USING INTERNAL SENSORS FOR COMPUTER INTRUSION DETECTION

A Thesis
Submitted to the Faculty of
Purdue University

by Diego Zamboni
CERIAS TR 2001-42

Center for Education and Research in
Information Assurance and Security,
Purdue University
August 2001

USING INTERNAL SENSORS FOR COMPUTER INTRUSION DETECTION

A Thesis

Submitted to the Faculty

of

Purdue University

by

Diego Zamboni

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2001

To my parents for giving me life,

and to Susana for sharing it with me.

ACKNOWLEDGMENTS

As usual, a large number of people were very important for the completion of this thesis work, and I would like to acknowledge at least some of them.

First the official acknowledgments: Portions of the research contributing to this dissertation were supported by the various sponsors of CERIAS, and my stay at Purdue was partially funded by a Fulbright/Conacyt fellowship. Their support is gratefully acknowledged.

I would like to thank my advisor, Eugene Spafford. He received me with open arms from my first day at Purdue, and provided continuous guidance and support throughout my stay here. For that I am very, very grateful.

I would like to acknowledge the other members of my Ph.D. committee: Stephanie Forrest, Mikhail Atallah, Jens Palsberg and Carla Brodley. They provided invaluable feedback and advice for my work and for this dissertation.

Many people contributed significantly to my work, and my deepest gratitude goes to all of them. Susana Soriano, Benjamin Kuperman, Thomas Daniels, Florian Kerschbaum, Rajeev Gopalakrishna and Jim Early provided me with many hours of discussion, continuously questioned my assumptions, and led to many new ideas. The original ideas for how internal sensors would be implemented evolved from discussions with Ben Kuperman, and he also came up with the name "ESP". Florian Kerschbaum poured enormous amounts of work into implementing and testing detectors, and Jim Early implemented the file integrity detector. Angel Soriano guided me through the statistical analysis of the experimental results, and suggested multiple avenues for future research. The staff at the Statistical Consulting Service at Purdue also provided invaluable guidance in the analysis of data. Other students at CERIAS provided support for my work: Sofie Nystrom (who let me use her office), Kevin Du, Hoi Chang, Chapman Flack, Chris Telfer, and many others.

Substantial administrative, technical and logistic support were needed for the completion of my work. The system administrators at CERIAS, Susana Soriano and Vince Koser, always maintained our computers up and running and kept up with my continuous requests and questions, even at times when what they really wanted to do was remove my account and get rid of me. All the administrative personnel at CERIAS, including Mary Jo Maslin, Lori Floyd, Paula Cheatham, Steve Hare and Andra Boehning, always gave me their support. In particular, I would like to thank Marlene Walls, who tirelessly stayed on top of things at CERIAS to make sure everything was going as it should. Without her, my life (and the scheduling for seeing my advisor) would have been infinitely more complicated.

Before and throughout my stay at Purdue there were people who contributed to my career by inspiring, supporting, and challenging me. These include Gerardo Cisneros (who may not know it, but he was my main inspiration for pursuing a Ph.D.); my friends Reynaldo Roel, Carlos González, Claudia Fajardo, Agustín and Adriana Casimiro, Luis and Claudia Graf, Luis and Carmen Teresa Martínez and Eduardo Asbun, all of whom gave me so much support and friendship; Ivan Krsul, Christoph Schuba, Tanya Mastin, Kathy Price, Keith Watson and Robin Sundaram, who gave me a great welcome to the COAST laboratory and helped me through my first years at Purdue. Thank you all.

Who I am is a result of my formation, and for that I have to thank my family. My parents, Laura Zamboni and Gilberto De La Rosa, and my sisters, Ana, Daniela and Inés, have been an inexhaustible source of inspiration, support, advice, and happiness.

And finally, but most importantly, I would like to thank my best friend and wife, Susana Soriano. She lifted me at times when I thought I could not keep going, and her support and care made it possible for me to start and finish this endeavor. She listened patiently to my ideas and tolerated me being locked up in my office 20 hours a day (although she always found ways of spending some time together!), and on top of that, she managed to expertly proofread my dissertation. Susana: you are the love of my life, and words cannot express how much I need to thank you.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

LIST OF DEFINITIONS

ABSTRACT

Zamboni, Diego. Ph.D., Purdue University, August, 2001. Using Internal Sensors for Computer Intrusion Detection. Major Professor: Eugene H. Spafford.

This dissertation introduces the concept of using internal sensors to perform intrusion detection in computer systems. It shows its practical feasibility and discusses its characteristics and related design and implementation issues.

We introduce a classification of data collection mechanisms for intrusion detection systems. At a conceptual level, these mechanisms are classified as direct and indirect monitoring. At a practical level, direct monitoring can be implemented using external or internal sensors. Internal sensors provide advantages with respect to reliability, completeness, timeliness and volume of data, in addition to efficiency and resistance against attacks.

We introduce an architecture called ESP as a framework for building intrusion detection systems based on internal sensors. We describe in detail a prototype implementation based on the ESP architecture and introduce the concept of embedded detectors as a mechanism for localized data reduction.

We show that it is possible to build both specific (specialized for a certain intrusion) and generic (able to detect different types of intrusions) detectors. Furthermore, we provide information about the types of data and places of implementation that are most effective in detecting different types of attacks.

Finally, performance testing of the ESP implementation shows the impact that embedded detectors can have on a computer system. Detection testing shows that embedded detectors have the capability of detecting a significant percentage of new attacks.

1. INTRODUCTION

It is feasible to perform computer intrusion detection at the host level for both known and new attacks using internal sensors and embedded detectors with reasonable CPU and size overhead. The present document discusses this assertion in detail, and describes the work done to show its validity.

1.1 Background and problem statement

The field of intrusion detection has received increasing attention in recent years. One reason is the explosive growth of the Internet and the large number of networked systems that exist in all types of organizations. The increased number of networked machines has led to a rise in unauthorized activity [20], not only from external attackers, but also from internal sources such as disgruntled employees and people abusing their privileges for personal gain [113].

In the last few years, a number of intrusion detection systems have been developed both in the commercial and academic sectors. These systems use various approaches to detect unauthorized activity and have given us some insight into the problems that still have to be solved before we can have intrusion detection systems that are useful and reliable in production settings for detecting a wide range of intrusions.

Most of the existing intrusion detection systems have used central data analysis engines [e.g. 32, 86] or per-host data collection and analysis components [e.g. 60, 112] that are implemented as separate processes running on one or more of the machines in a network. In their design and implementation, all of these approaches are subject to a number of problems that limit their scalability, reliability and resistance to attacks.

At CERIAS (Center for Education and Research in Information Assurance and Security) at Purdue University, we have developed a monitoring technique called *internal sensors* based on source code instrumentation. This technique allows the close observation of data and behavior in a program. It can also be used to implement intrusion detection systems that perform their task in near real-time, that are resistant to attacks and that impose a reasonably low overhead in the hosts, both in terms of memory and CPU utilization.

This dissertation describes the concept of using internal sensors for building an intrusion detection framework at the host level, their characteristics and abilities, and experimental results in an implementation.

1.2 Basic concepts

First, we introduce some basic concepts on which this dissertation is based.

1.2.1 Intrusion detection

Intrusion detection has been defined as "the problem of identifying individuals who are using a computer system without authorization (i.e., 'crackers') and those who have legitimate access to the system but are abusing their privileges (i.e., the 'insider threat')" [93].

We add to this definition the identification of *attempts* to use a computer system without authorization or to abuse existing privileges. Therefore, our working definition of *intrusion* matches the one given by Heady et al. [59]:

WORKING DEFINITION 1.1: INTRUSION

> Any set of actions that attempt to compromise the integrity, confidentiality, or availability of a computer resource.

This definition disregards the success or failure of those actions, so it also corresponds to attacks against a computer system. In the rest of this dissertation we use the terms *attack* and *intrusion* interchangeably.

The definition of intrusion results in our working definition of intrusion detection:

WORKING DEFINITION 1.2: INTRUSION DETECTION

> The problem of identifying actions that attempt to compromise the integrity, confidentiality, or availability of a computer resource.

The definition of the word *intrusion* in an English dictionary [96] does not include the concept of an insider abusing his or her privileges or attempting to do so. A more accurate phrase to use is *intrusion and insider abuse detection*. In this document we use the term *intrusion* to mean both intrusion and insider abuse.

WORKING DEFINITION 1.3: INTRUSION DETECTION SYSTEM

> A computer system (possibly a combination of software and hardware) that attempts to perform intrusion detection.

Most intrusion detection systems try to perform their task in real time [93]. However, there are also intrusion detection systems that do not operate in real time, either because of the nature of the analysis they perform [e.g. 76] or because they are geared for forensic analysis (analysis of what has happened in the past on a system) [e.g. 48, 147].

The definition of an intrusion detection system does not include preventing the intrusion from occurring, only detecting it and reporting it to an operator. There are some intrusion detection systems [e.g. 25, 135] that try to react when they detect an unauthorized action. This reaction usually includes trying to contain or stop the damage, for example, by terminating a network connection.

Intrusion detection systems are usually classified as host-based or network-based [93]. Host-based systems base their decisions on information obtained from a single host (usually audit trails), while network-based systems obtain data by monitoring the traffic in the network to which the hosts are connected.

1.2.2 Desirable characteristics of an intrusion detection system

The following characteristics are ideally desirable for an intrusion detection system (based on the list provided by Crosbie and Spafford [34]):

1. It must *run continually* with minimal human supervision.

2. It must be *fault tolerant*:

   (a) The intrusion detection system must be able to recover from system crashes, either accidental or caused by malicious activity.

   (b) After a crash, the intrusion detection system must be able to recover its previous state and resume its operation unaffected.

3. It must *resist subversion*:
   (a) There must be a significant difficulty for an attacker to disable or modify the intrusion detection system.
   (b) The intrusion detection system must be able to monitor itself and detect if it has been modified by an attacker.
4. It must impose a *minimal overhead* on the systems where it runs to avoid interfering with their normal operation.
5. It must be *configurable* to accurately implement the security policies of the systems that are being monitored.
6. It must be *easy to deploy*. This can be achieved through portability to different architectures and operating systems, through simple installation mechanisms, and by being easy to use and understand by the operator.
7. It must be *adaptable* to changes in system and user behavior over time. For example, new applications being installed, users changing from one activity to another, or new resources being available can cause changes in system use patterns.
8. It must be *able to detect attacks*:
   (a) The intrusion detection system must not flag any legitimate activity as an attack (false positives).
   (b) The intrusion detection system must not fail to flag any real attacks as such (false negatives). It must be difficult for an attacker to mask his actions to avoid detection.
   (c) The intrusion detection system must report intrusions as soon as possible after they occur.
   (d) The intrusion detection system must be general enough to detect different types of attacks.

We will refer to these characteristics throughout this dissertation for description of different intrusion detection architectures and systems, including those developed for this dissertation.

1.3 Problems with existing intrusion detection systems

Most existing intrusion detection systems (for example, those surveyed by Axelsson [7], plus others [e.g. 9, 137]) suffer from at least two of the following problems:

First, the information used by the intrusion detection system is obtained from audit trails or from packets on a network. Data has to traverse a longer path from its origin to the intrusion detection system, and in the process can potentially be destroyed or modified by an attacker. Furthermore, the intrusion detection system has to infer the behavior of the system from the data collected, which can result in misinterpretations or missed events. We refer to this as the *fidelity* problem. It corresponds to a failure to properly address desirable characteristic #8.

Second, the intrusion detection system continuously uses additional resources in the system it is monitoring even when there are no intrusions occurring, because the components of the intrusion detection system have to be running all the time. This is the *resource usage* problem and corresponds to a failure in addressing desirable characteristic #4.

Third, because the components of the intrusion detection system are implemented as separate programs, they are susceptible to tampering. An intruder can potentially disable or modify the programs running on a system, rendering the intrusion detection system useless or unreliable. This is the *reliability* problem and corresponds to a failure to address desirable characteristics #1, #2 and #3.

In this dissertation, we describe a mechanism that addresses all three of these problems and has several other desirable characteristics.

1.4 Thesis statement

This dissertation describes the work done to show the validity of the following two hypotheses:

1. It is possible to use internal sensors in a host to perform intrusion detection in a way that addresses the fidelity, resource usage and reliability problems described in Section 1.3.

2. Internal sensors can be used to detect both known and new attacks against a host.

In this context, *intrusion detection* is used as defined in Section 1.2.1. *Internal sensor* is a concept defined in Section 2.1.4.

1.5 Document organization

This dissertation follows: Chapter 1 introduces some basic concepts, the problems to address, and the thesis statement. Chapter 2 presents a classification of the existing architectures for the data collection and data analysis phases of intrusion detection systems, and provides the motivation and justification for the work described in this dissertation. It also presents related work. Chapter 3 describes the ESP architecture and its characteristics. Chapter 4 describes the prototype implementation for an intrusion detection system based on the ESP architecture, and the results of that implementation. Chapter 5 describes the performance and detection tests that were done to evaluate the properties of the ESP implementation, and how they relate to the predicted properties described in Chapter 3. Finally, Chapter 6 presents the conclusions, summarizes the contributions of this dissertation, and discusses directions for future research.

2. RELATED WORK AND ARCHITECTURES FOR INTRUSION DETECTION

Intrusion detection is conceptually—and in practice, in most cases—performed in two distinct phases: data collection and data analysis. Intrusion detection systems can be classified according to how they are structured in each of those phases.

In this chapter, we describe the different structures that an intrusion detection system can have in both data collection and data analysis and draw some conclusions that guide the rest of this dissertation. We also mention related work as appropriate. Tables 2.2 and 2.1 summarize how different existing intrusion detection systems are classified according to some of the architectures described in this chapter.

2.1 Data collection architectures

The performance of an intrusion detection system can only be as good in terms of desirable characteristics #2, #3, #7, and #8 (see Section 1.2.2) as the data on which it bases its decisions. For this reason, the way in which data is obtained is an important design decision in the development of intrusion detection systems. If the data is acquired with a significant delay, detection could be performed too late to be useful. If the data is incomplete, detection abilities could be degraded. If the data is incorrect (because of error or actions of an intruder), the intrusion detection system could stop detecting certain intrusions and give its users a false sense of security. Unfortunately, these problems have been identified in existing products. After examining the needs of different intrusion detection systems and the data provided by different operating systems, Price concluded that “the audit data supplied by conventional operating systems lack content useful for misuse detection.” [114, p. 107]

With the goal of better understanding the characteristics that make data collection mechanisms suitable for intrusion detection, in this chapter we provide two conceptual (centralized/distributed and direct/indirect) and two practical (host/network-based and ex-

Table 2.1
Classification of some existing intrusion detection systems according to the data analysis structure and data collection structure architectures described in Sections 2.1.1 and 2.2.

		Data analysis structure	
		Centralized (75%)	**Distributed (25%)**
Data collection structure	Centralized (47%)	ACME! [14], ALVA [90], ASAX [56], AppShield [124], BlackICE Sentry [95], Bro [107], CERN NSM [91], CaptIO [15], CompWatch [42], Defense Worx [129], Haystack [132], Hyperview [39], IDIOT [79], Janus [54], LANguard, LANguard SELM [82], LIDS [68], MIDAS [128], NID [28], NSM [60], Network Security Agent [149], OpenSnort Sensor [136], OpenWall Kernel patch for Linux [105], POLYCENTER [41], Psionic HostSentry [116], Psionic Logcheck/LogSentry [116], Psionic PortSentry [116], SecureNet Pro [88], Snort [121], T-sight [44], Tripwire [76], USTAT [70], Wisdom & Sense [150], alert.sh for FW-1 [138], auditGUARD [38], eTrust ID [27], pH [135]	
	Distributed (53%)	ADS [118], AID [134], CIDDS, CMDS [115], CyberCop Monitor [110], CyberTrace [123], CylantSecure [154], Entercept [46], IDA [6], IDES [86], Intruder Alert [148], Kane Security Monitor [29], Manhunt [119], NADIR [63], NIDES [3], NSTAT [74], NetProwler [148], NetRanger [25], PRCis [84], Shadow [100], UNICORN [22], eTrust Audit [26]	AAFID [137], AFJ [4], CARDS [156], CSM [153], Centrax [53], DIDS [133], DPEM [77], Dragon [45], EMERALD [112], FormatGuard [31], GrIDS [139], HP IDS/9000 [61], Hummer [51], JiNao [73], LISYS [64], NFR [99], NetSTAT [152], RealSecure [71], StormWatch [101]

Table 2.2
Classification of some existing intrusion detection systems according to the architectures they use for their data collection mechanisms and their data analysis structure, as described in Sections 2.1.2, 2.1.3, 2.1.4 and 2.2. Note that the assigned percentages for data collection mechanisms do not add to 100% because some intrusion detection systems use more than one type of data collection mechanism.

			Data analysis structure	
			Centralized (75%)	**Distributed (25%)**
Data collection mechanisms	**Indirect (85%)**	Network-based (44%)	ACME! [14], BlackICE Sentry [95], Bro [107], CERN NSM [91], CIDDS, CaptIO [15], CyberCop Monitor [110], CyberTrace [123], Defense Worx [129], LANguard, LANguard SELM [82], Manhunt [119], NID [28], NSM [60], NetProwler [148], NetRanger [25], Network Security Agent [149], OpenSnort Sensor [136], SecureNet Pro [88], Shadow [100], Snort [121], T-sight [44], eTrust ID [27]	AAFID [137], AFJ [4], Centrax [53], DIDS [133], Dragon [45], EMERALD [112], GrIDS [139], Hummer [51], LISYS [64], NFR [99], NetSTAT [152], RealSecure [71]
		Host-based (52%)	ADS [118], AID [134], ALVA [90], ASAX [56], CMDS [115], CompWatch [42], CyberCop Monitor [110], Haystack [132], Hyperview [39], IDA [6], IDES [86], IDIOT [79], Intruder Alert [148], Kane Security Monitor [29], MIDAS [128], NADIR [63], NIDES [3], NSTAT [74], POLYCENTER [41], PRCis [84], Psionic HostSentry [116], Psionic Logcheck/LogSentry [116], UNICORN [22], USTAT [70], Wisdom & Sense [150], alert.sh for FW-1 [138], auditGUARD [38], eTrust Audit [26]	AAFID [137], CARDS [156], CSM [153], Centrax [53], DIDS [133], DPEM [77], Dragon [45], EMERALD [112], GrIDS [139], HP IDS/9000 [61], Hummer [51], JiNao [73], RealSecure [71]
	Direct (18%)	Host-based external (10%)	AppShield [124], Entercept [46], Janus [54], Psionic PortSentry [116], Tripwire [76]	AAFID [137], HP IDS/9000 [61], StormWatch [101]
		Host-based internal (8%)	CylantSecure [154], LIDS [68], OpenWall Kernel patch for Linux [105], pH [135]	FormatGuard [31]

ternal/internal) classifications of data collection mechanisms. We discuss the advantages and disadvantages of each one of them.

The term *monitored component* is used in this chapter and in the rest of this dissertation as follows:

WORKING DEFINITION 2.1: MONITORED COMPONENT

A host or a program that is being monitored by an intrusion detection system.

By this definition, an intrusion detection system that is explicitly monitoring several programs in a host (for example, the kernel, the email server and the HTTP server) could be considered as monitoring several components even if they are all in the same host.

2.1.1 Data collection structure: centralized and distributed

When talking about data collection architectures for intrusion detection, the classification normally refers to the locus of data collection. The following definitions are based on those provided by Axelsson [7].

DEFINITION 2.2: CENTRALIZED DATA COLLECTION

Data used by the intrusion detection system is collected at a number of locations that is fixed and independent of the number of monitored components.

DEFINITION 2.3: DISTRIBUTED DATA COLLECTION

Data used by the intrusion detection system is collected at a number of locations that is directly proportional to the number of monitored components.

In these definitions, a *location* is defined as an instance of running code. So for example, a data collection mechanism implemented in a shared library could be considered as distributed provided that the library will be linked against multiple programs, because each running program will execute the mechanism separately. However, if the shared library will be linked against a single program and it will collect all the information needed by the intrusion detection system, we would consider it as centralized data collection. We can see that these definitions depend not only on how the data collection components are implemented, but also on how they are used.

As can be seen in Table 2.1, both distributed and centralized data collection have been widely used in existing intrusion detection systems and have almost equal percentages in the systems listed in the table. A report by Axelsson [7] shows that the trend over the years has been towards distributed intrusion detection systems, which need distributed data collection.

The distinction between centralized and distributed data collection is the feature most commonly used for describing the data collection capabilities of an intrusion detection system. However, for the purposes of this dissertation, we are more interested in discussing the mechanisms used to perform data collection and the data sources utilized. These are described in the next sections.

