to the known good profile.

The histogram method has been used by Endler [End98]. All system calls in a call sequence vector were inserted to a large vector so that the occurrence of a call number 25 in the sequence increments the value in the 25th position of the larger vector. The larger vector represents how many times each event of all possible events has occurred in the sequence vector. A histogram classifier was used to divide the sequences to normal and abnormal. However, the histogram system size grows exponentially with the larger vector size so different techniques were used to selecting the most meaningful events. Hand-picking based on experience with buffer overflow attacks proved to be the most successfull. To detect anomalous patterns from the histogram, a value was visually selected from histogram graphs for the anomaly threshold. This could be done, since this work was not real-time detection but work done on audit logs.

Additionally, module interaction can be used for execution monitoring. Elman and Munson have described a technique where any application written in C can be instrumented with a CLIC

toolkit to insert hooks into the application source code modules [EM99]. A kernel-level tool is then able to follow the module interaction and compare the interaction profile statistically to a normal profile obtained beforehand.

4.3 Anomaly detection in input semantics

There have been very few attempts to do semantic intrusion detection of any kind. Here the emphasis would be in analysing the user supplied data contents and its meaning in relation to the application and attempted action. In this sense this method falls under the class of anomaly detection.

Semantic analysis can be done in several levels. The most trivial one is writing these semantic checks into the code. This is of course dependent on the correct identification of potential hazard spots and illegal values. Therefore substantial technical risk analysis, such as demonstrated by Sielken [Sie99] must be done when developing the software. This risk analysis must then be translated to actual agents to detect impossible conditions that could signal an attack.

Although Sielken suggested building the application IDS system so that it contains both application specific and application independent parts, the criticism expressed in Chapter 4.1 with regard to misuse detection in application code is still valid: much design and analysis and some coding is necessary to make this a reality.

Raz et al have investigated semantic anomaly detection in general data feeds, i.e. on input level, such as stock quote streams [RKS02]. They define an anomaly as an observed behavior of a data feed that is different from our expectation. This expectation describes normal behavior of the data feed. Since there was no real semantic data specification for their stock quote feed available, their method was to use two unsupervised machine learning tools, Daikon and Mean to infer invariants from the data feed. After the invariant learning, the feed data was compared against the invariants and an anomaly alert was generated if the invariant did not hold for the received feed data. Their method was able to successfully detect most of the data anomalies in the test scenario, the false positive rate being 0.3. It was usually reduced to under 0.02 (max 0.15) when adding voting heuristics, i.e. cross-checking with another data feed, but it did not help to reduce false negatives.

4.4 Specification based AppIDS

There are also efforts to specifying the application behaviour more or less formally and alerting on any activity outside the specification. Two main styles have been identified; one specifies separately the policy that the application must follow, the other creates a model of the application based on the source or binary code and then observes whether there are any run-time deviations from that model. The found research and their relationships are shown in Figure 4.4.

Classic works in the specification based detection are modeling the network protocol(s) the application uses. This is more in the domain of network based intrusion detection (NIDS), for example [SGF+02] and they are not discussed in more detail here.

4.4.1 Policy based systems

There is an approach where a specification of the priviledged program execution is created and then their execution is monitored via Sun BSM [KFL94]. A policy specification language was also created for this task. The policy lists each system call the application is allowed to make,