2.1.2 Data collection mechanisms: direct and indirect monitoring

In the physical world, a direct observation is one in which we can use one or more of our senses to observe or measure a phenomenon, and an indirect observation is one in which we rely on a tool or on an observation by someone (or something) else to obtain the information.

We build similar definitions in the context of data collection for intrusion detection. When an intrusion detection system can measure a condition or observe behavior in the monitored component by obtaining data directly from it, we use the term *direct monitoring*. When the intrusion detection system relies on a separate mechanism or tool for obtaining the information, we use the term *indirect monitoring*. In other words, direct monitoring is the measurement or observation of a characteristic of an object, and indirect monitoring is the measurement or observation of the effects of the object having that characteristic.

For example, using the **ps** [146] command to observe CPU load on a Unix host is considered a case of direct monitoring because **ps** directly extracts the load data from the corresponding data structures in the kernel. By comparison, if the CPU load is recorded in a log file and later read from there, we consider it a case of indirect monitoring because we are relying on a separate mechanism (in this case, a file) for the observation.

Based on the above discussion, we state that all data collection methods can be classified as direct or indirect according to the following definitions:

DEFINITION 2.4: DIRECT MONITORING

The observation of the monitored component by obtaining data directly from it.

DEFINITION 2.5: INDIRECT MONITORING

The observation of the monitored component through a separate mechanism or tool.

Common examples of mechanisms through which indirect monitoring can be performed are log files and network packets. The data obtained from these mechanisms is an effect of the data having been present in the components that generated the data. If the data were obtained directly from the component that generated them (for example, by reading the appropriate data structures on the host before a packet is sent to the network), we would be performing direct monitoring.

To perform intrusion detection, direct monitoring is better than indirect monitoring for the following reasons:

Reliability: Data from an indirect data source (for example, a log file) could potentially be altered by an intruder before the intrusion detection system uses it. It could also be affected by non-malicious failures. For example, a disk becoming full or a log file being renamed could make the data unavailable to the intrusion detection system.

Completeness: Some events may not be recorded on an indirect data source. For example, not every action of the **inetd** daemon gets recorded to a log file.

Furthermore, an indirect data source may not be able to reflect internal information of the object being monitored. For example, a TCP-Wrappers [151] log file cannot reflect the internal operations of the **inetd** daemon. It can only contain data that is visible through its external interfaces. While that information may be sufficient for some purposes (for example, knowing what address a request came from), it may not be sufficient for others (for example, knowing which specific access rule caused a request to be denied).

Volume: With indirect monitoring, the data is generated by mechanisms (for example, the code that writes the audit trail) that have no knowledge of the needs of the intrusion detection system that will be using the data. For this reason, indirect data sources usually carry a high volume of data. For example, Kumar and Spafford [80] mention that a C2-generated audit trail might contain 50K-500K records per user per day. For a modest-size user community, this could amount to hundreds of megabytes of audit data per day, as pointed out by Mounji [92].

For this reason, when indirect data sources are used, the intrusion detection system has to spend more resources in filtering and reducing the data even before being able to use them for detection purposes.

A direct monitoring method has the ability to select and obtain only the information it needs. As a result, smaller amounts of data are generated. Additionally, the monitoring components could partially analyze the data themselves and only produce results when relevant events are detected. This would practically eliminate the need for storing data other than for forensic purposes.

Scalability: The larger volume of data generated by indirect monitoring results in a lack of scalability. As the number of hosts and monitoring elements grows, the overhead resulting from filtering data can cause degradation in the performance of the hosts being monitored or overload of the network on a centralized intrusion detection system.

Timeliness: Indirect data sources usually introduce a delay between the moment the data is produced and when the intrusion detection system can have access to them. Direct monitoring allows for shorter delays and enables the intrusion detection system to react in a more timely fashion.

However, as can be seen in Table 2.2, there is a notable disparity in the utilization of direct and indirect monitoring in intrusion detection systems. Less than 20% of the intrusion detection systems surveyed use some form of direct monitoring. This can be attributed to the main disadvantage of direct monitoring: complexity of implementation. Direct monitoring mechanisms have to be designed in a more specific manner to the monitored component and the type of information that it generates. Evidence to this is that most of the intrusion detection systems in Table 2.2 (except for CylantSecure [154] and pH [135]) that use direct monitoring are tailored for detecting specific types of attacks.

2.1.3 Data collection mechanisms: host-based and network-based

In practice, data collection methods are commonly classified as host-based or network-based according to the following definitions:

DEFINITION 2.6: HOST-BASED DATA COLLECTION

> The acquisition of data from a source that resides on a host, such as a log file, the state of the system or the contents of memory.

DEFINITION 2.7: NETWORK-BASED DATA COLLECTION

> The acquisition of data from the network. Usually done by capturing packets as they flow through it.

Most of the intrusions detected by intrusion detection systems are caused by actions performed in a host. For example: executing an invalid command or accessing a service and providing it malformed or invalid data. The attacks act on the end host although they may occur over a network.

Also, there is the case of attacks that act on the network infrastructure components such as routers and switches. Most of those components can be considered as hosts, and they have the ability to perform monitoring tasks on themselves [11]. Therefore, attacks on the network infrastructure can also be considered as acting on hosts. In cases where the network infrastructure components cannot perform monitoring tasks (for example, because they are not programmable), attacks on those components can only be detected using network-based data collection because the attacks do not act directly on any other hosts in the network. For the rest of our discussion, we will consider routers and switches as hosts.

The only attacks that act on the network itself are those that flood the network to its capacity and prevent legitimate packets from flowing. However, most of these attacks can also be detected at the end hosts. For example, a Smurf attack [69] could be detected at the ICMP layer in the host by looking for the occurrence of a large number of ECHO_RESPONSE packets.

In general, it is advisable to use host-based data collection for the following reasons (see also the discussion by Daniels and Spafford [37]):

- Host-based data collection allows collecting data that reflect accurately what is happening on the host, instead of trying to deduce it based on the packets that flow through the network.
- In high-traffic networks, a network monitor could potentially miss packets, whereas properly implemented host monitors can report every single event that occurs on each host.
- Network-based data collection mechanisms are subject to insertion and evasion attacks, as documented by Ptacek and Newsham [117]. These problems do not occur on host-based data collection because they act on data that the host already has.
- If the data needed by the intrusion detection system flows through disjoint paths (as might be the case with a network with multiple gateways or when using switching hubs), performing network-based data collection can become difficult and unreliable, and the task of unifying the data coming from different collectors for use by the intrusion detection system may not be trivial.
- The use of encryption renders network-based data collection mechanisms ineffective because they cannot examine the contents of encrypted communications.
- Network-based data collection mechanisms cannot observe actions that occur inside a host, so they will miss local attacks.

Network-based data collection also has some advantages, including the following:

- An intrusion detection system that uses network-based data collection can be deployed on an existing network without having to make any changes to the hosts. For this reason, a large number of commercial intrusion detection systems use network-based data collection.

- A network-based data collection component can be completely invisible to other hosts (this can be achieved even at the hardware level), providing a convenient vantage point from which to observe the actions on the network.

We consider network-based data collection as a form of indirect monitoring because the network traffic is an effect of the data and activity at the hosts (see Definition 2.5). In a more general sense, the advantages and disadvantages just described reflect the distinction between direct and indirect data collection.

The relationship between the traditional host-based/network-based classification of intrusion detection systems [93] and the types of monitoring described in Section 2.1.2 is as follows and can be seen in Table 2.2: Intrusion detection systems normally considered as “network-based” correspond to Indirect/Network-based monitoring mechanisms, whereas Indirect/Host-based and all Direct monitoring mechanisms correspond to the “host-based” intrusion detection systems.

Both host-based and network-based data collection have been widely used in intrusion detection systems. In recent years, an increasing number of intrusion detection systems have started to use both host-based and network-based components in an attempt to obtain the most complete view of the hosts being monitored.

The architecture described in this dissertation corresponds to a system that uses host-based data collection.

2.1.4 Data collection mechanisms: external and internal sensors

All direct monitoring methods are host-based. Direct monitoring of a host can be done using external or internal sensors according to the following definitions:

DEFINITION 2.8: EXTERNAL SENSOR

> A piece of software that observes a component (hardware or software) in a host and reports data usable by an intrusion detection system, and that is implemented by code separate from that component.

Definition 2.9: Internal sensor

> A piece of software that observes a component (hardware or software) in a host and reports data usable by an intrusion detection system, and that is implemented by code incorporated into that component.

For example, a program that uses the **ps** command [146] to obtain process information on a Unix system could be considered an external sensor. If the process-information gathering component was built into the Unix kernel, it would be considered an internal sensor. A library wrapper [81] is considered as an external sensor because its code is separate from that of the program it monitors. According to our definitions, an internal sensor could also be built into hardware components; for example, in the firmware of a network interface card.

Internal sensors are part of the source code of the monitored component. They can be added to an already existing program, and in that case they can be considered as a case of source code instrumentation. Ideally, internal sensors should be added during development of the program when the cost and effort of making changes and fixing errors is lower [108]. Also, at that point the sensors could be added by the original authors of the program instead of by someone else—who would have the added cost of understanding the program first.

Note that by our definitions, any portion of a program can be considered as an internal sensor, as long as it provides data that can be used by an intrusion detection system. No specification is made about how the data should be produced or transmitted.

External and internal sensors for direct data collection have different strengths and weaknesses and can be used together in an intrusion detection system. Tables 2.3 and 2.4 list the advantages and disadvantages of each type of sensor.

From the point of view of software engineering, internal and external sensors present different characteristics in the following aspects:

Introduction of errors: It is potentially easier to introduce errors in the operation of a program through the use of internal sensors because the code of the program being monitored has to be modified. Errors can also be introduced by external sensors (for

Table 2.3
Advantages and disadvantages of external sensors.

External sensors	
Advantages	**Disadvantages**
• Easily modified, added or removed from a host. • Can be implemented in any programming language that is appropriate for the task.	• There is a delay between the generation of the data and their use because after the data are produced they have to be made available on an external source before a sensor can access them. • The information can potentially be modified by an intruder before the sensor obtains it (for example, if the data are read from a log file). • Can potentially be disabled or modified by an intruder. • Added performance impact because the sensors are separate components—processes, threads, or loaded libraries–possibly running continuously. • Limited access to information because they depend on existing mechanisms (such as Unix commands or system calls) to obtain it.

Table 2.4
Advantages and disadvantages of internal sensors.

Internal sensors	
Advantages	**Disadvantages**
• Minimum delay between the generation of the information and its use because it can be obtained at its source. • It is practically impossible for an intruder to modify data to hide his tracks because data are never stored on an external medium before the sensor obtains them. • Cannot be easily disabled or modified because they are not separate processes. • Network traffic and processing load are reduced because embedded sensors can look for specific pieces of information instead of reporting generic data for analysis. Also, they can partially analyze the data at the moment of acquisition. • Embedded sensors do not cause a continuous CPU overhead because they are only executed when required. This makes it possible to incorporate a larger number of sensors on a single host. • Because they are implemented as part of the program they are monitoring, they can access any information that is necessary for their task.	• Their implementation requires access to the source code of the program that needs to be monitored. • Arguably harder to implement because they require modifications to the program being monitored. However, if the sensors are added during development of the program, this problem is reduced. • Need to be implemented in the same language as the program they are going to monitor. • If designed or implemented incorrectly, they can severely harm the performance or the functionality of the program they are part of. • Harder to update or modify and to port to different operating systems, or even to different versions of the same program. • Reduced portability, because the sensors depend on the specifics of the code where they are implemented.

example, an agent that consumes an excessive amount of resources, or an interposed library call that incorrectly modifies its arguments). We claim that most internal sensors can be fairly small pieces of code. Their size allows them to be extensively checked for errors. Also, this problem would be reduced if sensors were added during development of the program instead of afterwards.

Maintenance: External sensors are easier to maintain independently of the program they monitor because they are not part of it. However, when internal changes to the program occur, it can be simpler to update internal sensors (which can be changed at the same time the program is modified) than external sensors (which have to be kept up to date separately).

Size: Internal sensors can be smaller (in terms of code size and memory usage) than external sensors because they become part of an existing program; thus, avoiding the base overhead associated with the creation of a separate process.

Completeness: Internal sensors can access any piece of information in the program they are monitoring whereas external sensors are limited to externally-available data. For this reason, internal sensors can have more complete information about the behavior of the monitored program. Furthermore, because internal sensors can be placed anywhere in the program they are monitoring, their coverage can be more complete than that of an external sensor which can only look at the program "from the outside."

Correctness: Because internal sensors have access to more complete data, we expect them to produce more correct results than external sensors, which often have to act based on incomplete data.

External sensors are better in terms of ease of use and maintainability whereas internal sensors are superior in terms of monitoring and detection abilities, resilience and host impact. Both types of sensors can be used in an intrusion detection system to take advantage of their strengths according to the specific task each sensor has to accomplish.

We can see in Table 2.2 that a small percentage of the intrusion detection systems surveyed use internal sensors and most of those were designed for detecting specific types of attacks—the two exceptions are CylantSecure [154] which uses internal sensors for collecting information analyzed externally, and pH [135], which fully implements data collection and detection using internal sensors, and is a good example of the potential of internal sensors. This can be attributed to the considerable difficulty in the implementation of internal sensors: the monitored components themselves have to be modified. On closed-source systems, this is impossible unless the vendor provides the modifications, and on open-source systems it can be cumbersome and time consuming.

2.2 Data analysis architectures

Intrusion detection systems are classified as centralized or distributed with respect to how the data analysis components are distributed, as follows [137]:

DEFINITION 2.10: CENTRALIZED INTRUSION DETECTION SYSTEM

An intrusion detection system in which the analysis of the data is performed in a number of locations that is fixed and independent of the number of monitored components.

DEFINITION 2.11: DISTRIBUTED INTRUSION DETECTION SYSTEM

An intrusion detection system in which the analysis of the data is performed in a number of locations that is directly proportional to the number of monitored components.

Note that these definitions are based on the number of monitored components and not of hosts (as has been traditionally the case), so it is feasible to have an intrusion detection system that uses distributed data analysis within a single host if the analysis is performed in different components of the system.

In the definitions above, a *location* is defined as an instance of running code. So for example, an analysis component implemented in a shared library could be considered as a distributed analysis component if the library will be linked against multiple programs, because each running program will execute the analysis component separately. However, if the shared library will be linked against a single program and all the data analysis will occur there, we would consider it as centralized analysis. So we can see that these definitions depend not only on how the analysis components are implemented, but also on how they are used.

Both distributed and centralized intrusion detection systems may use host- or network-based data collection methods, or a combination of them.

Some strengths and weaknesses of centralized and distributed intrusion detection systems are shown in Table 2.5.

It can be observed from Table 2.1 that the vast majority of intrusion detection systems surveyed use centralized data analysis. This can be attributed to the difficulty in the implementation of a distributed analysis mechanism.

Most weaknesses of distributed intrusion detection systems can be overcome through technical means whereas centralized intrusion detection systems have some fundamental limitations (for example, with respect to scalability and graceful degradation of service). In the last few years, an increasing number of distributed intrusion detection systems has been designed and built [e.g. 11, 66, 112, 137, 139].

2.3 Experiences in building a distributed intrusion detection system

AAFID [137] is a framework for distributed monitoring of hosts in a network specifically oriented towards intrusion detection. It uses a hierarchical structure of entities. At the lowest level in the hierarchy, AAFID agents perform monitoring functions on a host and report their findings to the higher levels of the hierarchy where data reduction is performed.

During the implementation of the AAFID system, we faced decisions regarding the type of monitoring to use, and we experienced the limitations of indirect monitoring and of external sensors. Even when trying to do direct monitoring, we encountered problems with the specific techniques used to perform it. These experiences prompted us to investigate new data collection techniques for intrusion detection.

AAFID was designed to use host-based data collection; therefore the agents run in each host and collect data from it. Audit trails are the most abundant source of data in a

Table 2.5
Comparison between centralized and distributed intrusion detection systems with respect to some of the desirable characteristics described in Section 1.2.2.

Characteristic	**Centralized**	**Distributed**
Reliability	A relatively small number of components need to be kept running.	A larger number of components need to be kept running.
Fault tolerance	The state of the intrusion detection system is centrally stored, making it easier to recover it after a crash, but also making it easier to get corrupted by a failure.	The state of the intrusion detection system is distributed, making it harder to store in a consistent and recoverable manner, but improving the chances that most parts will survive after a failure.
Added overhead	Impose little or no overhead on the hosts except for the ones where the analysis components run. In those hosts a large load is imposed and they may need to be dedicated to the analysis task.	Impose little overhead on the hosts because the components running on them are smaller. However, the extra load is imposed on most of the hosts being monitored.
Scalability	The size of the intrusion detection system is limited by its fixed number of components. As the number of monitored hosts grows, the analysis components will need more computing and storage resources to keep up with the load.	A distributed intrusion detection system can scale to a larger number of hosts by adding components as needed. Scalability may be limited by the need to communicate between the components and by the existence of central coordination components.
Graceful degradation of service	If one of the analysis components stops working, most likely the whole intrusion detection system stops working. Each component is a single point of failure.	If one analysis component stops working, some programs or hosts may stop being monitored, but the rest of the intrusion detection system can continue working.
Dynamic reconfiguration	A small number of components analyze all the data. Reconfiguring them may require the intrusion detection system to be restarted.	Individual components may be reconfigured or added without affecting the rest of the intrusion detection system.

Unix system and are the data source used by most intrusion detection systems. In the first implementation of the AAFID system, most of the agents obtained their data from log files. However, audit trails are an indirect data source and suffer from the drawbacks mentioned in Section 2.1.2.

To perform direct data collection appropriately, operating system support is needed, possibly in the form of hooks to allow insertion of checks at appropriate points in the system kernel and its services. Lacking this support, we implemented direct monitoring using the following mechanisms:

- Separate entities that run continuously, obtaining information and looking for intrusions and notable events.

  This is the form of most existing AAFID agents. Some agents obtain information from the system by running commands (such as **ps** [146], **netstat** [144] or **df** [142]), others by looking at the state of the file system (for example, checking file permissions or contents) and others by capturing packets from a network interface (note that this is not necessarily the same as doing network-based monitoring because in most cases these agents will only capture packets destined to the local host, and not to other hosts). Some agents have to resort to indirect monitoring by looking at audit trails because in some cases an audit trail is the only place where information can be obtained by an external sensor.

- Wrapper programs that interact with existing applications or utilities and try to observe their behavior by looking at their inputs and outputs.

- Wrapper libraries using library interposition [81].

  Using this technique, calls to library functions can be intercepted, monitored, modified or even cancelled by the interposing library. This is a powerful technique that can detect a wide range of attacks, but it is limited because it can only look at the data available as arguments to each call and at the global variables of a program. It cannot have access to any other internal data of the calling program or the called subroutine.

All these techniques of data collection can be classified as external sensors and have the weaknesses described in Section 2.1.4. For this reason, we started further exploration of the use of internal sensors which formed the basis for this dissertation.

2.4 Comments about intrusion detection architectures

In this chapter, we have described some of the main architectural concepts that are used in intrusion detection. In data collection, host-based direct monitoring using internal sensors presents multiple benefits in terms of efficiency, reliability and data collection abilities.

However direct monitoring using internal sensors has been used by few intrusion detection systems, as shown in Table 2.2. This is a consequence of the implementation difficulty of internal sensors and of the lack of studies regarding their properties and the related design and implementation issues. Also, internal sensors lack portability and increase the cost of deployment and maintenance of the intrusion detection systems because they require dealing with the source code of each program that needs to be monitored. However, the general idea of using internal sensors is to get as close as possible to the data sources needed by the intrusion detection system, and are the only mechanism able to provide other desirable characteristics, including fidelity, reliability and resistance to attacks. For these reasons, it is important to explore their capabilities and limitations.

In data analysis, a distributed architecture provides multiple benefits in terms of scalability, reliability and efficiency. This is the reason why through the years, intrusion detection system research and development has tended towards working on distributed systems [7].

Note that distributed intrusion detection systems are usually associated with operation on multiple hosts. However, according to the definitions given in this chapter, the components of a distributed intrusion detection system do not necessarily have to be in different hosts. If multiple parts of a single host are being monitored, and the data analysis components reside on different parts of the system, they could be considered as a single-host distributed intrusion detection system.

This chapter is organized around the distinction between the data collection and data analysis steps. Conceptually, this distinction is useful for analysis and for reasoning about the intrusion detection process. Its usefulness has been shown in efforts to model the intrusion detection process [7] and intrusion detection systems [111].

In practice, essentially every intrusion detection system has followed this separation by making data collection and analysis two distinct steps separated in time and often in space. However, this separation has the following shortcomings:

- It creates a window of time between the generation and the use of data. This can cause inconsistencies between what the intrusion detection system "sees" and the state of the system at the time the data is analyzed. It also increases the possibility that the data get modified before the intrusion detection system analyzes them, either by accident or malicious action. Furthermore, it reduces the timeliness of the reactions of the intrusion detection system: by the time it analyzes the data and reacts to an intrusion, it may be too late to do anything about it.

- It lengthens the path through which the data has to flow between its generation and its use. This increases the amount of traffic in the system (within the host or over the network), reducing the scalability of the intrusion detection system. It also increases the time between the generation and use of the data and brings along all the problems described in the previous item.

For these reasons, in practice, the data collection and analysis steps should be as close together as possible.

The rest of this dissertation describes a distributed architecture based on internal sensors that addresses many of the problems mentioned above and has significant beneficial properties with respect to the desirable characteristics described in Section 1.2.2.

2.5 Related work

The lack of adequate audit data for intrusion detection was documented by Price [114], showing that most intrusion detection systems in existence today operate with incomplete data, which are insufficient to support adequate detection. Internal sensors are able to overcome limitations in auditing systems by performing direct monitoring and completely skipping the operating system's auditing system.

The work by Crosbie and Spafford [33, 35] provided the foundation for using a large number of small independent components in intrusion detection. This work also provides an idea of how internal sensors could become more complex entities when necessary. They could even learn or evolve as they capture data about their environment.

The analysis of system call sequences to detect intrusions proposed by Forrest et al. [49] is a technique that lends itself naturally to be implemented using internal sensors. This was demonstrated in practice by the further development of the pH system based on that technique [135], which is implemented almost completely inside the Linux kernel. The pH system also responds to attacks by slowing down or aborting system calls, showing the potential that internal sensors have for providing not only detection but also response capabilities.

The collection of data using specialized mechanisms for detecting certain vulnerabilities in a Unix kernel was described by Daniels and Spafford [37]. That work focused on low-level IP vulnerabilities, and described the generation of new audit events (which could be classified as internal sensors) and the implementation of methods for detecting certain vulnerabilities.

Erlingsson and Schneider [47] described the use of *reference monitors* to monitor the execution of a program. The reference monitors they describe are implemented as code that evaluates a security automaton and is inserted automatically before any instruction that accesses memory. These reference monitors could be considered as internal sensors that check for generic violations of policy. The monitors also halt the program when a violation is detected, so it can be considered as a reactive intrusion detection system.

The concept of application-level intrusion detection has been described by Sielken [131], who discussed advantages of the approach from a theoretical standpoint. We agree with the advantages that application-specific monitoring can provide for intrusion detection. Internal sensors are an ideal tool for this purpose because they can be embedded into any program, whether it is a system program or a user-level application.

The idea of using library interposition for intrusion detection, as described by Kuperman and Spafford [81] was a first step in doing direct data collection for intrusion detection. We classify it as a form of external sensors, but we think it can be further developed to provide good application-specific intrusion detection by, for example, tailoring interposed libraries to specific applications, or combining data generated by interposed libraries with data provided by internal sensors to get a complete picture of what is happening in a program.

As shown in Table 2.2, few intrusion detection systems have been developed using internal sensors. CylantSecure [154] utilizes internal sensors but only in the form of counters whose values are used to build a profile of program behavior. The values reported by the sensors are analyzed and profiled by an external program. The LIDS [68] and Openwall [105] projects have developed kernel patches for Linux [10] that prevent certain operations defined as “dangerous.” These patches add checks that constitute internal sensors, but are specifically tuned for preventing those operations.

Another example of the use of internal sensors is FormatGuard [31]. This is a specialized tool for detecting and preventing format-string-based buffer overflows [98, 127]. By recompiling the affected programs, code is inserted for checking when a format string attack is attempted against any of the functions instrumented. These pieces of code constitute internal sensors that detect attacks in a distributed fashion (because even within a single host, the “data analysis” is done by the sensors at each monitored component). Format string attacks are difficult to detect, and FormatGuard is one clear example of one of the advantages of internal sensors over external sensors: They can access internal information of the monitored component and can even add or re-implement functionality or information as needed to aid in the detection.

3. AN ARCHITECTURE FOR INTRUSION DETECTION BASED ON INTERNAL SENSORS

Internal sensors have many characteristics that make them well-suited for performing host monitoring and intrusion detection, as described in Section 2.1.4. In this chapter, we describe an architecture for intrusion detection called ESP (from *Embedded Sensors Project*) with the following main characteristics:

- Internal sensors are the main data collection component.
- It provides for distributed, localized data reduction through the use of embedded detectors.
- It also contemplates the existence of external sensors when necessary for data processing and higher-level operations.

The ESP architecture could be classified in the [Distributed, Distributed] cell of Table 2.1 and in the [Direct/Host-based internal, Distributed] cell of Table 2.2.

3.1 Embedded detectors

The ESP architecture uses embedded detectors as a mechanism for localized data reduction.

WORKING DEFINITION 3.1: EMBEDDED DETECTOR

An internal sensor that looks for specific attacks and reports their occurrence.

Embedded detectors should exist in the code at the point where an attack can be detected by using the data available at that moment. If implemented correctly, detectors are able to determine whether an attack is taking place by performing simple checks.

```
1   char buf[256];
2   strcpy(buf, getenv("HOME"));
```

Code before inserting detector

```
1   char buf[256];
2   {
3     if (strlen(getenv("HOME"))>255) {
4       log_alert("buffer overflow");
5     }
6   }
7   strcpy(buf, getenv("HOME"));
```

Code after inserting detector

Figure 3.1. Example of code vulnerable to a buffer overflow before and after inserting an embedded detector. On the right, lines 2–6 form the embedded detector.

Because of their detection abilities, embedded detectors are a mechanism for performing localized data reduction. This is particularly important in a distributed intrusion detection system for reducing the amount of data generated by the system.

3.1.1 How embedded detectors work

Figure 3.1 shows an example of a simple embedded detector. The code on the left is potentially vulnerable to a buffer-overflow attack [2] because the value of the HOME environment variable is being copied to a buffer without checking its length. On the right, lines 2–6 have been added and constitute an embedded detector. This detector computes the length of the HOME environment variable. If it is longer than the buffer into which it will be copied, the detector generates an alert. This example assumes that the function `log_alert` has been defined elsewhere. The string "buffer overflow" is shown only as an example—in a real detector, a more descriptive message should be provided.

This example gives an idea of how embedded detectors work in general: they look at the information available in the program to determine if an attack is taking place. If such a condition is found, an alert is generated.

The example in Figure 3.1 does not try to prevent the overflow from happening. It only reports its occurrence, as per our definition of embedded detector. Potentially, embedded detectors could try to stop the intrusions they detect. For example, our sample detector could cut the HOME environment variable to 255 characters to ensure that it will fit in the allocated buffer. However, the effects of detectors modifying data or altering the program flow are much harder to analyze. Our current work has focused on detection and not on reaction.

3.1.2 Relationship between internal sensors and embedded detectors

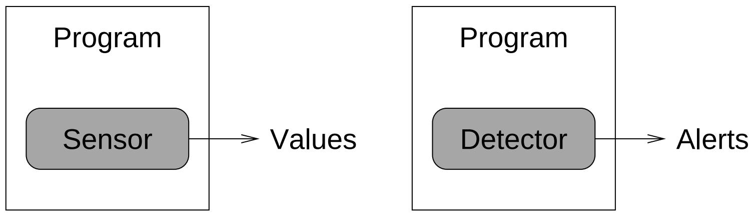
The difference between an internal sensor and an embedded detector is that sensors can observe any condition in a program and report its current state or value; whereas a detector looks for specific signs of attacks (Figure 3.2(a)). Embedded detectors are a specialized form of internal sensors.

Conceptually, an embedded detector can be considered as an internal sensor with added logic for detecting attacks, as shown in Figure 3.2(b). In some cases, the internal sensor is clearly differentiable in the code. For example, a detector for port scans [52] bases its decision on a sensor that keeps track of connections to ports and reports their number and sources.

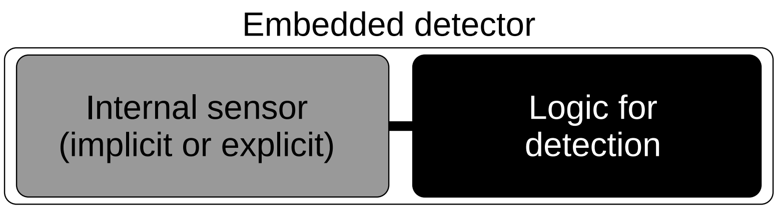
In other cases, the internal sensor is implicitly built into the embedded detector and its value is immediately used to take a decision. For example, a detector for a Ping-of-death [16] attack can check the size of a ping packet by comparing a variable against a certain threshold and emitting an alert if it is larger. In this case, the conceptual “sensor” would be the act of reading the value of the variable, and the “detector” portion would be the comparison of the value against a threshold.

This difference between the data collection and data analysis portions of an embedded detector can be significant in practice. In some cases, data from a single internal sensor—for example, the accumulated non-requested packets that have been received from a host—can be used by multiple detectors to look for different attacks (Figure 3.2(c)). It is also possible that a single detector collects data from multiple sensors (Figure 3.2(d)).

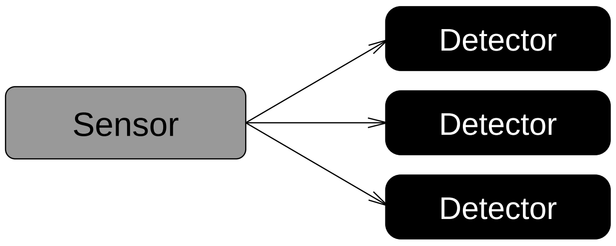
As mentioned in Section 2.4, the data collection and data processing phases of the intrusion detection process should be as close together in time and space as possible. For this reason, most embedded detectors should be of the type in which the sensor and the detector are tightly coupled together.



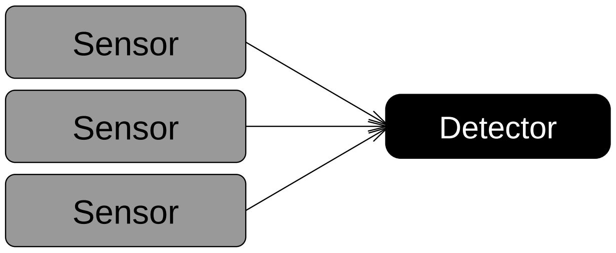
(a) Sensors generate values; detectors generate alerts.



(b) Conceptual structure of an embedded detector.



(c) One sensor can provide data to multiple detectors.



(d) Multiple sensors can provide data to a single detector.

Figure 3.2. Relationships and differences between internal sensors and embedded detectors.

3.1.3 Stateless and stateful detectors

One of the distinguishing characteristics of internal sensors (and of embedded detectors by extension) is that they can be placed at any point in the monitored component. Ideally, they should be placed at the point at which the information needed to detect an attack is readily available.

However, there are some cases in which a detector may need to collect information over a period of time to detect an attack. One example is the detection of port scans. A port scan cannot be signaled at the first packet received from a host because other packets could be on their way, and they should be observed to make a proper determination about the type and scope of the port scan that is taking place. So the detector (or its associated sensor) needs to accumulate information about packets that have been received from other hosts. When enough evidence is accumulated, an alert should be produced.

Considering this possibility, embedded detectors are classified in two groups:

WORKING DEFINITION 3.2: STATELESS EMBEDDED DETECTOR

> A detector that bases its decisions solely on information present in the program at the time of evaluation, or that can be obtained from the system at the moment it is needed.

WORKING DEFINITION 3.3: STATEFUL EMBEDDED DETECTOR

> A detector that adds information to the program for the purpose of detection. It may decouple data gathering and evaluation into two separate tasks.

This classification has an impact in the way sensors are designed and implemented. Stateless sensors are usually short because they check for an existing condition. Stateful detectors almost always add additional state-keeping code, and the state kept is used later for the detection.

In many cases, a stateful detector has a clearly differentiable internal sensor associated with it as discussed in Section 3.1.2.

```
1   char buf[256];
2   strncpy(buf, getenv("HOME"),
3           sizeof(buf));
```

Code before inserting detector

```
1   char buf[256];
2   {
3     if (strlen(getenv("HOME"))>255) {
4       log_alert("buffer overflow");
5     }
6   }
7   strncpy(buf, getenv("HOME"),
8           sizeof(buf));
```

Code after inserting detector

Figure 3.3. Example of code not vulnerable to a buffer overflow before and after inserting an embedded detector. On the right, lines 2–6 form the embedded detector.

3.1.4 Strengths and weaknesses of embedded detectors

Using embedded detectors in an intrusion detection system has the advantages and disadvantages mentioned for internal sensors in Table 2.4.

Additionally, embedded detectors can look for attempts to exploit a vulnerability independently of whether the vulnerability actually exists in the host where the detector is running. For this reason, embedded detectors can detect attacks against vulnerabilities that have already been fixed, or even that are specific to a different platform or operating system. For example, a detector in a Unix system could detect attacks specific to Windows NT. In this manner, embedded detectors can be used to implement a “universal honeypot” (a honeypot is the name given to a host that is connected to a network with the purpose of allowing attackers to explore it, usually with the objective of studying the attacker in action). This feature is used in this dissertation for exploring the detection of intrusions over multiple architectures and platforms.

Figure 3.3 shows code similar to the one in Figure 3.1, but this code is not vulnerable to a buffer overflow because the `strncpy` function is being used. However, the same detector can be added to this code as shown on the right side of Figure 3.3. This example shows how embedded detectors can exist even in code that is not vulnerable to the attacks for which the detectors look.

Another advantage of embedded detectors is that they can use the existing defense mechanisms of the monitored component (for example, if the program already looks for malicious activity) and combine them with detection.

3.2 The ESP architecture

The ESP architecture consists of three classes of components:

- Internal sensors and embedded detectors.
- Per-host external sensors
- Network-wide external sensors

3.2.1 Internal sensors and embedded detectors

These are the lowest-level components of the architecture. Internal sensors are used for direct monitoring of a host, and embedded detectors are used for performing localized data analysis in which certain types of intrusions are detected.

This layer of the architecture is its main distinguishing characteristic, and its study is the main subject of this dissertation.

3.2.2 Per-host external sensors

Although embedded detectors are able to detect a significant number of attacks, there are some attacks that may require a higher-level analysis by observing data generated by multiple internal sensors and embedded detectors, or even possibly data observed using other types of sensors.

For this reason, the ESP architecture allows for a layer of components running on each host that perform these higher-level operations. The number, structure and function of these components are left unspecified. However, it is feasible to imagine using ideas and components from other existing intrusion detection architectures [e.g. 97, 137] for this function.

3.2.3 Network-wide external sensors

To operate in a network environment, an intrusion detection system needs to be able to correlate information from multiple hosts. For this reason, the ESP architecture also allows for a layer of components that monitor the operations on multiple hosts and receive data from per-host external sensors or possibly even from network-based sensors.

The organization and specific functions of these components are outside the scope of this dissertation which focuses on the internal sensors layer. The use of network-wide monitoring components has been studied in other intrusion detection architectures [e.g. 97, 133, 137], and many of those concepts could potentially be applied to an ESP-based intrusion detection system.

3.3 Distinguishing characteristics of the ESP architecture

An implementation of the ESP architecture is described in Chapter 4 and is used to test the validity of the Thesis Hypotheses described in Section 1.4. In this section, the distinguishing features of the ESP architecture are discussed. They are related as appropriate with the desired characteristics of an intrusion detection system (Section 1.2.2) and with the drawbacks of the ESP architecture.

3.3.1 Types of data observed

ESP is fundamentally different from other intrusion detection systems in that it does not observe network packets or audit trails. By being part of the programs that are monitored, embedded detectors can obtain all the information that could be obtained from those sources plus more information that is not available from them. The data on which an ESP intrusion detection system bases its decisions is a combination of the following elements:

- The execution flow of the program being monitored as reflected by the location of the detector.
- The data being used by the program as stored in the variables and data structures available to the detector.
- Other system and program state that can be obtained by the detector.

By performing direct monitoring, ESP has all the advantages described in Section 2.1.2. When compared to other intrusion detection techniques that observe program behavior [e.g.

65, 66], ESP has the advantage of being able to observe the internal data and state of the program and not only its externally observable behavior.

These advantages come at the cost of losing generality in the implementation of the intrusion detection components. Detectors are built specifically for looking at certain data in certain locations, and the only way of porting them to some other location is to rewrite the code. Without a formal analysis, it is difficult to guarantee completeness of the data that is analyzed by the detectors in a practical implementation. However, as will be shown in Section 5.2, it is possible to build embedded detectors able to look for generic signs of malicious activity, regaining some generality.

Compared to other generic intrusion detection systems that use internal sensors [e.g. 36], ESP has the advantage that each embedded detector is optimized for the tasks it performs. The data it produces does not need (in the general case) to be post-processed to detect intrusive behavior.

Although ESP detectors look for specific actions that indicate intrusive activity, they do not necessarily have to be tied to a specific attack. As will be shown in Chapter 4, detectors can be built both for detecting specific attacks and for detecting generic intrusive activity; therefore being able to detect both known and new attacks.

Finally, because all the data is being observed from within the program that uses it, embedded detectors are able to examine information that would normally be unavailable. One example would be data that is only decrypted in memory while the program is running. This improves the completeness of the data that is available to the intrusion detection system.

These characteristics relate to desirable characteristic #8 and potentially to # 7 (see Section 1.2.2).

3.3.2 Tighter coupling between event collection and event analysis

As mentioned in Section 2.4, data collection and data analysis have traditionally been two clearly distinguishable, loosely coupled steps of the intrusion detection process.

The use of embedded detectors reduces this distinction because in most cases the data used for detecting attacks is not composed of discrete events that are collected and later analyzed, but of the factors described in Section 3.3.1. Therefore, ESP has the ability of not only performing matching operations on the data it receives but is able to actually examine the actions involved in the execution of the program.

The tighter coupling between data collection and analysis allows the detector to make a better determination about the occurrence of an attack and reduces the length of the path that the data has to traverse between its generation and its use; therefore it reduces the possibility that the data could be modified, destroyed or otherwise disrupted before they are used by the analysis components of the intrusion detection system. This aids in obtaining desirable characteristics #3 and #8.

Note that in some cases, it might be impossible to detect an attack using information available to a single embedded detector. In those cases, it is advisable to use internal sensors to generate data, and perform the analysis using components that have access to data provided by multiple internal and external sensors. Also, preservation of data may also be necessary for forensic purposes or for analyzing long-term events. Embedded detectors are not intended to replace other forms of data collection and analysis, but to provide a mechanism for performing localized data reduction when appropriate.

3.3.3 Intrusion detection at the application and operating system level

Application-based intrusion detection systems [131] can detect high-level attacks and are a good complement to network-based and operating-system–based intrusion detection systems.

The ESP architecture can be used to perform intrusion detection at the application, operating-system and network levels. In general, embedded detectors can be implemented at any point in the system depending on where the information that identifies malicious activity is available.

This flexibility helps in obtaining desirable characteristics #1 and #8.

3.3.4 Size of the intrusion detection system

Embedded detectors can be written to look specifically for the pieces of information that they need to perform the detection without having to go through a generic process of event collection and analysis. This makes it possible for the lowest-level components in the

ESP architecture to be highly optimized to their task and in most cases to be simple and short.

The small size of the sensors and detectors provides the ESP architecture with desirable characteristic #4.

3.3.5 Timeliness of detection

Embedded detectors can be located at the point where an intrusion would have an adverse effect, or at the point at which the malicious behavior can first be detected. This allows the ESP architecture to detect problems before they happen (or while they are happening) and creates the possibility of taking preemptive report, control and response actions. Although not discussed in this dissertation, it is conceivable that the intrusion detection system could also stop the intrusions before they cause any damage. This could be done by modifying the data that the program is using, by altering its state, or in extreme cases, by stopping or killing the program itself.

The timeliness of detection relates to desirable characteristics #1 and #8c.

3.3.6 Impact on the host

Embedded detectors in the ESP architecture are intended to perform simple checks to determine whether an attack is taking place. For this reason, they can have low impact on the host they are monitoring. For the same reason, it is possible to have a larger number of detectors in a host, increasing detection capabilities without imposing a large overhead.

However, note that because sensors and detectors can exist anywhere in the monitored components (even in critical sections of the code), a defective or poorly implemented detector has the possibility of significantly harming performance or reliability.