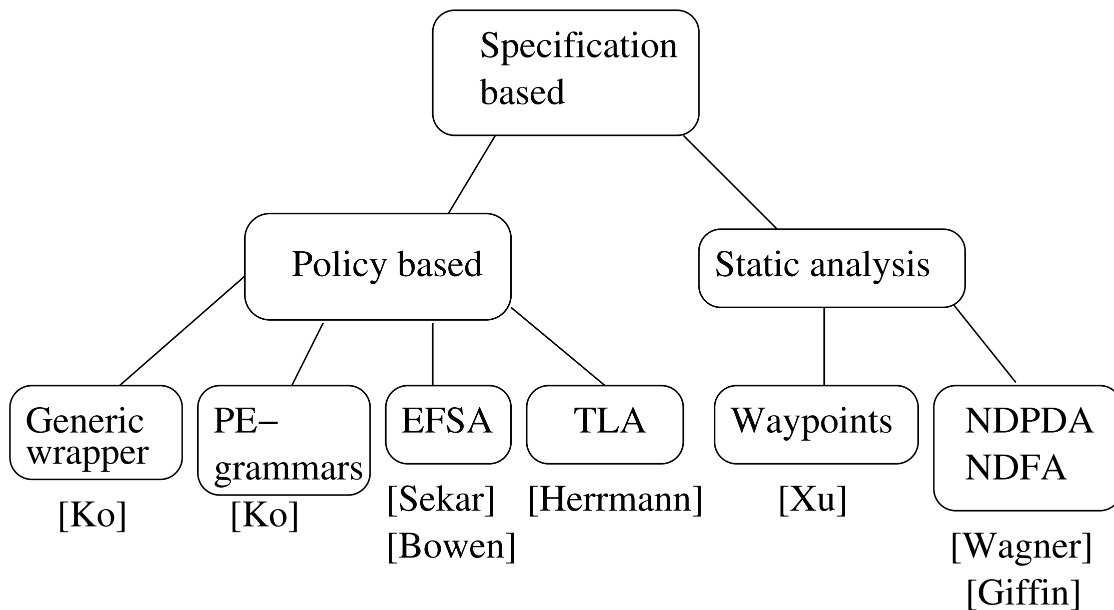


Figure 4.3: The specification based methodology map

including key parameters. The policy language has some semantic information on the application and the kernel API, so it is able to process concepts like users, files and ports. An example policy for `sendmail` is below:

```
#define mailboxdir            ``/usr/spool/mail''
#define mailboxdir            ``/usr/spool/mqueue''
#define mailport              25
#define root_mail_handler     ``/home/root/mail_handler''
PROGRAM sendmail(U)
  read(X) :- worldreadable(X);
  write(X) :- inside(X, mailboxdir);
  write(X) :- inside(X, mailspooldir);
  write(``/etc/sendmail.''+''[\.]*'');
  bind(mailport);
  exec(``/bin/mail'');
  exec(root_mail_handler) :- U.uid = 0;
END
```

Most of these application policy definitions are site independent, i.e. a policy for `sendmail` in one Unix host is very similar to the policy in other Unix hosts, but some variables (such as log file names) are site dependent and must be tailored to fit each system.

Compared to some other system call AppIDS methods, this system does not specify the order of the system calls in the policy file. If the exploit uses allowed system calls in untypical order, or if the policy does not limit the system call parameters enough, or the exploitable problem lies in an application executed by this application, a penetration may still occur unnoticed, i.e. we get a false negative result.

A refinement of the method above was done a couple of years later [KRL97]. There a formalism for specifying the program specific policy (*trace policy*) was developed. The policy language, *parallel environment grammar*, is able to express parallelism and sequencing in the language. As above, it also has semantic devices to test some input values, such as the directory where the file or directory is to be created. They mention the system able to detect simultaneous edits of the same file by two parallel processes in their tests but do not give specifics.

Sekar et al have also modified this method [SBS99, BCS+00]. They defined a language, Auditing Specification Language (ASL) or Behaviour Monitoring Specification Language (BMSL), for application behaviour policy specification and integrated this with a compiler to produce an extended finite state automaton (EFSA) C++ class, from which an application specific system call detection engine is built. This engine then runs as part of the operating system kernel, intercepts system calls and uses the automaton to check them against the defined policy. In actual tests the system call monitoring system (SMS) caused 1.5% overhead. The overhead increases with the number of system calls in the application: an increase of 2000% in system calls caused the overhead to rise to 5% [BCS+00].

Calvin Ko has continued to work on specification based IDS with his team. Their kernel-level software wrapper system can be used also for specification based IDS, since the wrappers are defined in a wrapper definition language (WDL) which can be used to specify the allowed behaviour of the wrapped application [KFBK00]. Also this wrapper definition can access system call parameters and is able to impose semantic restrictions. The performance decrease from the specification based wrapper alone was 5-7% in tests.

All of these systems require some kind of security monitor, analyzer or wrapper to detect the behaviour out of specification, as does the work of Herrmann and Krumm [HK01]. They have a formally defined and simulated security state automata with state dependent security constraints integrated with the container (i.e. wrapper). The security constraints are expressed with a formal specification technique cTLA, which extends Temporal Logic of Actions (TLA).

The basic implementation has Java application beans, adapters, observers and a monitor. There is an adapter wrapped around each bean. The bean interfaces with the environment only through the adapter. Thus all the bean interactions can be observed and if necessary, the bean can be sealed if anything attack-like occurs. The observers check that the bean complies with the cTLA process specification. If the adapter detects an interface event, it forwards it to the observers by calling the corresponding action methods. If all relevant observers signaled the compliance of the event, the adapter really transfers the interface event. If, however, one observer refuses its corresponding action, the adapter seals the bean and reports the violation to the monitor. The monitor interfaces with the system administrator. The observers also link to the trust manager, which determines the intensity of security checks by their wrappers depending on the trust values carried by the scrutinized components [HK01].