If properly implemented, the internal sensors and embedded detectors of the ESP architecture can obtain desirable characteristic #4.

3.3.7 Resistance to attack

At the lowest data collection and analysis level (that of the internal sensors and the embedded detectors), the ESP architecture is completely integrated into the monitored components, and there are no separate processes that belong to the intrusion detection system running on the host. For this reason, such an intrusion detection system is less vulnerable to tampering or disabling by an intruder. To disable the intrusion detection system, an attacker would have to disable the monitored component. Although this potentially constitutes an attack unto itself, it makes it impossible for the intruder to tamper with the monitored component to make it act in an unauthorized way (for example, by increasing its privileges) without being detected.

This high level of integration with the monitored components helps the ESP architecture to obtain desirable characteristic #3.

Because the monitored components have to be modified, the cost of implementation for an intrusion detection system that uses the ESP architecture may be higher than that for one which uses only external sensors. If the intrusion detection system is implemented on an existing system, the source code must be available, and the implementer needs to study and understand the source code before making any modifications. Ideally, ESP sensors and detectors should be incorporated into a program during its development.

4. THE ESP IMPLEMENTATION

This chapter describes the details of the implementation of a prototype intrusion detection system based on the ESP architecture. This prototype uses embedded detectors and constitutes the main testing and analysis platform for this dissertation.

4.1 Purpose of the implementation

The two hypotheses that underlie this dissertation (Section 1.4) are practical in nature. First, they intend to show that it is feasible to build an intrusion detection system using the ESP architecture. Second, it can be used to detect both known and new attacks. Therefore, an implementation was a center point for the development of this dissertation and was used both for practical verification of the intended features of the architecture and for aiding in reasoning about and experimenting with its characteristics.

The ESP implementation was also used to confirm the possibility of building both specific and generic detectors.

In the rest of this chapter, the term "detector" is used to mean both internal sensors and embedded detectors except when explicitly stated otherwise.

4.2 Specific and generic detectors

Related to the ESP implementation, the concepts of specific and generic embedded detectors are introduced.

WORKING DEFINITION 4.1: SPECIFIC DETECTOR

An embedded detector designed to detect one specific attack.

WORKING DEFINITION 4.2: GENERIC DETECTOR

An embedded detector designed to look for signs of intrusive activity that can be used to detect a group of attacks with certain common characteristics.

For example, a detector implemented in the **eject** program that looks for long command-line arguments in an attempt to exploit buffer overflows in that program would be considered a specific detector. A detector implemented in the Unix kernel that looks for long command-line arguments passed to any program is considered a generic detector, and it would detect not only the attacks against **eject**, but also against other programs.

The efficacy of generic detectors is one of the main premises of this dissertation because they enable ESP to detect previously unknown attacks. The overall methodology was to start by implementing different specific detectors. The expectation was that through this implementation, some patterns would start to emerge, and those patterns would lead to the creation of generic detectors.

4.3 Sources of information

We used the CVE (Common Vulnerabilities and Exposures) database [21, 89] as a source of attacks for the implementation of specific detectors. The CVE is a database that has been widely adopted by the intrusion detection community as a naming convention for vulnerabilities and attacks against computer systems. It does not provide a classification or taxonomy, but a unique identifier for each entry, pointers to sources of information, and a best-effort guarantee that no duplicate entries exist in the database. Furthermore, it includes entries corresponding to multiple computer architectures, operating systems and types of vulnerabilities. Because of these features, it can be used as a fairly complete, diverse and recognized list of known vulnerabilities and attacks.

The specific detectors implemented map directly to entries in the CVE. Linking each detector to a CVE entry facilitates discussion and reference, and ensures that no repeated detectors are implemented. For the ESP prototype implementation, version 20000712 of the database was used. This version of the CVE was published on July 7 of 2000 and contained 815 records.

Detailed information about each CVE entry, including exploits, was gathered from common sources on the Internet [e.g. 13, 106, 122, 130, 155].

As a special case, we implemented detectors for different variants of port scanning [52]. Port scans are not considered attacks by themselves but are commonly a prelude to an attack; therefore they are useful to detect. Port scans do not have CVE numbers.

4.4 Implementation platform

The detectors in our prototype have been implemented in OpenBSD [103]. This version of the Unix operating system was chosen for the following reasons:

- The source code is available, which makes it easy to instrument the detectors both in the kernel and in system programs. Extensive documentation is available [87, 140] about the internals of the kernel.

- The OpenBSD source code is managed and distributed as a single directory tree. This makes it more manageable than Linux, for example, where the source code for different components of the operating system is distributed as separate packages. The OpenBSD source tree closely mimics the layout of the system itself, making it easy to locate the code for different programs and subsystems.

- The OpenBSD project is known for its attention to security and has gone through an extensive code security audit process. Most of the security problems for which detectors were implemented had already been fixed in OpenBSD. Looking at the security patches and at the change log for each file made it easier to locate the portions of code where the problems existed, and helped in determining where the detectors for each attack had to be placed. In some cases the code that fixed the problem could be identified, helping in the determination of where to put the detector code for producing a notification. Additionally, because the problems themselves no longer existed, it was easier to try attacks against the instrumented system without worrying about the adverse effects they could have on the host.

As described in Section 3.1.4, although OpenBSD was used as the implementation platform, we were able to build detectors for attacks that are specific to other platforms, or for exploitations of vulnerabilities that have already been fixed in OpenBSD. Furthermore, because most of the detectors were implemented with simple modifications or additions to existing code, they should be relatively easy to port to other systems without extensive redesign, particularly for other Unix-like systems.

For the particular implementation described in this document, the platform used was OpenBSD 2.7 running on a computer with an Intel processor.

4.5 Reporting mechanism

All the detectors need a mechanism for generating reports when they detect an attack. The following characteristics were determined to be desirable for the reporting mechanism:

Exclusivity: The reporting mechanism used by the embedded detectors should not be used by any other system in the host. This ensures that detector reports can be obtained from a single source without having to filter extraneous messages.

Efficiency: Because large numbers of embedded detectors will exist in a host, the reporting mechanism needs to use a minimum of resources in terms of memory and CPU. Also, reports need to be available as soon as possible after a detector generates them.

Note that because embedded detectors only generate reports when they detect an attack, the generation of reports should be a relatively rare event on a normal host.

Security: It should be difficult for an attacker to disrupt the reporting mechanism, either by inserting invalid messages, or by intercepting or modifying the messages that detectors generate.

We considered the intra-host communication mechanisms described by Balasubramaniyan et al. [8], but decided against them primarily because of the overhead they require and because they are geared towards exchanging messages between separate processes. For our purposes, we needed a mechanism for all the different detectors to produce messages that could then be accessed by a higher-level mechanism and that satisfied the requirements given above.

We decided to implement the reporting mechanism for embedded detectors as a new system call in OpenBSD and to base it partially on the kernel-messaging mechanism that already existed in the operating system. It is implemented by a circular buffer in kernel memory. Messages are written to the buffer using a new system call called `esp_log`, and read through a new device called `/dev/esplog`.

This mechanism satisfies the requirements we set almost completely. It is exclusive to the detectors because it is completely separate from all other logging mechanisms in the host. Also, it is efficient for generating messages from detectors within the kernel because the call happens within the kernel context, and the only operation performed is copying the message to the buffer. When called from user-level processes, a context switch occurs.

The messages are stored inside kernel memory, so they cannot be modified by an attacker unless it has root privileges, and even then, it is a complex task to locate the buffer within the kernel memory and overwrite the messages. Furthermore, messages disappear from the buffer when they are read, so if an intrusion detection system is constantly reading the messages, they exist in kernel memory for only a short period of time.

With respect to access control, the `/dev/esplog` device provides exclusive access, so that only one process can read it at a time. Therefore, if an intrusion detection system opens the device and never closes it, no other processes can access the messages generated by the detectors. Moreover, messages are never stored on a disk file or any other external storage medium from the moment they are generated until they are read by an external process.

This mechanism also has some drawbacks. User processes need to make a system call (causing a context switch) when they need to generate a detector message, which may have a negative impact on performance. Additionally, there is no fine-grained access control in the current implementation of the reporting mechanism both for reading and for generating messages. This results in two problems. First, if an attacker manages to open the `/dev/esplog` device before the intrusion detection system, he will be able to read messages generated by the detectors. Second, any program can generate messages, so it is possible for an attacker to generate bogus messages to interfere with authentic detector messages.

Note that these drawbacks are limitations of our current implementation of the detector reporting mechanism and not of the ESP architecture itself. Mechanisms such as rate-limiting on messages and capability-based access control [109] could be employed to address these problems in future implementations.

Access to the `esp_log` system call and some other utility functions is provided through a library we implemented for this purpose, called `libesp`. A full description of the functions in the `libesp` library is provided in Appendix B.

4.6 Methodology for implementation of detectors

We followed a consistent methodology for the implementation of all the specific detectors:

1. Select a detector to implement. In the case of specific detectors, this corresponded to selecting an entry from the CVE database. Most entries were selected at random from the CVE to ensure coverage of different types of attacks.

2. Determine the applicability of the detector (see Section 4.7) to the implementation platform. If the entry is determined to be non-applicable, return to step 1.

3. Obtain information about the entry, including advisories, exploit scripts, patches and workarounds, etc. The first step was to check the references provided with the CVE entry, followed by consulting other sources of information as described in Section 4.3.

4. Determine if the attack corresponding to this entry would be detected by an existing detector. In this case, mark it as "detected by" the existing detector and return to step 1.

5. Examine the source code of the affected program, and determine where the vulnerability occurs. This was usually the most time-consuming step because it involved studying and understanding the source code of the program.

6. Implement the detector. Once the vulnerability was understood the code for the detector was added and the program was recompiled and tested.

In some cases, the new detector can be implemented by extending the functionality of another existing detector (for example, by adding code to check for a different but similar case). In this case, the new detector is marked as "implemented by" the existing detector.

Generic detectors were constructed as they became apparent during the implementation of the specific detectors. For example, after a few specific detectors were built for checking the length of command-line arguments in different programs, a generic detector for checking the length of command-line arguments in the whole system became apparent and was implemented.

4.7 Applicability of CVE entries

The CVE contains records for vulnerabilities and attacks of a wide variety of types, including coding errors, race conditions, configuration errors, and unsecure features of software. Also, it includes entries affecting a wide variety of operating systems and platforms, including multiple versions of Windows and Unix, as well as platform-specific vulnerabilities for routers, switches and other devices.

Therefore, it is clear that not all CVE entries are applicable for implementation of detectors in the chosen implementation platform. From the CVE entries selected at random, we accepted as implementable those that satisfied any of the following conditions:

- The CVE entry corresponds to an attack that can be launched against an OpenBSD system with reasonable ease. This includes, for example, attacks that are launched using any standard Unix command. It also includes many web-based attacks because those can be launched against any web server, independent of the platform in which it runs.

- The CVE entry corresponds to a program that exists in the OpenBSD ports collection [104], or that can be compiled and installed on OpenBSD without extensive porting effort.

- The CVE entry corresponds to a vulnerability that is clearly described and whose operation could in theory be observed and understood on an OpenBSD system even when the specific affected programs do not exist in OpenBSD.

These criteria allow for the selection of CVE entries corresponding to a wide variety of operating systems and platforms.

4.8 Design and implementation considerations for detectors

We developed a few guidelines for the design and implementation of embedded detectors. These guidelines help to improve the maintainability and usefulness of the detectors.

Once an intrusion is detected, it would be relatively easy for the detector to react to it, possibly even modifying the behavior of the program under attack. However, for research purposes, the effects of detectors modifying the behavior of a program is harder to analyze, so we decided to use the detectors only as observers. For this reason, an early design decision was that detectors must not interfere with the program to which they are added. This means that they do not have to modify any data that the program uses, nor alter its flow in any way. We refer to this guideline as "the prime directive for detectors" [102].

To make them more understandable and easier to maintain, detectors must be as short and unfragmented as possible. This means that detectors should not perform any unnecessary actions. In most cases, because detectors only need to test for certain specific conditions, this is possible to achieve. There are some detectors that need to keep a certain amount of state to compare between different points in the program. In those cases detectors must be composed of more than one code fragment, but they should be easily identifiable.

We should be careful to notice cases in which the detector already exists in the program—for example, many modern operating systems include code to detect SYN Flood [126] attacks—to avoid adding unnecessary code to the system.

To facilitate testing and deployment, detectors must be configurable at compile time. This means that the inclusion of the detectors into the program must be a compile-time option. We usually achieved this in C programs using appropriate **#ifdef** directives, which are set from the program's `Makefile`. We decided to use different labels for each detector, so they can be enabled or disabled individually. Our convention was to use macros of the form ESP_*ID*, where *ID* is the identifier of the detector. For example, the code corresponding to the detector for entry CVE-1999-0103 is surrounded by:

```
#ifdef ESP_CVE_1999_0103
  code for the detector
#endif
```

The ESP_CVE_1999_0103 macro must be defined in the `Makefile` for the detector to be compiled.

Finally, to increase their effectiveness, detectors should look for exploitations of the general vulnerability that allows the intrusion to take place. However, during our development we have found that in some cases it is difficult to differentiate between normal behavior of a program and its behavior under attack. This is particularly true when the detector is being implemented in a version of the program in which the vulnerability has been fixed. In these cases, we have resorted to some heuristics to detect attacks, such as examining the data involved and comparing it with the data used by common attack scripts for the corresponding entry.

4.9 Naming, testing and measuring detectors

Each detector is given a unique identifier. For detectors inspired by CVE entries, this identifier is the corresponding CVE name. CVE names consist of the string “CVE”, the year in which the entry was added to the database and a four-digit number separated by dashes. For example, “CVE-1999-0016” and “CVE-2000-0279” are valid CVE names.

For other detectors (particularly generic detectors), the identifier consists of the string “ESP” followed by a descriptive name, all in capital letters, with the words separated by dashes. For example, “ESP-PORTSCAN” and “ESP-TMP-SYMLINK” are valid identifiers.

During the initial implementation process, each detector implemented was tested by launching the corresponding attack against the host in which the detector was implemented and verifying that it detected the attack correctly. The trail of messages generated by the detectors was monitored continuously, and when false positives occurred, the corresponding detectors were tuned to prevent them, whenever possible.

An important aspect of the ESP detectors is their small size, so we were interested in measuring them. Initially, we considered lines of code as a measure of detector size, but discarded it because of its subjectivity. Instead, the unit we used for measuring detector size was the "number of executable statements added to or modified in" (ESAM) a program to implement the sensor or detector. The definition of "executable statement" was used as provided in the Source Code Counting Rules described by Jones [72] and as implemented by Metre [85].

For example, the detector shown in the right side of Figure 3.1 has an ESAM count of 2 because the **if** statement and the call to **log_alert()** each count as 1 executable statement.

As a measure of the "fragmentation" of each detector's implementation, we used the number of Blocks of Code Added or Modified (BOCAM). For example, the detector shown in Figure 3.1 has a BOCAM count of 1, because all its code is in a single contiguous block.

4.10 Relationships between detectors

There are two main relationships that can exist between detectors. These relationships were extracted from observations made during the implementation of the ESP prototype.

The "detected by" relationship exists between detectors A and B (in the form "A is detected by B") when the attack corresponding to detector A is also detected by B. This relationship exists mainly when B is a generic detector.

When detector A is detected by B, we also say that detector B "covers" detector A.

WORKING DEFINITION 4.3: COVERAGE OF A DETECTOR

The coverage of a detector is the number of other detectors it covers.

During the implementation, we made the following observations about the "detected by" relationship:

1. It is transitive. If A is detected by B, and B is detected by C, then A is also detected by C.

2. Not all detectors are covered by some other.

3. A detector can be covered by more than one other detector.

4. Multiple detectors may be covered by a single one.

5. If detector A is covered by B, A does not have code of its own in the implementation. The exception to this rule is when A was implemented first, in which case it will have code of its own in addition to being detected by B.

The second main relationship we observed between detectors is “implemented by.” It exists between detectors A and B (in the form “A is implemented by B”) when detector A is implemented by adding code to a previously existing detector B. This relationship usually occurred between two detectors that correspond to closely-related intrusions, so that existing code could be extended to detect a new attack.

During the implementation of the ESP prototype, we made the following observations about the “implemented by” relationship:

1. Not all detectors are implemented by some other.

2. Multiple detectors may be implemented by a single one.

3. A detector was never implemented by more than one other detector.

4. If detector A is implemented by detector B, there will be some code in detector B corresponding to A. This code was normally counted as belonging to detector A. For example, if detector A was implemented by adding two statements to the existing code of detector B, the ESAM count of A is 2, and assuming those statements are contiguous, its BOCAM count is 1 (A’s code block is counted additionally to the BOCAM count of B).

4.11 Recording information about sensors and detectors

During the implementation of the ESP detectors, we found the need to document in detail the process by which the detectors were implemented and their characteristics. Initially, free-form text files were used, but it soon became apparent that a more structured format was necessary for later analysis. Furthermore, some characteristics of the detectors are measurable or can be described using discrete values—for example, whether a detector is stateless or stateful.

For this purpose, we developed an XML [12, 58] representation of information about ESP detectors. This representation includes measurable and discrete information such as the following:

- Identifier of the detector.
- Size of the detector in different units (ESAM and BOCAM).
- Type of the detector (Stateless or Stateful).
- Requirements of a detector (other detectors or programs that need to be present for the detector to operate).
- “Detected-by” and “Implemented-by” relationships (see Section 4.10).
- Classification of data sources.
- Description of the format of the messages produced by the detector.
- Source directory in which the detector is implemented.
- Classification of the vulnerability that corresponds to the detector (when applicable).
- Operating system to which the intrusion is applicable.

It also includes free-form information about the detector, such as the following:

- Description of the detector.
- Cause of the associated vulnerability (when applicable).
- Textual descriptions of data sources and how the data is observed.
- A log of activity in the implementation of the detector.
- Miscellaneous notes and comments.
- Listing and description of files related to the implementation of the detector.

The format and contents of this XML representation was defined using a DTD (Document Type Definition) defined for this purpose.

Figure 4.1 shows an example of the XML representation of a detector.

4.12 Case studies

As an initial proof of feasibility, two groups of detectors were selected for implementation: those for attacks against the Sendmail program, and those for network-based attacks. We describe these two groups in detail as a representative sample of the issues encountered during the implementation process. Later sections present overall results and further comments about all the detectors implemented.

4.12.1 Embedded detectors for network-based attacks

We implemented a number of embedded detectors for common network-based attacks. We use the term *network-based attacks* to encompass those that exploit both low-level IP vulnerabilities and network-based vulnerabilities as described by Daniels and Spafford [37]. In this section, we describe this implementation and the results obtained.

Detectors implemented

We chose network-based attacks because several interesting attacks of this type have appeared over the last few years. Also, they are the type of attacks that intrusion detection systems using network-based data collection usually detect, and our implementation shows how effective embedded detectors can be for these attacks.

Table 4.1 lists the detectors that were implemented for network-based attacks during the initial study phase.

In the next sections, we describe some representative attacks. We show the code of the corresponding detectors (in many cases the code has been reformatted for space) and explain where they have been placed within the operating system. We will see that detectors are short and simple, yet provide advanced detection capabilities.

The lines of code added or modified by a detector have been highlighted in each code section. The detectors have been wrapped in `#ifdef` directives and in an `if` clause, so they can be disabled both at compile time and at run time. We explored the possibility of

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE ESP-Component SYSTEM "ESP-Component.dtd">

<ESP-Component type="sensor_detector">
    <ID type="CVE">CVE-1999-0164</ID>
    <Description>
        A race condition in the Solaris ps command allows an attacker to
        overwrite critical files.</Description>
    <Detector-Info>
        <detector-type>Stateless</detector-type>
        <detected-by>ESP-SYMLINK-CHOWN</detected-by>
        <detected-by>ESP-SYMLINK-CHMOD</detected-by>
        <detected-by>ESP-TMP-SYMLINK</detected-by></Detector-Info>
    <Cross-Ref>ESP-SYMLINK-CHOWN</Cross-Ref>
    <Notes>

        <item> This was a well-known problem in old versions of
            Solaris that allowed changing the ownership of arbitrary
            files to root. The problem was a predictable filename in
            /tmp (coupled with bad permissions in /tmp that allowed
            any user to remove other users’ files) followed by a
            chown() of that filename to root. By removing that file
            and creating it as a symlink to another file after the
            creation but before the chown(), it was possible to change
            any file to root.</item></Notes>

    <Files>
        <file>
            <file-name>exploit</file-name>
            <file-description>
                Exploit program</file-description></file></Files>
    <Operating-System>
        <OS-name>OpenBSD</OS-name>
        <OS-version>2.7</OS-version></Operating-System>
    <Operating-System type="vulnerable">
        <OS-name>Solaris</OS-name>
        <OS-version>2.3</OS-version>
        <OS-version>2.4</OS-version>
        <program>ps</program></Operating-System>
    <Source-Directory type="vulnerable">/solaris/usr/bin/ps
        </Source-Directory>
    <Classification type="Krsul">2-7-1-4</Classification>
</ESP-Component>
```

Figure 4.1. Example of the XML representation of detector information.

Table 4.1
Summary of network-related detectors that were implemented during the initial study phase. All but CVE-1999-0103 exist in the kernel code. The Type column indicates whether the detector is stateful or stateless as defined in Section 3.1.3. The ESAM and BOCAM columns indicate the sizes as defined in Section 4.9.

ID	**Description**	**Type**	**ESAM**	**BOCAM**
CVE-1999-0016	Land	Stateless	2	1
CVE-1999-0052	Teardrop	Stateless	2	1
CVE-1999-0053	TCP RST DoS	Stateless	2	1
CVE-1999-0077	TCP sequence number prediction	Stateless	2	1
CVE-1999-0103	Echo-chargen connections	Stateless	8	4
CVE-1999-0116	TCP SYN flood	Stateless	2	1
CVE-1999-0128	Ping of death	Stateless	3	1
CVE-1999-0153	Win nuke	Stateless	3	1
CVE-1999-0157	Pix DoS	Stateless	2	1
CVE-1999-0214	ICMP unreachable messages	Stateless	2	1
CVE-1999-0265	ICMP redirect messages	Stateless	8	1
CVE-1999-0396	NetBSD TCP race condition	Stateful	3	2
CVE-1999-0414	Linux blind spoofing	Stateless	3	1
CVE-1999-0513	Smurf	Stateful	22	5
CVE-1999-0514	Fraggle	Stateful	12	5
ESP-PORTSCAN	Port scanning	Stateful	151	9

integrating the run-time control variables to the kernel parameters mechanism available in OpenBSD through which some kernel parameters can be modified at run time. The ability to disable the detectors at runtime may not be desirable in a production system because it offers the possibility for an attacker to disable the detectors if he manages to obtain sufficient privileges in the system. However, for the purposes of testing, the capability of enabling and disabling detectors at runtime was considered appropriate.

Stateless Detectors

Twelve of the 16 detectors in Table 4.1 are stateless. Those detectors test if an attack condition is met and call the alert mechanism. They use information from the network stack and are placed within its execution path. An example of this type of attack is the Land [18] attack (CVE-1999-0016). It consists of a TCP SYN packet sent to an open port with the source address and port set to destination address and port. OpenBSD filters those packets when processing SYN packets in the TCP_LISTEN state and drops them. The detector exploits this and is placed before the packet drop, so it is effectively only a single statement (with additional code for detector management).

```
case TCPS_LISTEN: {
 ...
  if (ti->ti_dst.s_addr == ti->ti_src.s_addr) {
/* ESP */
#ifdef ESP_CVE_1999_0016
  if (esp.sensors.land)
      esp_logf("CVE-1999-0016: LAND attack\n");
#endif
      goto drop;
  }
  ...
}
```

The CVE-1999-0103 (Echo-chargen denial-of-service attack [17]) detector was implemented within the **inetd** [143] program and not in the kernel. Also, it is longer than other detectors because it has to query additional information that is not readily available outside the kernel.

In this group of detectors we found the first instance of an “implemented-by” relationship. The PIX DoS attack [24] (CVE-1999-0157) exploits the same vulnerability (failure to handle a special case of overlapping IP packets) as Teardrop [18] (CVE-1999-0052) but with a variation to bypass a PIX firewall. In this case, the ESAM count indicates the number of statements added to or modified in CVE-1999-0052 to implement the detector for CVE-1999-0157, and the BOCAM count of 1 indicates that those statements are contiguous.

SYN flooding [126] is a denial-of-service attack based on exhaustion of the resources allocated in a host for half-open TCP connections. The detector for SYN flooding was implemented as stateless. OpenBSD does resource allocation for half-open connections and drops old connections after a threshold has been reached. The detector triggers when such a connection is dropped. This shows an advantage of embedded detectors: they can use the defense mechanisms of the operating system itself and combine them with detection.

Other attacks are ICMP unreachable messages (CVE-1999-0214) and ICMP redirects (CVE-1999-0265), both of which allow an attacker to cause a denial-of-service attack by faking ICMP control messages. The problem is that those faked ICMP messages may be indistinguishable from legitimate messages created by hosts at the end points of the connection or by interior routers. These type of attacks are inherent to the design of TCP/IP. OpenBSD tries to protect itself from malicious messages with extensive checks against its local state and we placed the detectors after those, i.e. that packets that are accepted by OpenBSD will not raise an alarm, while rejected will. Nevertheless cleverly forged packets still may exploit those vulnerabilities.

Stateful Detectors

Stateful detectors accumulate data about events that indicate attacks. In some of our detectors, a separate timer routine reads these data and triggers an alarm if a threshold has been met. Two typical examples are the Smurf and Fraggle [19, 69] attacks. They try to flood the host with packets of a certain type and make it unavailable to its users.

Those attacks rely on traffic amplification mechanisms. Traffic amplification is based on mechanisms that generate a response significantly larger than the request that originates it. This enables a single attacker to generate the amount of traffic necessary to exceed the victim’s capacity. Stateless detectors may detect the packets that use those mechanisms to generate the attack. However, often the attacked site and the amplifying site are different, so a different detector for the victim host is necessary. Identifying the vulnerability at the amplifying site can assist in tracing the attack.

The Smurf attack [19] sends ICMP ECHO_RESPONSE packets. Those do not differ from legitimate packets (for example, in response to a **ping** command [145]) except that there is no program expecting them. For implementing the detector, we assumed the semantics of the **ping** program, that stores its Process ID in the ICMP ID field to identify its replies. Based on that technique, we store the Process ID of all ICMP raw sockets in the socket data structure when they are created:

```
case PRU_ATTACH:
 ...
/* ESP */
#ifdef ESP_CVE_1999_0513
   if (esp.sensors.smurf && ((long) nam) == IPPROTO_ICMP)
      so->so_pgid = curproc->p_pid;
#endif
```

We check this information at arriving ICMP echo replies and increase a counter for unrequested echo replies if there is no matching socket (this is done in the `esp_smurf()` function, not shown).

```
case ICMP_ECHOREPLY:
/* ESP */
#ifdef ESP_CVE_1999_0513
  if (esp.sensors.smurf) {
    if (esp_smurf(ip, icp))
      goto freeit;
    goto raw;
  }
#endif
```

The technique used above shows another advantage of embedded detectors: additional information can be made available when necessary for the purposes of detection.

The alarm for Smurf is rate-limited. A legitimate use of **ping** will probably be interrupted when there are still echo reply packets in the network to be delivered to the host, and those packets should not raise an alarm although they do match the signature. A network layer timer that runs for three seconds examines the counter and raises an alarm only if it exceeds a threshold.

Port scanning [52] is a probing technique used to determine what ports are open on a host, and is commonly performed as an exploration phase by an attacker. For this reason, although port scans themselves are not attacks, we consider it desirable to detect them. We implemented a port scan detector that reacts to all known types of port scanning techniques (including stealth and slow scans) by using the state of the network stack. Also, it has more advanced monitoring and reporting capabilities because it reports multiple probes as one scan and identifies its type.

The NetBSD race attack detector (CVE-1999-0396) is a special case of the port scan detector and uses its reporting routine. For this reason, CVE-1999-0396 is “detected-by” ESP-PORTSCAN.

A detailed description of all the detectors for network-based attacks is available [75].

Testing the detectors

A test suite of exploit programs was assembled to test the detectors. The exploit programs were acquired preferably from the same sources that published the vulnerabilities when they made them available. If they were not available or not working, we wrote our own exploits according to the descriptions. The test suite was run supervised from a remote machine on the same local area network (LAN) and all attacks were detected reliably.

An independent tester ran the same set of attacks. The attacks were run over the campus network, with different network technologies and possibly even filtering in between. The results were that only a small number of attacks arrived at the target. This experience shows that most attacks are of rather low quality and are dependent on the network environment. The packet log shows that all received attacks were detected. The test was repeated from a machine on the same LAN and the results match those of the supervised test.

In the testing period the host reported some attacks not generated as a controlled experiment, notably port scans. To verify their correctness, they were compared to the packet log and all could be verified as real events.

4.12.2 Embedded detectors for sendmail attacks

Sendmail [30] is the most widely used mail-delivery agent on Unix machines. A number of security problems have been encountered in sendmail over the years, and many of them can still be found in systems connected to networks [23].

Sendmail is a complex user-level process with multiple clearly identifiable vulnerabilities in its past. For this reason, it was an ideal candidate for the implementation of detectors outside the kernel.

Detectors implemented

We implemented the detectors in version 8.10.1 of sendmail which is the version included with OpenBSD 2.7. During the initial test phase, 11 sendmail detectors were implemented, and they are summarized in Table 4.2.

In the next sections we will describe in more detail some of these detectors. As with the network detectors described in Section 4.12.1, the sendmail detectors are surrounded by **#ifdef** statements that allow to disable or enable them individually at compile time. No runtime mechanism exists for disabling or enabling these detectors.

Stateless detectors

Seven of the 11 sendmail detectors implemented in this phase were stateless. Most of the vulnerabilities to which these detectors correspond have been fixed in the newer versions of sendmail. In some cases the new code specifically looks for and avoids the corresponding attacks. In those cases, the detectors consisted of simple checks or only the calls to the reporting mechanism. This is the case for most of the detectors that consist of only one or two executable statements.

As an example, we present the detector for CVE-1999-0096 corresponding to the use of the “decode” alias to overwrite arbitrary files on a system. This alias is no longer enabled by default in new versions of sendmail, but because there are still old versions of sendmail in use on the Internet, it is important to detect attempts to use those aliases. In this case, the detector specifically looks for mail sent to the `decode` address or the equivalent `uudecode` address:

Table 4.2
Summary of sendmail-related detectors implemented during the initial study phase. All but CVE-1999-0057 exist in the sendmail program itself. The Type column indicates whether the detector is stateful or stateless as defined in Section 3.1.3. The ESAM and BOCAM columns indicate the sizes as defined in Section 4.9.

ID	Description	Type	ESAM	BOCAM
CVE-1999-0047	Buffer overflow vulnerability in sendmail 8.8.3/8.8.4	Stateful	10[1]	7[1]
CVE-1999-0057	Multiple vendor vacation(1) vulnerability	Stateless	2	1
CVE-1999-0095	Debug command in sendmail	Stateless	1	1
CVE-1999-0096	Sendmail decode aliases can be used to overwrite files	Stateless	6	2
CVE-1999-0129	Sendmail group permissions vulnerability	Stateless	2	1
CVE-1999-0130	Sendmail Daemon Mode vulnerability	Stateful	3	3
CVE-1999-0131	Sendmail GECOS buffer overflow and resource starvation	Stateless	1	1
CVE-1999-0204	Execution of root commands using malformed identd responses	Stateful	4	2
CVE-1999-0206	MIME buffer overflow in sendmail 8.8.0 and 8.8.1	Stateful	5	8
CVE-1999-0478	Denial-of-Service attack using excessively long headers	Stateless	1	1
CVE-1999-0976	Sendmail allows users to reinitialize the alias database, then corrupt the alias database by interrupting sendmail	Stateless	1	1

[1] These counts include a subroutine that is shared with CVE-1999-0206 that consists of 4 executable statements.

```
 ...
a->q_next = al;
a->q_alias = ctladdr;
#ifdef ESP_CVE_1999_0096
{ if (a != NULL && a->q_user != NULL) {
    if((strcmp(a->q_user,"decode")==0)||
       (strcmp(a->q_user,"uudecode")==0)){
     esp_logf("CVE-1999-0096: name='%s'\n", a->q_user);
    }
  }
}
#endif
 ...
```

Note that this detector works even if the addresses it looks for do not exist on the system and shows one of the advantages of embedded detectors: they can look for attempts to exploit vulnerabilities that do not exist on the host being monitored.

Stateful detectors

Stateful detectors are more complex than stateless ones. In the simplest cases, the detector has to collect some piece of information at an early stage before being able to make a decision later on. For example, the detector for CVE-1999-0130 needs two pieces of information to determine that an attack is occurring: the sendmail program needs to be run under the name **smtpd** and the user that invoked it must not be **root**. Because these two pieces of information are available at different points in the program, the detector is split in two code segments. The first one sets a flag when sendmail is being run as **smtpd**:

```
#ifdef ESP_CVE_1999_0130
  bool esp_RunAsSmtpd = FALSE;
#endif
 ...
else if (strcmp(p, "smtpd") == 0) {
  OpMode = MD_DAEMON;
#ifdef ESP_CVE_1999_0130
  esp_RunAsSmtpd = TRUE;
#endif
}
```

The second code segment is executed in the same block in which sendmail already generates an error message when “daemon mode” is requested by a non-root user, and generates the corresponding alert:

```
 usrerr("Permission denied");
#ifdef ESP_CVE_1999_0130
 if (esp_RunAsSmtpd) {
  esp_logf("CVE-1999-0130: user=%d\n", RealUid);
 }
#endif
 finis(FALSE, EX_USAGE);
```

A more complex example of a stateful detector is the one for CVE-1999-0047, which detects attempts to exploit a buffer overflow in the MIME-decoding subroutine of sendmail 8.8.3/8.8.4. This detector is interesting because it illustrates how in some cases it is difficult to differentiate between normal and intrusive behavior.

Under normal circumstances, the `mime7to8()` function of sendmail uses a fixed-length buffer that gets repeatedly filled and flushed as necessary while decoding a MIME message. In the vulnerable versions, a typo in the code (checking the wrong variable to see if the buffer was already full) prevented the buffer from being flushed, allowing the program to keep writing past the end of the buffer and causing the buffer overflow.

Once the problem was fixed, the buffer is correctly flushed every time it fills. However, it is impossible in the fixed code to detect an attack against this vulnerability by looking at the behavior of the program because both regular and attack data behave exactly the same: they fill the buffer, which gets flushed, and the process repeats as many times as necessary.

Therefore, to build this detector we resorted to heuristics. In this particular case, we look at the data that are being written into the buffer and compare them against the data used by the most common exploit script that was circulated for this vulnerability. This is done in the function `esp_mime_buffer_overflow()`:

```
#ifdef ESP_CVE_1999_0047
char
esp_mime_buffer_overflow(char c, int filled, char *msg) {
 char egg[]=
   "\xeb\x37\x5e
      (more binary data omitted)"
 static int pos=0;
 static int count=0;
 if (esp_match_char(egg, c, &pos, &count, 0x00, 0) && filled) {
   esp_logf("%s\n", msg);
   pos=0;
 }
 return c;
}
#endif
```

This subroutine does a character-by-character matching against the binary “egg” used by the exploit script and returns success when a complete match is found. In a more complex version of the detector, a fuzzy or partial match could be done, or the search could look for more than one binary string in the data.

From the `mime7to8()` function, the `esp_mime_buffer_overflow()` function is called every time a character is inserted in the decoding buffer:

```
  *fbufp = (c1 << 2) | ((c2 & 0x30) >> 4);
#ifdef ESP_CVE_1999_0047
  esp_mime_buffer_overflow(*fbufp, esp_filled, "CVE-1999-0047");
#endif
  ...
```

An additional heuristic used to signal an attack is that the decoding buffer must have been filled and flushed at least once when the binary string is encountered (otherwise a buffer overflow would not have occurred in the vulnerable code), so the detector also keeps track of how many times the buffer has been filled:

```
  ...
  putxline((char *) fbuf, fbufp - fbuf, mci, PXLF_MAPFROM);
  fbufp = fbuf;
#ifdef ESP_CVE_1999_0047
  esp_filled++;
#endif
}
```

This detector keeps track of several pieces of information available only inside the sendmail code, which shows the advantage that embedded detectors have by being able to access internal information of the program. This detector also shows one of the drawbacks of the embedded detectors approach: when the vulnerability for which the detector is built no longer exists in the code, it can be difficult to differentiate between normal behavior of the program and behavior under attack. This problem is common to all existing signature-based intrusion detection systems.

Testing the detectors

Each detector was tested using the exploit scripts available for each vulnerability. In most cases the exploit scripts were available from the same sources in which the problem was described, but in others we had to develop our own exploits. Each detector correctly signaled the attacks when they were launched using the exploit scripts.

4.13 Detectors implemented

After the implementation of the two case studies described in Section 4.12, the implementation continued by drawing random entries from the CVE, and implementing detectors for those that were deemed applicable according to the criteria set in Section 4.7. In total, 291 CVE entries were examined, of which 161 were not applicable, resulting in the implementation of 130 specific detectors. During this process, 20 generic detectors and 3 "pure" sensors (that collect and report information of some kind, but do not perform any detection) were designed and implemented, resulting in a total of 153 sensors and detectors implemented. They are listed in Tables A.1 and A.2.

In this section we present some information about the results of the implementation.

4.13.1 By vulnerable platform or program

Because detectors can be used to look for both successful and unsuccessful attacks, during the implementation process it was possible to build detectors for attacks that are specific for platforms other than OpenBSD, or that existed in multiple platforms and operating systems.

As mentioned in Section 3.1.4, this offers the possibility of building a "universal honeypot." However, this is not the main purpose of ESP. By building detectors for different attacks against multiple platforms, we show that the ESP architecture could be successfully applied on almost any computing platform, and aids us in gaining information about the types of vulnerabilities that are commonly responsible for computer security problems.

Figure 4.2 shows the distribution of detectors built according to the original vulnerable platform or program. This graph does not show the generic detectors, which are able to detect attacks possibly related to multiple platforms.

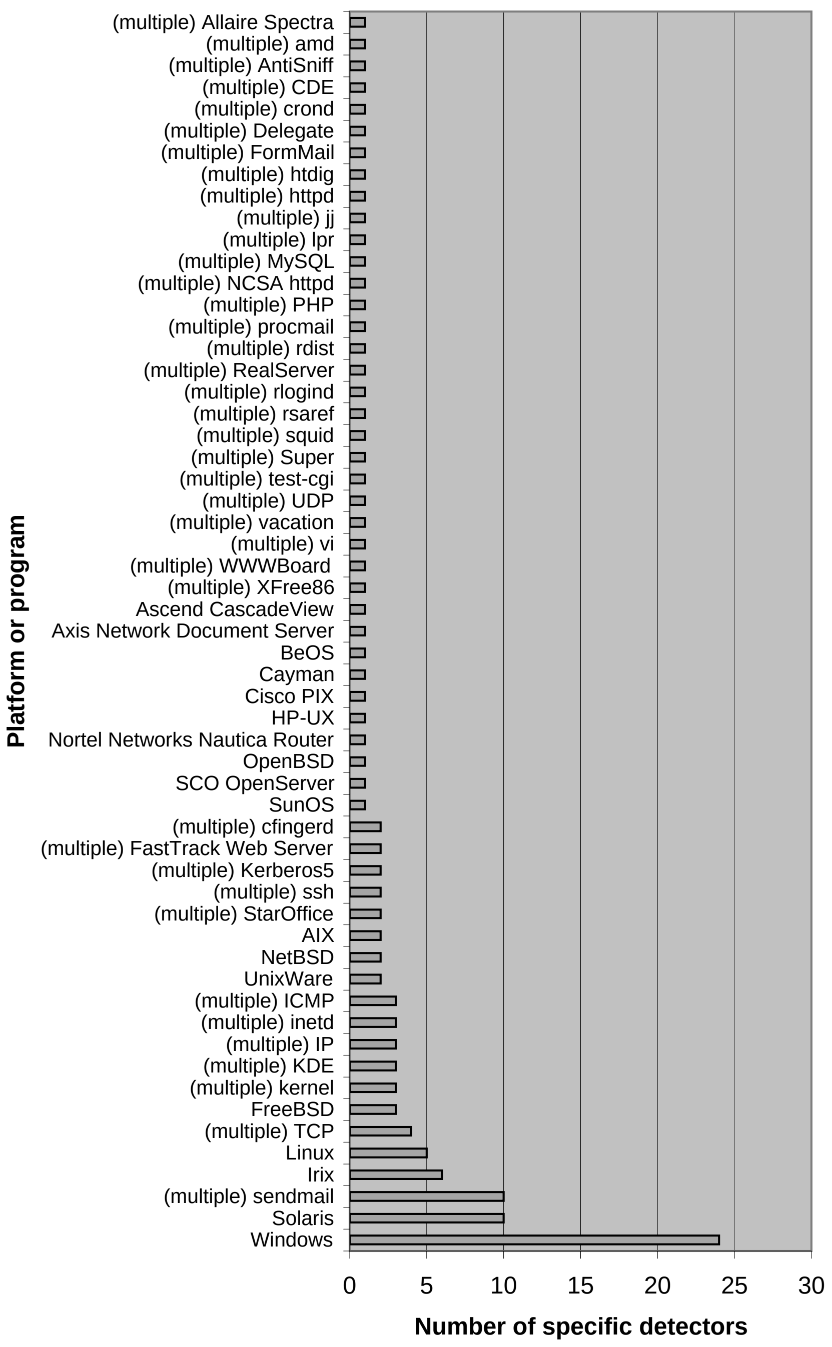


Figure 4.2. Distribution of specific detectors by vulnerable platform. This indicates the platform or program to which the intrusion for which the detector was implemented was applicable. Entries marked as "(multiple)" correspond to programs that exist in several operating systems.

4.13.2 By implementation directory

All the detectors were implemented inside the standard OpenBSD source code directory (`/usr/src/`) or inside the corresponding section of the OpenBSD ports collection (stored in directory `/usr/ports/`). We recorded the specific directory in which each detector was implemented, and the histogram in Figure 4.3 shows the overall counts. The graph shows both the number of generic and specific detectors that were implemented in each directory. Not all the detectors have an implementation directory. In particular, detectors that do not have an implementation of their own because they are covered by others do not appear in this table.

Figure 4.3 clearly shows that there are four major individual contributors: the Apache HTTP daemon (`/usr.sbin/httpd`), the networking section of the OpenBSD kernel (`/sys/netinet`), the general section of the OpenBSD kernel (`/sys/kern`), and the sendmail program (`/gnu/.../sendmail/`). These four directories account for more than 70% of the detectors. While `httpd` has the largest contribution of specific detectors, `/sys/kern` has the largest amount of generic ones.

The large representation of detectors in the `sendmail` and `netinet` directories could be partially attributed to the intentional selection of these types of detectors for the initial test runs. However, these two classes were selected precisely for the large number of attacks that have occurred in them over the years. At least for the network attacks, the number of detectors increased considerably even after the test phase.

4.13.3 By size

One of the distinguishing characteristics of the ESP architecture is its ability to perform effective detection with little overhead on the system, both in terms of CPU and memory usage. Because the detectors exist at the point in the programs where the information necessary for detection is readily available, they can be small in size.

As described in Section 4.9, we used two metrics for the size and fragmentation of the detectors: Executable Statements Added or Modified (ESAM) and Blocks of Code Added or Modified (BOCAM), respectively. Figures 4.4 and 4.5 show the distribution of detector sizes in these two units. We can see that in both measurements, the distribution is heavily

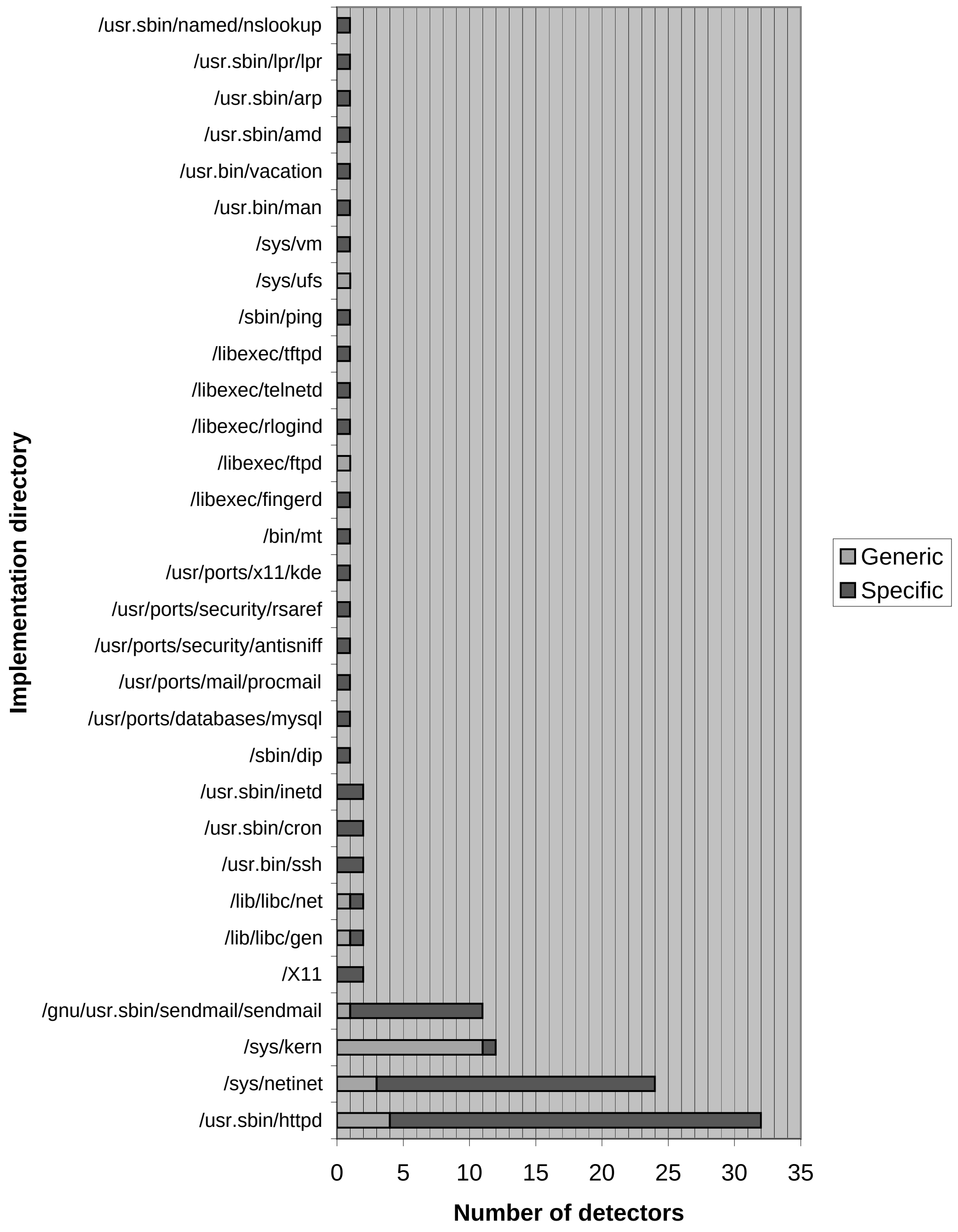


Figure 4.3. Distribution of detectors by implementation directory. For space considerations, the `/usr/src` prefix has been dropped on all entries except those within `/usr/ports`.

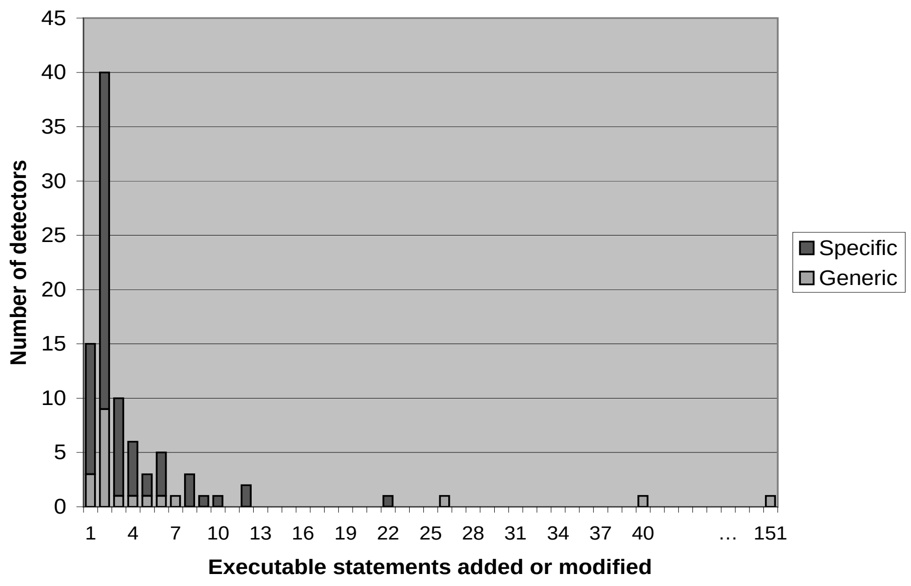


Figure 4.4. Distribution of detector sizes (ESAM metric).

biased toward the low end of the scale, with 78% of the detectors being 4 ESAM or less in size. The majority of the detectors are small in size and non-fragmented. Figure 4.6 combines the ESAM and BOCAM measurements and shows the count of detectors against each combination of ESAM and BOCAM values. This graph confirms the small-size, non-fragmented nature of most detectors.

All the detectors implemented account for 507 ESAM, resulting in an average detector length of 5.57 ESAM (this counts only the 91 detectors that have an actual implementation; the average counting all the detectors is 3.31 ESAM). All the detectors are under 50 ESAM in size, except for ESP-PORTSCAN. This is the most complex of the detectors implemented: it includes a sensor that collects information about suspicious packets received by the host and periodically traverses the list and produces the port-scanning reports. The same sensor is used by other detectors which make a decision based on packets received by the monitored host.

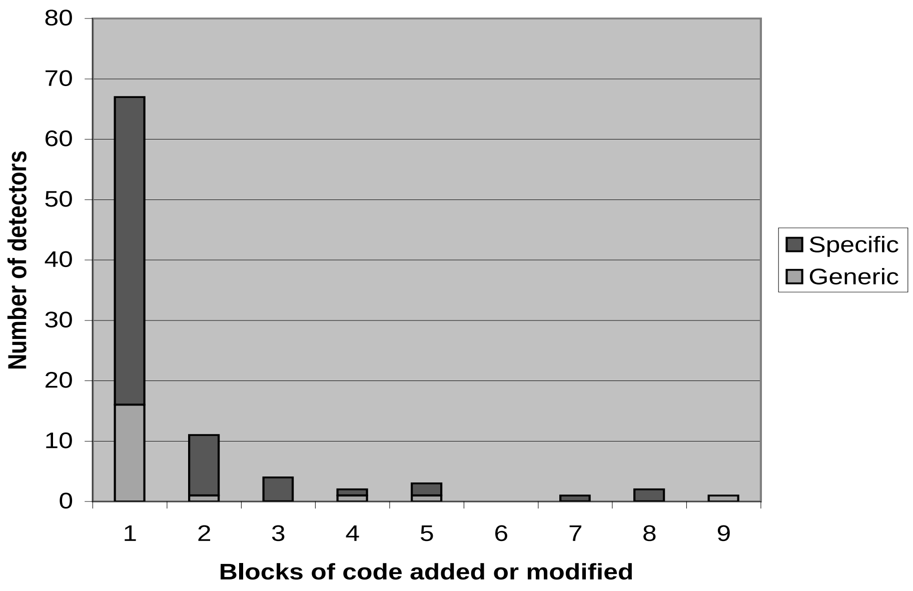


Figure 4.5. Distribution of number of contiguous code blocks per detector (BOCAM metric).

These results show that embedded detectors can be added to existing programs with few modifications to the code, and that they do not add significantly to the program they monitor in terms of size.

4.13.4 By type

By the way they operate, detectors can be classified as stateless or stateful. Figure 4.7 shows the number of each type that was implemented. We can see that the number of stateless detectors (both specific and generic) is considerably larger than that of stateful ones. This is a good result because stateful detectors usually are larger and more complex than stateful ones, as evidenced by the average ESAM count for stateful detectors being 19.14, whereas it is 2.44 for stateless detectors. Even without counting ESP-PORTSCAN (which single-handedly contributes 151 ESAM to the count), the average for stateful detectors is 9 ESAM.

Stateful detectors are individually more powerful than stateless ones, but the simplicity and small size of the stateless detectors makes it possible to put them anywhere in the

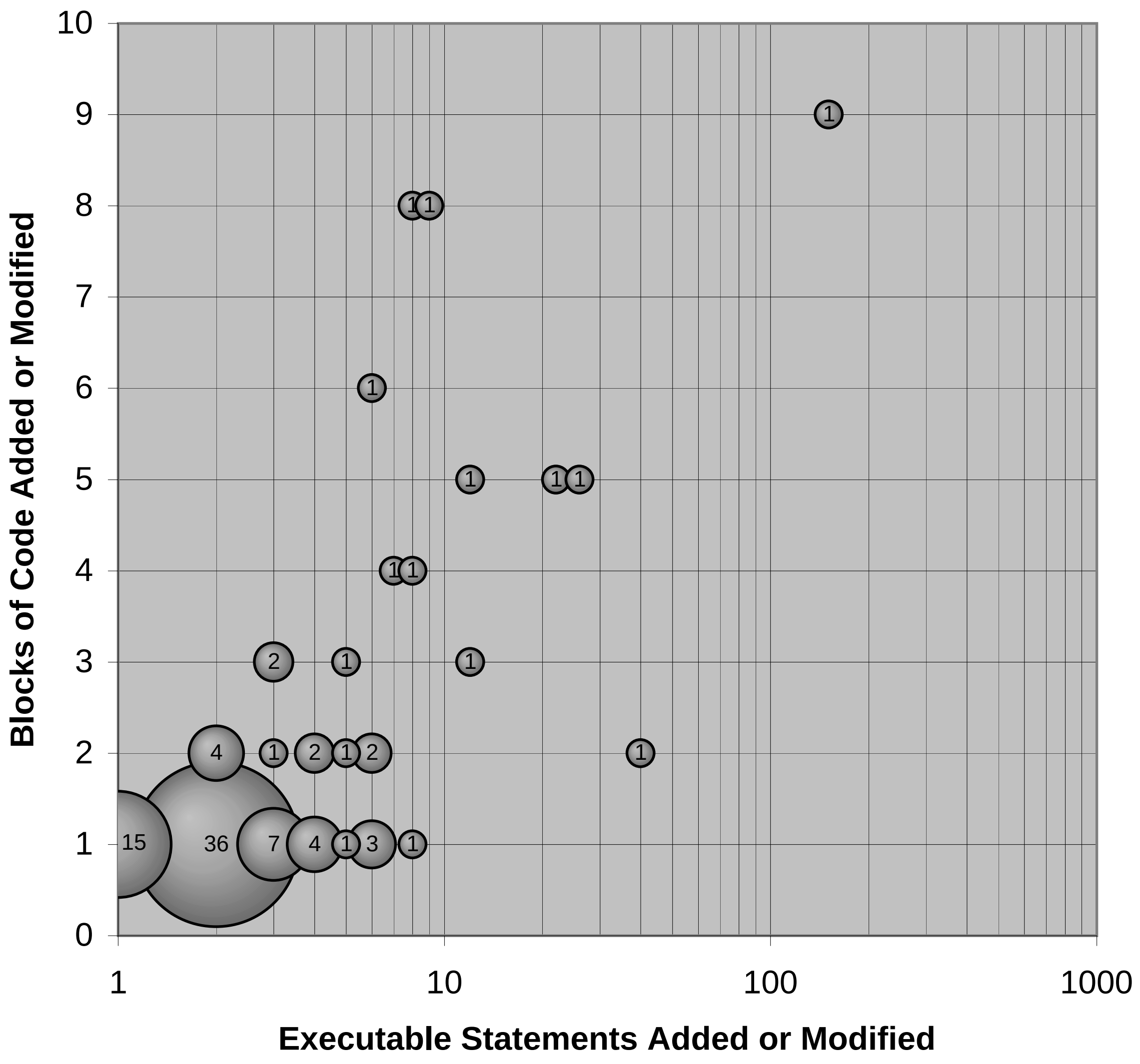


Figure 4.6. Graph of detector sizes by combination of the ESAM and BOCAM metrics. Bubble size represents the number of detectors that have each combination of parameters. The horizontal axis has been made logarithmic to better display the different values at the lower end of the scale.

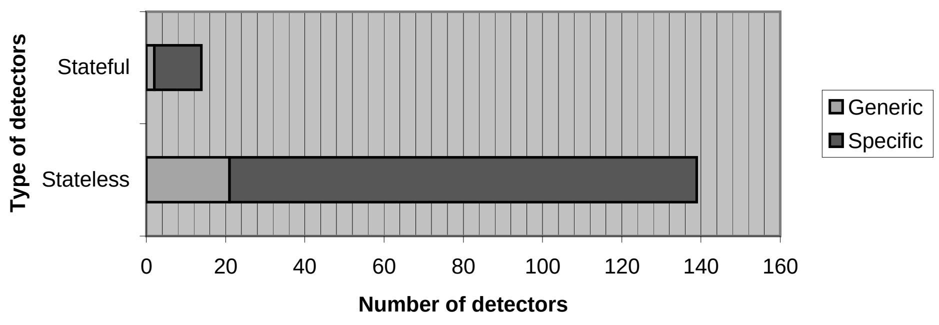


Figure 4.7. Distribution of detectors by type.

monitored component (even inside critical sections), giving the ESP architecture the ability to perform its detection functions without the need to keep as much state as traditional intrusion detection systems.

4.13.5 By data sources used

During the implementation, we recorded the sources from which each detector obtains information. These numbers are summarized in Figure 4.8. The types of data sources are defined as follows:

Network data: Data obtained from the network or from a network connection.

> Examples: contents of TCP packets received by the host; contents of ARP requests being propagated in the network.

System state: Information about the current internal state of the system.

> Examples: list of processes currently running on the system; number of currently open network connections.

User-provided data: Data provided by the user of a program.

> Examples: length of command-line arguments given to a program; contents of a configuration file created by the user.

File system state: Information about the current state of the file system.

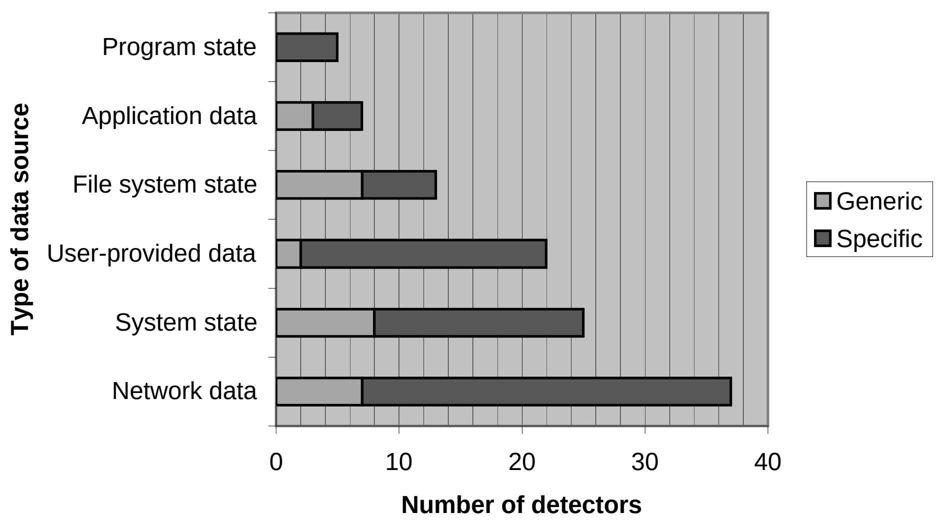


Figure 4.8. Distribution of detectors by type of data sources used. Some detectors use more than one data source.

Example: permissions of a file, existence of symbolic links in a path.

Application data: Information about data being used internally by a program.

Examples: length of an internal buffer, return value of a library function call.

Program state: Information about the current state of a program.

Examples: number of times an action has been attempted, rate of requests of a certain service.

The numbers in Figure 4.8 show that at least in our implementation, a large fraction of attacks involve data coming off the network, which is a reflection of the large number of network-based attacks that exist (and which is also reflected in the large number of detectors in the HTTP daemon and the networking layers, as shown in Figure 4.3). Also interesting is the large number of detectors with “System State” data sources. This is an indication that many problems are caused by programs not checking the conditions on the system (for example, the length of a user name or an environment variable) before performing an action. The third largest group is “User-provided data,” which corresponds to detectors for attacks that possibly could be prevented if the programs checked user input for validity (for example, command-line arguments) before using it. Although these comments are based only on the numbers encountered in our implementation, they intuitively correspond to the causes of most vulnerabilities seen in production systems.

4.13.6 By vulnerability type

Each detector is also classified according to the type of vulnerability that made the corresponding intrusion possible. For this classification we used the taxonomy of software vulnerabilities proposed by Krsul [78]. The taxonomy was used as presented originally: only category 2 ("Environmental assumptions") is expanded into sub-levels, and categories 1 ("Design"), 2 ("Coding faults") and 4 ("Configuration errors") are considered only at their top level. Figure 4.9 shows the distribution of detectors according to this classification. All the categories used in this dissertation are listed in Appendix C.

We can see that the largest number of detectors correspond to classes 2-2-1-1 (User Input — Content — is at most x), and 2-5-1-1 (Command Line Parameters — Content — length is at most x) which correspond to buffer overflow problems. Also, note that eight of the generic detectors have a classification of "n/a," which indicates that these detectors do not correspond to a specific type of vulnerability, but that can detect intrusions that exploit multiple types of vulnerabilities.

Additions to the taxonomy

The specific instantiation of the taxonomy presented by Krsul [78] was developed using the data from the vulnerability database developed in his work. When assigning categories to the ESP detectors implemented, we encountered some of them that could not be assigned to any of the existing categories. For this reason, we created some new categories as extensions to the original classification (this type of extensions was predicted by Krsul). All the new categories belong to the top-level category 2 ("Coding faults") and are described below.

(2-4-1-4) Network stream — Content — matches a regular expression. Vulnerabilities in this class correspond to those in which the programmer assumed that the data

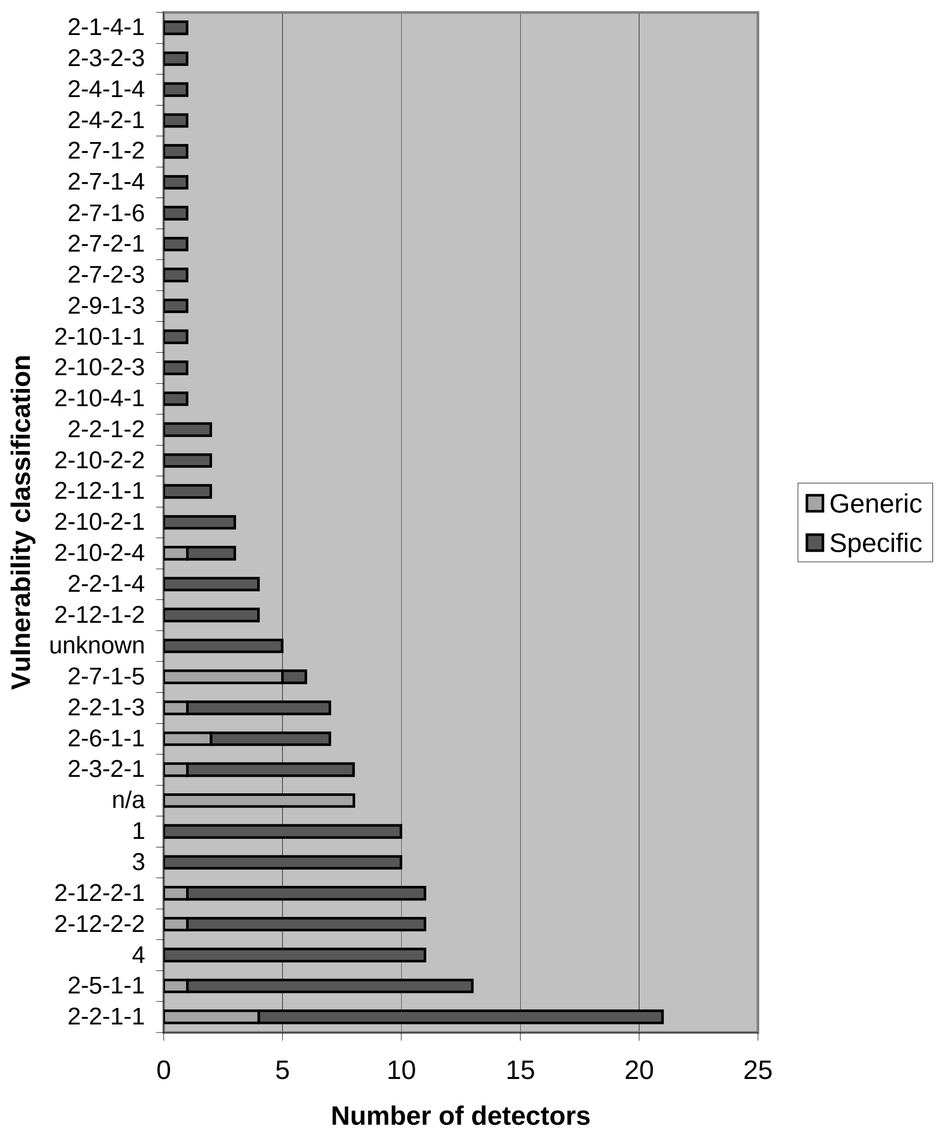


Figure 4.9. Distribution of detectors by type of vulnerability according to the taxonomy proposed by Krsul [78]. See Appendix C for a listing of the categories.

read from a network stream would always match a certain pattern or have a certain structure.