The work also extended the Java security manager, since beans can also access system resources via standard streams, thus bypassing the adapter. However, resource access this way is controlled by the Java security manager and the manager was modified to report standard stream accesses.

4.4.2 Static analysis

Static analysis is a form of testing that analyzes the application source code or binary to detect errors or problems. This technique has also been used for application IDS.

Wagner and Dean used source code to create a model of the system call sequences [WD01]. The state machine is created from the source code, so this technique assumes the availability of it.

On the other hand, Giffin et al, use the system binary to create the state machine [GJM02]. Both of these are in essence a form of static analysis. We differentiate this work from the state automata discussed in 4.2.3, since the state machines here are generated from a specification: the source code or the application binary.

The work of Wagner and Dean suggests two non-trivial techniques for creating the state machine; a callgraph method, which creates a non-deterministic finite automaton (NDFA), and an abstract stack method, which creates a non-deterministic pushdown automaton (NDPDA). In this system there are no false alarms, since the state machine will accept all dynamically possible execution traces. On the other hand, the callgraph system will accept also impossible paths. The real problem with these methods is that they are non-trivial to implement and can incur a severe run-time overhead (in the order of minutes) [WD01]. However, the work on static analysis on binaries achieved significantly better efficiency (run-time increase of 13%). The reason for this was identified to differences in operating system library functions (Linux vs. Solaris) [GJM02]. This implies that static analysis is not as universal in practice as it is in theory.

Similarly, a set of kernel-supported waypoints can be created from the source code by static analysis. These function entrance and exit waypoints are then used for the application control-flow monitoring. When the application is run, a push-down automaton of the entrance/exit waypoints is created. An access monitor uses this push-down automaton information for control flow monitoring and what is more novel, permission monitoring. During the static analysis of waypoint generation, also the set of system calls (i.e. permissions) for each function is recorded and the access monitor can ensure system calls invoked in the context of a function appear in its permission set. From the analysis point of view this system has the good point that it can detect global mimicry attacks and return-into-others impossible path attacks [XDC04]. While Xu et al classify their work as anomaly detection, their method of using static analysis for the waypoint generation is more specification based.

However, local mimicry attacks are not detected by this waypoint method proper, although the authors have deployed other means (interface randomization) to counter this problem. The performace tests were also somewhat problematic. When all system calls were monitored, the execution of an ordinary Unix program increased 3-5 times, most increase was in the time spent in kernel mode. By monitoring only dangerous system calls, the overhead was reduced by 16-70%.

Chapter 5

Application IDS Reaction models

There has been very little research done into the application IDS reaction systems, apart from the traditional "send an alert" feature. This is surprising since application level detection mechanisms, which are more or less integrated into the application, could also signal the application to change behaviour when a problem is observed. Some work exists, though.

One system call level example is in the pH system [SF00, Som02], where the mechanism is able to delay all subsequent system calls at kernel level by a configurable time factor if a problem with system call sequences is observed. Additionally, there is a separate congifuration item for the `execve()` call delay threshold which configures the suspicion threshold where the execution of other programs via the monitored program should be blocked. This latter option is very useful against buffer overflows, which try to execute a shell. Although the performance hit of the pH system is several microseconds per system call, the overall performance penalty for real applications was between 4-10% in tests.

One other example is in the context of an auditing subsystem in the Java virtual machine. There is a separate response module that can be used to react to detected attacks in a specific environment. The response module initiates an appropriate response action when an attack or threat coming from a Java application is detected. The module sends to a dedicated thread in the JikesRVM a request for a particular response action [SKV03].

The specification based mechanism by Bowen, Sekar et al [SBS99, BCS+00] has also reactive capabilities. The policy defined in their language (ASL,BMSL) can include directives to delay execution, changing the application scheduling priority, changing the application environment, executing other monitor applications etc. And since the system call interceptor and the policy interpretation program function at kernel level, they are able to enforce these directives.

Chapter 6

Conclusion

While application level intrusion detection systems are a less investigated area than network based intrusion detection systems, a body of research is clearly identifiable. Some of the reviewed work can be considered to belong in the domain of network or host based intrusion detection, but the main body of the reviewed work is clearly focused on the application level.