(2-4-2-1) Network stream – Socket — is the same object as x. In this case, we identified a new attribute (2-4-2: Socket) as well as a new assumption ("is the same object as x"). Vulnerabilities in this class correspond to those in which the programmer assumed that two distinct operations on a network socket will access the same connection, and that the socket will be available.

(2-7-1-6) File — Name — length is at most x. Vulnerabilities in this correspond to those in which the programmer assumed that path names would always be under a certain length.

(2-10-1-1) Network IP packets — Source address — is different than destination address. Corresponds to vulnerabilities caused by the programmer assuming that the source and the destination address on an incoming packet are always different. Note that the attribute of this class (source address) was one of the attributes predicted by Krsul in his work.

(2-10-2-2) Network IP packets — Data segment — is a proper fragment. Corresponds to vulnerabilities caused by the programmer assuming that a packet containing an IP fragment would be properly formed.

(2-10-2-3) Network IP packets — Data segment — corresponds to a fully established connection. Corresponds to vulnerabilities caused by the programmer assuming that incoming packets correspond to a connection that has already been completed successfully (this is not always the case, particularly during the initial TCP handshake).

(2-10-2-4) Network IP packets — Data segment — length is at most x. Vulnerabilities caused by the programmer assuming that the contents of a packet will not exceed a certain length.

(2-10-4-1) Network IP packets — TCP sequence number — is in proper sequence.
This is another case of a new attribute (TCP sequence number). Corresponds to vulnerabilities caused by the programmer assuming that incoming packets will have the correct sequence number.

These categories are minor additions to the original taxonomy because most of them were new assumptions about previously identified attributes. As can be seen in Figure 4.9, none of these categories had a large representation in the ESP detectors implemented, with 2-10-2-4 having the largest number of instances (3 detectors).

4.13.7 By detection and implementation rates

As described in Section 4.10, some detectors are "detected by" others, and some detectors are "implemented by" others. These relationships are interesting because they indicate the capabilities of the detectors that implement or cover others.

Figure 4.10 shows the distribution of detectors by their detection rate, as defined in Section 4.10. We can see that almost all the detectors in the list are generic detectors. Of the top six detectors, two (ESP-ARGS-LEN and ESP-LONGURL) correspond to buffer overflow vulnerabilities, two (ESP-TMP-SYMLINK and ESP-SYMLINK-OPEN) correspond to race conditions or filename-binding vulnerabilities, and one (ESP-URI-DOTDOT) corresponds to filename permissions checking vulnerabilities.

Notice that the only detector that covers other generic detectors is ESP-TMP-SYMLINK. This detector is a generalization of ESP-SYMLINK-OPEN, ESP-SYMLINK-CHMOD, ESP-SYMLINK-CHOWN and ESP-SYMLINK-CONNECT; therefore it covers the functionality of all four of them.

Figure 4.11 shows the two detectors that implement others: ESP-BADURLS (a generic detector that implements multiple detectors for web-based attacks) and CVE-1999-0052 (which implements CVE-1999-0157, corresponding to a similar attack).

4.14 Auxiliary components

In addition to the detectors themselves, there were two major components of the ESP implementation: the logging mechanism for the detectors, and the ESP library, both de-

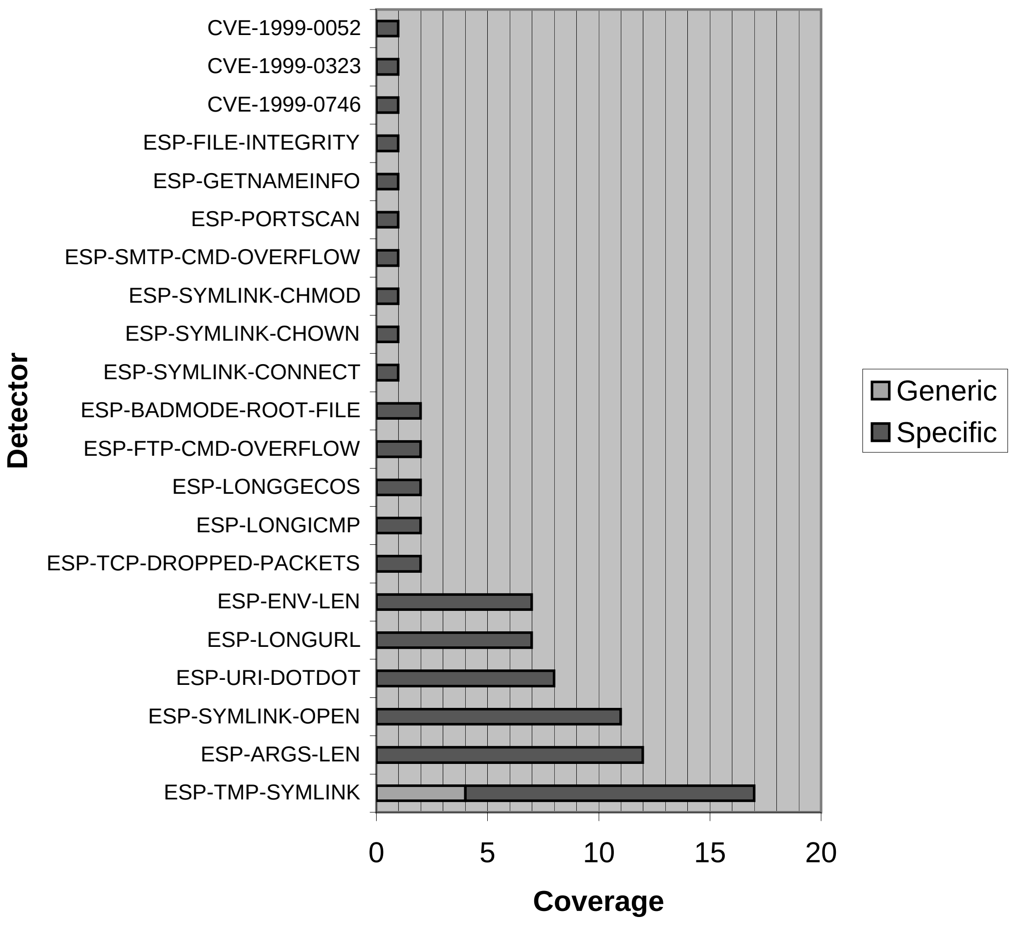


Figure 4.10. Distribution of detectors by coverage, as defined in Section 4.10.

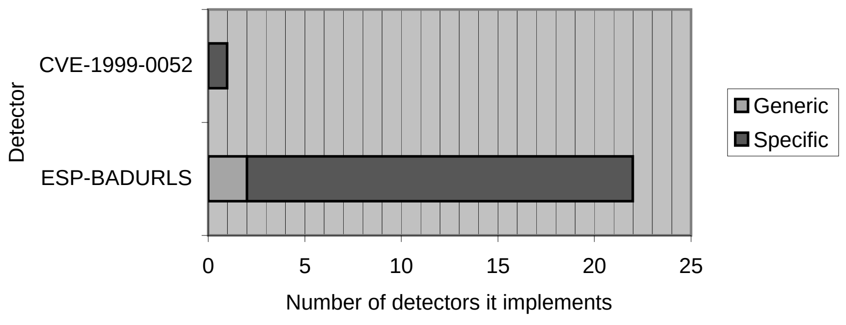


Figure 4.11. Distribution of detectors by number of detectors they implement, as defined in Section 4.10.

Table 4.3
Information about the implementation of the logging mechanism, the ESP library, and auxiliary code for the networking detectors. The `/usr/src` prefix is omitted from the implementation directories. The ESP library does not have a BOCAM count because it is implemented as an independent component.

Component	**ESAM**	**BOCAM**	**Implementation directory**
Logging mechanism	139	20	`sys/kern`, `sys/arch`
ESP library (`libesp`)	135	n/a	`lib/libesp`, `/lib/libc`
Networking code	21	1	`sys/netinet`

scribed in Section 4.5. Also, code was added to the networking layers of the OpenBSD kernel to provide support functions for some of the sensors and detectors that exist in that layer.

Table 4.3 shows some information about these three auxiliary components. We can see that the whole ESP logging mechanism (which adds the new `esp_log` system call, plus a new device file `/dev/esplog` from which messages can be read) is shorter than the ESP-PORTSCAN detector (see Table A.2). The ESP library, which includes 12 support functions plus the access point for the `esp_log` system call, has a similar size at 135 ESAM.

Some parts of the logging mechanism are in the architecture-specific portions of the kernel (`sys/arch`). This is primarily for early initialization of the memory needed by the circular buffer used in the logging mechanism. The rest of the code is architecture-independent. For this reason, the logging mechanism could be ported with relative ease to other versions of OpenBSD, and possibly to other BSD versions of Unix (such as NetBSD [94] and FreeBSD [50]).

The ESP library is implemented mostly as a separate library, except for the interface to the `esp_log()` system call and the `esp_logf()` function. These were added to the standard C library (`libc`) to make it possible for other libraries to access them and to allow detectors in any program to generate ESP messages without having to link against an additional library (unless the additional functionality is needed).

The support code for networking detectors adds some initializations and timers used for bookkeeping.

4.15 Comments about the ESP implementation

The ESP implementation, as described in this chapter, shows the feasibility of building an intrusion detection system based on the ESP architecture. With comparatively little code, it was shown possible to implement an intrusion detection system with considerable detection capabilities.

As an example, the generic (`kern` subdirectory) and network (`netinet` subdirectory) portions of the OpenBSD 2.7 kernel consist of roughly 88,830 raw lines of C code, including comments, blank lines and preprocessor directives[1]. All the ESP detectors that have been implemented in those sections of the kernel total approximately 1,340 lines of code (again, including comments, blank lines and preprocessor directives), which corresponds to 1.5% of the size of the kernel code, yet detect 54 specific attacks, and include 9 generic detectors.

Because the purpose of this ESP implementation was to explore the capabilities and issues related to the architecture, we implemented some detectors that would probably not be needed in a production system, such as detectors for attacks that do not correspond to the implementation architecture. However, by doing so we demonstrated that the concepts of the ESP architecture can be used on a wide variety of platforms to detect a wide variety of attacks.

The most significant drawback in our implementation of the ESP architecture was the cost of the implementation in terms of effort and time. Because we were modifying an existing system, a significant effort was spent in understanding the code before being able to make meaningful modifications. However, the knowledge about the most useful detectors and types of data can be applied in the design and implementation of future systems which include the internal sensors and embedded detectors needed for covering the most common intrusions and attacks. The effort and time needed to implement sensors and detectors could

[1]This is the only case in which we use lines of code as a metric because of the limitations of the Metre tool [85], which made it difficult to measure ESAM for the whole kernel.

be significantly smaller if they were implemented by the original authors of the programs, possibly aided by component libraries or automated tools, as mentioned in Section 6.3.

5. TESTING THE ESP IMPLEMENTATION

After the initial ESP implementation was completed, a series of tests was performed to measure its responses and to obtain qualitative and quantitative results about its behavior. The tests were designed to evaluate the performance impact of the ESP intrusion detection system on an instrumented host and its detection abilities for previously unknown attacks.

5.1 Performance testing

5.1.1 Test design and methodology

The purpose of the performance tests was to determine the impact that the ESP sensors and detectors have on the instrumented host, under severe but non-intrusive operating conditions (the detectors were not triggered during these tests). For this purpose, we decided to focus on two groups of detectors:

1. Detectors in the networking portions of the kernel (24 detectors).

2. Detectors in a web servers (32 detectors).

These are the two largest groups of detectors (see Figure 4.3) and are good representatives of detectors in the kernel and in a user space application respectively. The detectors are additional code to be executed. Because they do not interfere with their surrounding code, their main impact is in terms of additional execution time. We measured CPU utilization and compared systems compiled with and without the detectors.

The general setup for the tests was as shown in Figure 5.1: One server B, that would be the one instrumented with the detectors when appropriate, and where the CPU utilization would be measured; and a client A, from which the tests would be launched against B. These two machines were on a dedicated point-to-point network connected with a third machine R operating as a transparent bridge between A and B. The purpose of R was to allow the artificial reduction of the bandwidth available for the connection between A and

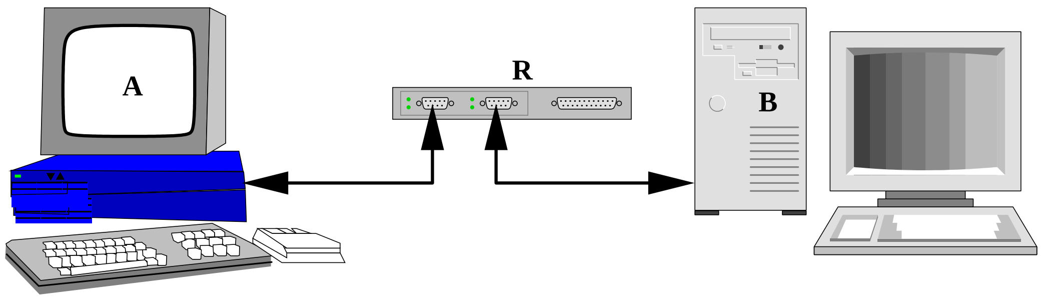


Figure 5.1. General setup for the performance tests of the ESP implementation. Host B is the server, host A is the client, and R represents a host acting as a transparent bridge between A and B. The three hosts are on dedicated point-to-point connections over 100Mbit/s full-duplex Ethernet.

B. Hosts A and B were 600 MHz Intel Pentium III machines with 128 MB RAM running OpenBSD 2.7, and host R was a 700MHz Intel Celeron machine with 128 MB RAM running FreeBSD [50] and **dummy_net** [120] for imposing constraints on the bandwidth. Unnecessary programs and services were stopped on the test machines (including the **syslog** and **cron** daemons, and the X Windows system) to reduce the factors that could confound the performance measurements.

The first test was done using a subset of the NetPerf [62] benchmark. The NetPerf test we used measures network performance as the maximum throughput between two hosts by sending a stream of data from a source to a sink over a TCP or UDP connection. We selected the TCP version of the test because 16 of the 24 detectors implemented in the networking sections are in the IP or TCP layers. In this test, the independent variable was the maximum bandwidth allowed between the source (A) and the sink (B) and was controlled by setting bandwidth constraints on R using **dummy_net**. We measured CPU utilization on B under increasing bandwidth, from 5 Mbps up to 100 Mbps.

In the second test, a web server was running on host B while host A was generating requests. The web server used was Apache [5] as included in the OpenBSD 2.7 distribution. We used **http_load** [1] to generate the requests by randomly choosing URLs from a list. The independent variable in this test was the number of simultaneous connections that

Table 5.1
Summary of the parameters for the performance tests. Both tests were repeated for the ESP and NOESP cases. L represents the length of each test repetition, and S the number of samples of the CPU load taken during each repetition.

Test name	**Indep. variable (X)**	**Range of X**	**L (sec)**	**S**
NetPerf	Bandwidth	$5 - 100$ Mbps	60	54
http_load	Parallel HTTP connections	$5 - 100$	60	54

http_load was allowed to establish. We measured CPU utilization on host B under an increasing number of simultaneous connections, from 5 up to 100.

For each value of the independent variable, twenty runs of the test were performed. All the runs were duplicated in two blocks: one for host B with detectors (ESP block) and one without detectors (NOESP block). Each run lasted for 60 seconds and during that period, snapshot observations of the CPU load in host B were taken each second. The CPU load was obtained using the **top** [83] command, which uses information gathered in the `statclock()` function within the kernel context switch [87, p. 58]. Three observations at the beginning and the end of each run were ignored (to eliminate ramp-up and ramp-down measurements), and the rest (54 observations) were averaged to obtain an average CPU load for each run.