From the data collection point of view, there are two main categories in the investigated research: one uses some form of an operating system level mechanism to gather application level data and the other uses some kind of controlled execution environment to do the same. Network based data collection and direct monitoring efforts are marginal.

From the analysis point of view the body of research has been mainly focused on one area; detecting anomalies in execution monitoring, which includes monitoring system, library or function calls or module interaction. Misuse detection and specification based systems have also been researched, but they are a very clear minority.

But interestingly enough, reasearch is lacking on the semantic front, there are very few efforts that use any kind of semantic information either in aiding the analysis process or in defining the normal application behaviour. For example, practically all system call analysis mechanisms discard the call parameter information. The use of semantic information in the application level IDS is definitely one area where more research is needed.

Bibliography

[AL01] Almgren, M. and Lindqvist, U., Application-integrated data collection for security monitoring. *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection RAID 2001, LNCS 2212*. Springer-Verlag, 2001, pages 22–36, URL http://springerlink.metapress.com/app/home/content.asp?wasp=5%n5d6ynmwg1xwvf1bd9l&referrer=contribution&format=2&page=1&pagecount=0.

[Axe00] Axelsson, S., Intrusion-detection systems: A survey and taxonomy. Technical report no 99-15, Dept. of Computer Engineering, Chalmers University of Technology, Sweden, 2000.

[BCS+00] Bowen, T., Chee, D., Segal, M., Sekar, R., Shanbhag, T. and Uppuluri, P., Building survivable systems: An integrated approach based on intrusion detection and damage containment. *Proceedings of the DARPA Information Survivability Conference & Exposition, DISCEX'00*, 2000, pages 1084–1099.

[CLM01] Cabrera, J. B. D., Lewis, L., and Mehra, R. K., Detection and classification of intrusions and faults using sequences of system calls. *SIGMOD Record*, 30,4(2001), pages 25–34. URL http://delivery.acm.org/10.1145/610000/604269/p25-cabrera.pdf%?key1=604269&key2=4040363011&coll=GUIDE&dl=ACM&CFID=33931896&CFTOKEN=21280247.

[Den87] Denning, D., An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13,2(1987), pages 222–232. URL http://www.cs.georgetown.edu/denning/infosec/ids-model.rtf.

[EM99] Elbaum, S. and Munson, J. C., Intrusion detection through dynamic software measurement. *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*. USENIX Association, 1999, URL http://www.usenix.org/events/detection99/full_papers/elbaum/e%lbaum.pdf.

[End98] Endler, D., Intrusion detection — applying machine learning to Solaris audit data. *Proceedings of 14th Annual Computer Security Applications Conference*, 1998, pages 268–279, URL http://ieeexplore.ieee.org/xpl/abs_free.jsp?arNumber=738647.

[FBF99] Fraser, T., Badger, L. and Feldman, M., Hardening COTS software with generic software wrappers. *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999, pages 2–16.

[FHS97] Forrest, S., Hofmeyr, S. and Somayaji, A., Computer immunology. *Communications of the ACM*, 40,10(1997), pages 88–96. URL http://portal.acm.org/citation.cfm?id=262793.262811.

[FHSL96] Forrest, S., Hofmeyr, S. A., Somayaji, A. and Longstaff, T., A sense of self for Unix processes. *Proceedings of the 1996 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1996, pages 120–128.

[FKFG03] Feng, H., Koleshnikov, O., Fogla, P. and Gong, W., Anomaly detection using call stack information. *Proceedings of the 2003 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2003, pages 62–75.

[GJM02] Giffin, J. T., Jha, S. and Miller, B. P., Detecting manipulated remote call streams. *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, 2002, pages 61–79.

[GMS00] Ghosh, A. K., Michael, C. and Schatz, M., A real-time intrusion detection system based on learning program behavior. *Proceedings of the 3rd International Symposium on Recent Advances in Intrusion Detection RAID'00. LNCS 1907*. Springer-Verlag, 2000, pages 93–109.

[GR99] Gunetti, D. and Ruffo, G., Intrusion detection through behavioral data. *Proceedings of the Third International Symposium Advances in Intelligent Data Analysis, LNCS 1642*. Springer-Verlag, 1999, pages 383–394, URL http://www.springerlink.com/app/home/contribution.asp?wasp=g1%9tay35wlcyyn9c0vtp&referrer=parent&backto=issue,32,44;journal,1577,1828;linkin%gpublicationresults,1:105633,1.

[GSS99] Ghosh, A. K., Schwartzbard, A. and Schatz, M., Learning program behavior profiles for intrusion detection. *Proceedings of the 1st Workshop on Intrusion Detection*. USENIX Association, 1999, pages 51–62, URL http://www.usenix.org/publications/library/proceedings/detect%ion99/full_papers/ghosh/ghosh_html/.

[HFS98] Hofmeyr, S. A., Forrest, S. and Somayaji, A., Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6,3(1998), pages 151–180.

[HK01] Herrmann, P. and Krumm, H., Trust-adapted enforcement of security policies in distributed component-structured applications. *Proceedings of the 6th IEEE Symposium on Computers and Communications. Hammamet, Tunisia*. IEEE Computer Society Press, 2001, pages 2–8, URL http://ls4-www.cs.uni-dortmund.de/RVS/Pub/TS/ISCC01.pdf.

[IF02] Inoue, H. and Forrest, S., Anomaly intrusion detection in dynamic execution environments. *Proceedings of the 2002 Workshop on New Security Paradigms*. ACM Press, 2002, pages 52–60, URL http://portal.acm.org/ft_gateway.cfm?id=844112&type=pdf&coll=%portal&dl=ACM&CFID=19905017&CFTOKEN=31765782.

[JL01] Jones, A. and Lin, Y., Application intrusion detection using language library calls. *Proceedings of the 17th Annual Computer Security Applications Conference, ACSAC'01*. IEEE Computer Society, 2001, pages 442–449, URL http://ieeexplore.ieee.org/xpl/abs_free.jsp?arNumber=991561.

[KFBK00] Ko, C., Fraser, T., Badger, L. and Kilpatrick, D., Detecting and countering system intrusions using software wrappers. *Proceedings of the 9th USENIX Security Symposium*, 2000, URL http://ieeexplore.ieee.org/iel5/8932/28288/01264947.pdf.

[KFL94] Ko, C., Fink, G. and Levitt, K., Automated detection of vulnerabilities in privileged programs by execution monitoring. *Proceedings of the 10th Annual Computer Security Application Conference*, 1994, pages 134–144, URL http://seclab.cs.ucdavis.edu/papers/pdfs/ck-gf-kl-94.pdf.

[KH97] Kosoresow, A. P. and Hofmeyr, S. A., Intrusion detection via system call traces. *IEEE Software*, 14,5(1997), pages 35–42.

[KRL97] Ko, C., Ruschitzka, M. and Levitt, K., Execution monitoring of security-critical programs in distributed systems: A specification based approach. *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 1997, pages 175–187, URL http://portal.acm.org/citation.cfm?id=884386.

[KS99] Kuperman, B. and Spafford, G., Generation of application level audit data via library interposition. Technical report CERIAS TR 99-11, CERIAS, Purdue University, 1999. URL https://www.cerias.purdue.edu/tools_and_resources/bibtex_arch%ive/archive/99-11.pdf.

[LB97] Lane, T. and Brodley, C. E., Sequence matching and learning in anomaly detection for computer security. *Proceedings of AAAI-97 Workshop on AI Approaches to Fraud Detection and Risk Management*. AAAI Press, 1997, pages 43–49, URL http://citeseer.ist.psu.edu/lane97sequence.html.

[LS98] Lee, W. and Stolfo, S., Data mining approaches for intrusion detection. *Proceedings of the 7th USENIX Security Symposium*. USENIX Association, 1998, pages 79–94, URL http://citeseer.ist.psu.edu/article/lee98data.html.

[LSC97] Lee, W., Stolfo, S. J. and Chan, P. K., Learning patterns from Unix process execution traces for intrusion detection. *Proceedings of the AAAI-97 Workshop on AI Approaches to Fraud Detection and Risk Management*. AAAI Press, 1997, URL http://www1.cs.columbia.edu/ids/publications/wenke-aaai97.ps.

[Mar00] Marceau, C., Characterizing the behavior of a program using multiple-length N-grams. *Proceedings of the 2000 Workshop on New Security Paradigms*. ACM Press, 2000, pages 101–110, URL http://delivery.acm.org/10.1145/370000/366197/p101-marceau.pd%f?key1=366197&key2=8276292011&coll=GUIDE&dl=ACM&CFID=33734302&CFTOKEN=20018340%.

[MG02] Michael, C. C. and Ghosh, A., Simple, state-based approaches to program-based anomaly detection. *ACM Transactions on Information and System Security*, 5,3(2002), pages 203–237. URL http://portal.acm.org/ft_gateway.

cfm%3Fid%3D545187%26type%3Dp%df%26dl%3DGUIDE%26dl%3DACM%26CFID%3D11111111%26CFTOKEN%3D2222222.

[Pro03] Provos, N., Improving host security with system call policies. *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, 2003, pages 257–272, URL http://niels.xtdnet.nl/papers/systrace.pdf.

[RKS02] Raz, O., Koopman, P. and Shaw, M., Semantic anomaly detection in online data sources. *Proceedings of the 24th International Conference on Software Engineering, ICSE'02. Orlando, Florida, May 22-24*, 2002, pages 302–312, URL http://pag.csail.mit.edu/daikon/pubs-using/raz-icse-2002-abst%ract.html.

[SBDB01] Sekar, R., Bendre, M., Dhurjati, D. and Bollineni, P., A fast automaton-based method for detecting anomalous program behaviors. *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001, pages 144–155, URL http://citeseer.ist.psu.edu/sekar01fast.html.

[SBS99] Sekar, R., Bowen, T. and Segal, M., On preventing intrusions by process behavior monitoring. *Proceedings of the USENIX Workshop on Intrusion Detection and Network Monitoring*. USENIX Association, 1999, pages 29–40.

[SF00] Somayaji, A. and Forrest, S., Automated response using system-call delays. *Proceedings of the the 9th USENIX Security Symposium*, 2000, pages 185–197.

[SGF+02] Sekar, R., Gupta, A., Frullo, J., Shanbhag, T., Tiwari, A., Yang, H. and Zhou, S., Specification-based anomaly detection: a new approach for detecting network intrusions. *Proceedings of the 9th ACM conference on Computer and communications security, Washington, DC, USA*, 2002, pages 265–274, URL http://doi.acm.org/10.1145/586110.586146.

[Sie99] Sielken, R., Application intrusion detection. Technical report CS-99-17, Department of Computer Science, University of Virginia, USA, 1999. URL ftp://ftp.cs.virginia.edu/pub/techreports/CS-99-17.ps.Z.

[SKV03] Soman, S., Krintz, C. and Vigna, G., Detecting malicious Java code using virtual machine auditing. *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, 2003, pages 153–168, URL http://cs.ucsb.edu/~ckrintz/papers/usenix03.pdf.gz.

[SMS99] Stillerman, M., Marceau, C. and Stillman, M., Intrusion detection for distributed applications. *Communications of the ACM*, 42,7(1999), pages 62–69.

[Som02] Somayaji, A. B., *Operating System Stability and Security through Process Homeostasis*. Ph.D. thesis, University of New Mexico, 2002.

[Sun94] Sun Microsystems, *SunSHIELD Basic Security Module Guide*. 1994. URL http://docs.sun.com/app/docs/doc/801-6636/6i10gcto4?q=BSM&a=v%iew.

[TCL90] Teng, H. S., Chen, K. and Lu, S. C., Adaptive real-time anomaly detection using inductively generated sequential patterns. *Proceedings of the 1990 IEEE Symposium on*

Security and Privacy, 1990, pages 278–284, URL http://ieeexplore.ieee.org/xpl/abs_free.jsp?arNumber=63857.

[WD01] Wagner, D. and Dean, D., Intrusion detection via static analysis. *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2001, pages 156–168, URL http://ieeexplore.ieee.org/xpl/abs_free.jsp?arNumber=924296.

[WDD00] Wespi, A., Dacier, M. and Debar, H., Intrusion detection using variable-length audit trail patterns. *Proceedings of the the 3rd International Symposium on Recent Advances in Intrusion Detection RAID'00. LNCS 1907*. Springer-Verlag, 2000, pages 110–129, URL http://www.cs.fit.edu/~pkc/id/related/wespi-raid00.pdf.

[WFP99] Warrender, C., Forrest, S. and Pearlmutter, B. A., Detecting intrusions using system calls: Alternative data models. *Proceedings of the 1999 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1999, pages 133–145, URL http://citeseer.ist.psu.edu/article/warrender98detecting.html%.

[XDC04] Xu, H., Du, W. and Chapin, S. J., Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths. *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection, RAID'04. LNCS 3224*. Springer-Verlag, 2004, pages 21–38, URL http://citeseer.ist.psu.edu/702165.html.

[Zam01] Zamboni, D., *Using Internal Sensors for Computer Intrusion Detection*. Ph.D. thesis, Purdue University, 2001. URL http://www.cerias.purdue.edu/homes/zamboni/pubs/thesis-techreport.pdf.