All the information for each test is summarized in Table 5.1. The order in which the independent variable was modified for the NetPerf and http_load tests was generated using a pseudo-random number generator with a fixed seed to be able to reproduce the sequence.

Before each block all the systems were rebooted and a "warm-up" sequence was run by applying all the values of the independent variable in increasing order. This was done to bring the hosts to a stable state in terms of caching, disk spinning and any possible unknown factors, before taking any measurements.

Although we attempted to arrange the experimental setup to minimize extraneous effects on the measurements, there are still factors that could affect them, including virtual memory, process scheduling and caching. The measurement process itself runs on the

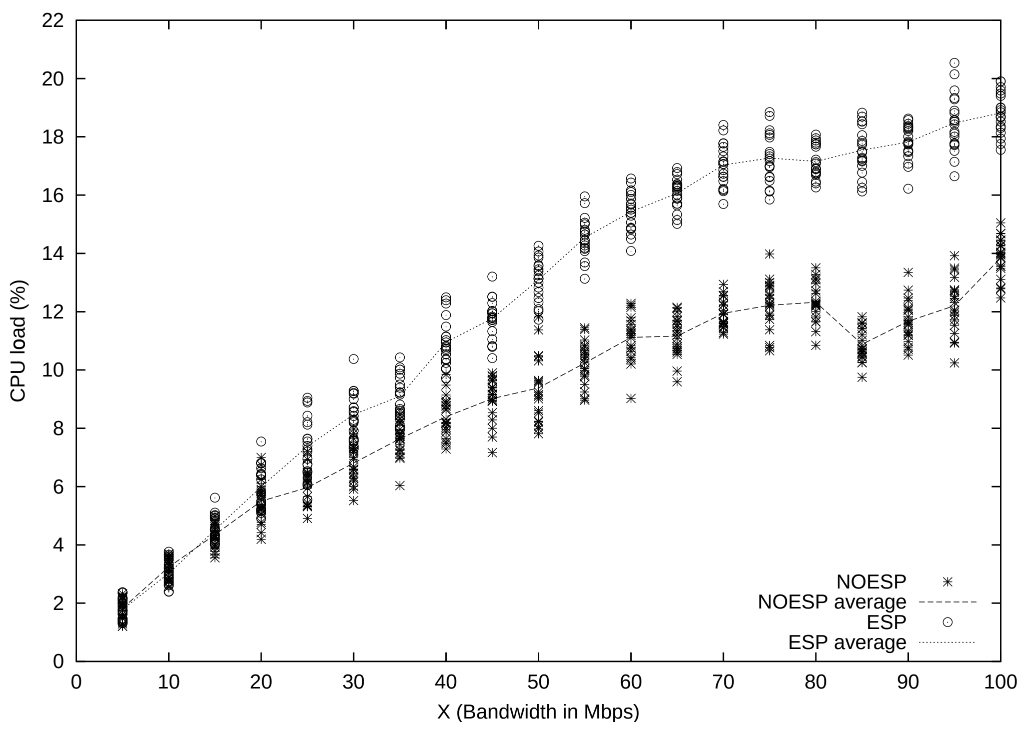


Figure 5.2. Plot of the CPU utilization measurements from the NetPerf experiment, showing the mean values for the ESP and NOESP cases.

CPU being measured, which may affect the observations as well. Finally, as mentioned, the results reported consist of an average of averages, which may compound errors in the measurements.

However, the purpose of these experiments was to compare the behavior of hosts with and without detectors, and not to establish absolute measurements of performance. As such, this setup and methodology is adequate for showing the impact that embedded detectors have on the host in which they reside.

5.1.2 Results of the NetPerf test

Figure 5.2 shows the CPU measurements obtained in host B during the execution of the NetPerf experiment. There are 20 points at each value of X for each block (ESP and NOESP) and the lines connect the mean values at each value of X. We can see in this graph that for lower values of X, the CPU utilization is essentially the same, but the dif-

Table 5.2
Statistics and analysis results for data from the NetPerf experiment.

	Mean CPU %			**Difference**	
X	**NOESP**	**ESP**	**Diff.**	**95% C.I.**	**p-value**
5	1.8296	1.7878	-0.0418	-0.4775–0.3939	0.8507
10	3.2396	3.0246	-0.2150	-0.6507–0.2207	0.3330
15	4.3484	4.5107	0.1624	-0.2734–0.5981	0.4647
20	5.5018	5.9936	0.4918	0.0560–0.9275	0.0270
25	5.9651	7.3771	1.4120	0.9762–1.8477	< 0.0001
30	6.8102	8.4731	1.6629	1.2272–2.0987	< 0.0001
35	7.6407	9.0994	1.4588	1.0231–1.8945	< 0.0001
40	8.3942	10.9556	2.5613	2.1256–2.9970	< 0.0001
45	9.0185	11.7683	2.7498	2.3141–3.1856	< 0.0001
50	9.3859	13.0943	3.7084	3.2727–4.1441	< 0.0001
55	10.2375	14.5434	4.3059	3.8701–4.7416	< 0.0001
60	11.1171	15.4268	4.3097	3.8740–4.7455	< 0.0001
65	11.1625	16.0658	4.9032	4.4675–5.3389	< 0.0001
70	11.9442	17.0314	5.0873	4.6515–5.5230	< 0.0001
75	12.2195	17.2707	5.0511	4.6154–5.4869	< 0.0001
80	12.3222	17.1559	4.8337	4.3979–5.2694	< 0.0001
85	10.8815	17.5441	6.6625	6.2268–7.0983	< 0.0001
90	11.6744	17.8184	6.1440	5.7082–6.5797	< 0.0001
95	12.2040	18.4758	6.2718	5.8361–6.7075	< 0.0001
100	13.8461	18.8180	4.9719	4.5361–5.4076	< 0.0001

ference grows larger as X increases, because the detectors in the networking layers of the kernel introduce additional work that needs to be done for every packet that is received. To quantify the difference, a pair-wise F-test was done for each value of X, and its results are shown in Table 5.2. The p-values in this table show that the difference in means between ESP and NOESP can be considered statistically non-significant up to about $X = 20$, but after that point it is statistically significant. Figure 5.3 shows the difference between the means at each point, with its 95% confidence interval.

The results of this experiment are what would be expected, with the detectors having a larger impact as the amount of work that the system does increases. The detectors can have a considerable impact on the CPU load of the host, particularly for high values of

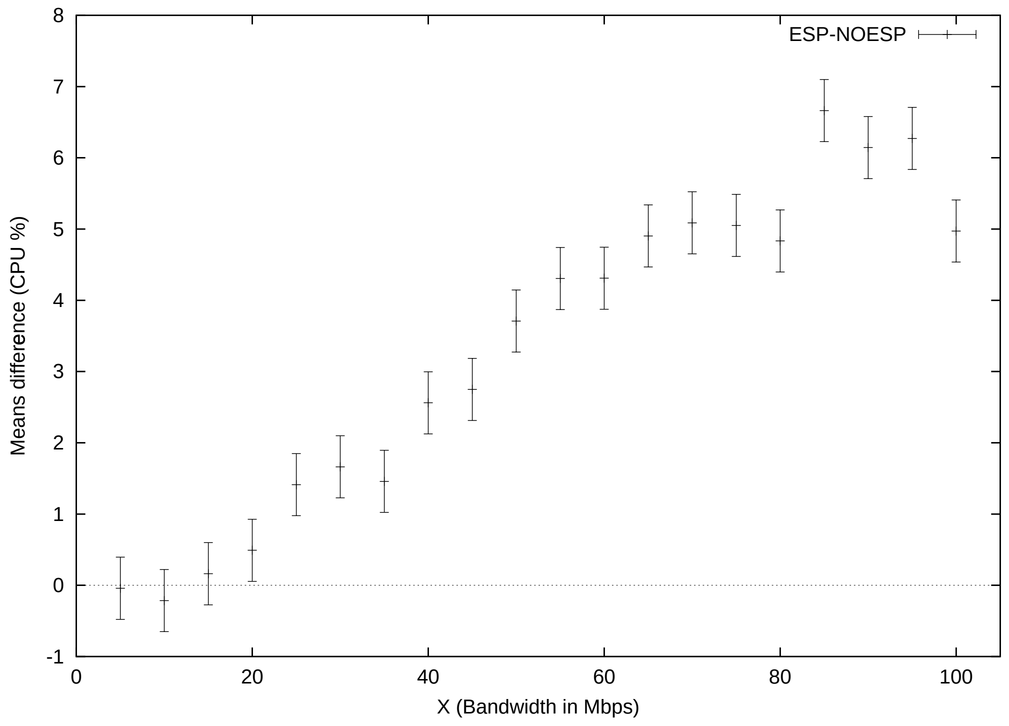


Figure 5.3. Difference in mean CPU utilization between the system with (ESP) and without (NOESP) detectors in the NetPerf experiment, with the 95% confidence interval for the difference at each point.

X. However, we should keep in mind that this test was specifically designed to stress the host by maintaining a constant stream of the appropriate bandwidth fed to it. Under normal operating conditions, the average network load being processed by a host is lower; therefore the impact of the detectors should not be as noticeable. Furthermore, although the difference is *statistically* significant, it is never more than 7% of CPU utilization, which in practical terms could be considered acceptable. At its maximum value (for $X = 85$), the difference in means is 6.6%. The NetPerf test is exercising a maximum of 24 detectors (those implemented in the networking layers of the kernel), so on average each detector adds less than 0.3% to the CPU load of the system. In reality, not all detectors have the same impact (because of their functionality, implementation, and where they are placed), but this number is an indication of the small impact that each individual detector has.

5.1.3 Results of the http_load test

The CPU measurements obtained in host B during the execution of the http_load experiment are shown in Figure 5.4, with the mean values plotted as lines. The results from a pair-wise F-test are shown in Table 5.3, and Figure 5.5 shows the difference between the means with their 95% confidence interval.

In this case, the results are indicative of the detectors having a large impact on the CPU utilization of the host. Counter intuitively, we can see that the CPU utilization on the system with the detectors decreases as X increases. This can possibly be attributed to caching effects (as the load increases, there is a larger chance that simultaneous or sequential requests will be for the same URL), but it should be the subject of further study.

5.1.4 Comparison and comments about the tests

The results of the NetPerf experiment are not surprising and show that the impact of the detectors increases as the network load increases. The impact of the detectors on the host could potentially be reduced by improving the implementation of some of the detectors, particularly the ESP-PORTSCAN detector. It keeps considerable state and maintains some complex data structures, so it is likely to be one of the major contributors to the impact that the detectors have.

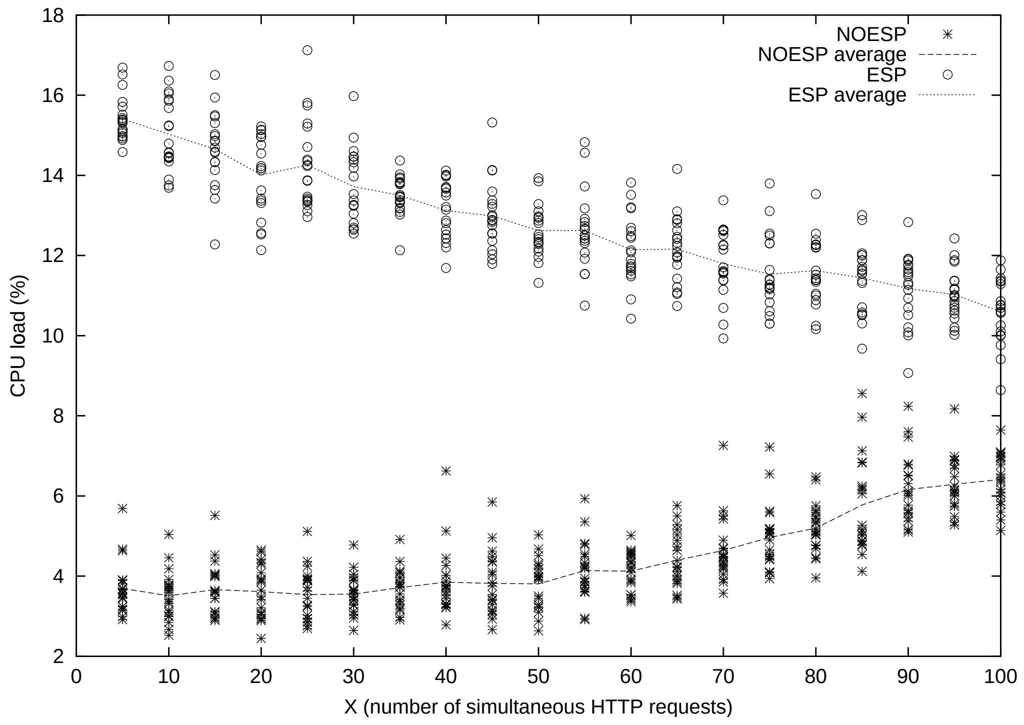


Figure 5.4. Plot of the CPU utilization measurements from the http_load experiment, showing the mean values for the ESP and NOESP cases.

Table 5.3
Statistics and analysis results for data from the http_load experiment.

	Mean CPU %			**Difference**	
X	**NOESP**	**ESP**	**Diff.**	**95% C.I.**	**p-value**
5	3.6933	15.4078	11.7145	11.2280–12.2011	< 0.0001
10	3.5093	15.0285	11.5191	11.0326–12.0057	< 0.0001
15	3.6630	14.6531	10.9901	10.5036–11.4766	< 0.0001
20	3.6128	14.0115	10.3987	9.9121–10.8852	< 0.0001
25	3.5372	14.2591	10.7220	10.2354–11.2085	< 0.0001
30	3.5509	13.7199	10.1691	9.6825–10.6556	< 0.0001
35	3.7131	13.5013	9.7882	9.3017–10.2747	< 0.0001
40	3.8526	13.1243	9.2718	8.7852– 9.7583	< 0.0001
45	3.8177	12.9858	9.1680	8.6815– 9.6546	< 0.0001
50	3.8070	12.6226	8.8156	8.3290– 9.3021	< 0.0001
55	4.1388	12.6228	8.4840	7.9975– 8.9706	< 0.0001
60	4.1258	12.1468	8.0210	7.5345– 8.5076	< 0.0001
65	4.3988	12.1588	7.7600	7.2735– 8.2465	< 0.0001
70	4.6428	11.7982	7.1554	6.6688– 7.6419	< 0.0001
75	4.9663	11.5338	6.5675	6.0809– 7.0540	< 0.0001
80	5.1974	11.6255	6.4280	5.9415– 6.9146	< 0.0001
85	5.7775	11.4404	5.6628	5.1763– 6.1494	< 0.0001
90	6.1601	11.1788	5.0187	4.5321– 5.5052	< 0.0001
95	6.2901	11.0235	4.7334	4.2468– 5.2199	< 0.0001
100	6.4134	10.6010	4.1876	3.7011– 4.6742	< 0.0001

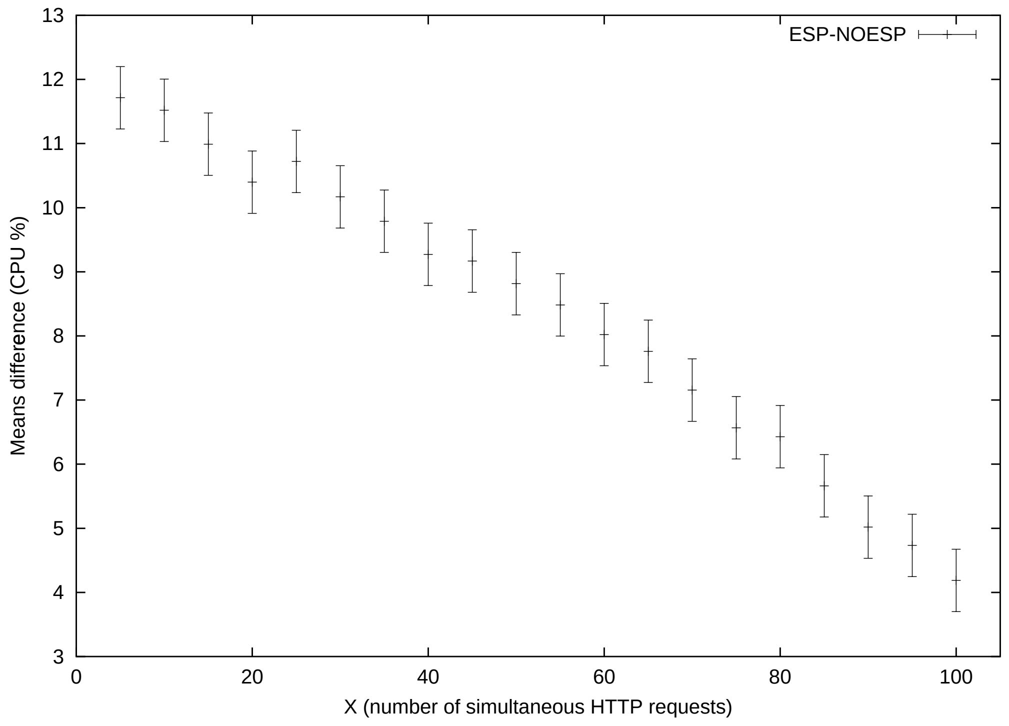


Figure 5.5. Difference in mean CPU utilization between the system with (ESP) and without (NOESP) detectors in the http_load experiment. Also shown is the 95% confidence interval for the difference at each point.

Overall, the NetPerf results can be seen as indicative of detectors that were designed and implemented with moderate success: their impact is proportional to the amount of activity in the host, and their impact is not excessive.

The http_load results are visibly different from the NetPerf results. Initially there is an extremely large difference between the ESP and NOESP cases (over 300% for $X = 5$), but this difference decreases as X increases to the point where it is 65% at $X = 100$. While still being a considerable difference, the reduction from the initial value is dramatic.

The reduction in difference could be explained by a number of factors, including caching effects on the web server. The http_load makes requests from a fixed-size list of URLs, and as the number of simultaneous requests increases, the likelihood of requesting the same URL several times simultaneously or in close sequence increases. If the web server is doing any caching of requests (so that it can serve the same request multiple times without having to do all the processing repeatedly), it could account for the reduction in CPU load for larger values of X.

More interesting for our purposes is the large effect that the detectors have on the CPU load. This can be explained by the types of detectors involved. The largest detector in the HTTP server is ESP-BADURLS, a generic detector that does not directly cover any other detectors, but that provides mechanisms for implementing several others (see Figure 4.11). These mechanisms consist mainly of a string-matching capability for detecting different web-based attacks. What ESP-BADURLS does for every HTTP request is to sequentially compare it against several strings using different qualifications (such as anchoring the test string in different parts of the request, checking the arguments of the URL, etc.). In this respect, ESP-BADURLS is different from all the other detectors (which check for a fixed condition) and could be expected *a priori* to have a larger impact on CPU utilization. Furthermore, the initial implementation of ESP-BADURLS (used in these tests) uses a naive approach, sequentially and blindly comparing the strings against each request. By improving the implementation to use an efficient regular expressions engine [e.g. 57, 67] or some other string-matching mechanism, it may be possible to reduce the impact of the HTTP detectors considerably.

In conclusion, these tests point out a major consideration for the use of internal sensors: they are heavily dependent on implementation decisions, and a different implementation might make a significant difference in performance and impact. Moreover, extreme care must be taken in their implementation because when not implemented carefully, they can have a severe impact on the performance of the monitored component. These tests are not intended to provide measurements of the performance costs of embedded detectors in general, but only of our implementation, and to show the feasibility of doing low-overhead intrusion detection using the ESP architecture.

This consideration is one of the reasons why internal sensors had not been extensively studied before. Their implementation is complex, and the possible consequences are severe. However, as will be described in the next section, the payoff in detection capabilities can be significant, and worth the work necessary to implement the sensors and make them efficient.

5.2 Detection testing

The purpose of the detection test was to determine the validity of the second hypothesis (see Section 1.4): by using internal sensors it is possible to detect new attacks. Additionally, we wanted to get an idea of the effort needed to improve the detection capabilities of the detectors when necessary. To this end, a number of previously unknown attacks were tested against a host instrumented with ESP detectors.

5.2.1 Test design and methodology

As a source of information, we monitored the BugTraq mailing list [13] for a period of slightly over one month, from May 3, 2001 to June 8, 2001. During this period, 157 messages to the mailing list were examined corresponding to reports of new vulnerabilities, attacks and exploits against different systems. Of these messages, 80 were determined to be applicable using the criteria defined in Section 4.7. Additionally, only messages that described specific attacks (and not only generic or vague vulnerabilities) were selected as applicable.

The 80 applicable attacks were tested against a host instrumented with the ESP implementation. We performed two types of testing depending on the attack:

Real testing: When an attack could be directly attempted against the OpenBSD system running ESP, we did so and recorded any responses from the existing detectors.

For example, one of the attacks tested was against the **crontab** [141] program. Because this program exists in OpenBSD, the exploit script could be run directly in our test system to determine whether the detectors would react to it.

Simulated testing: Sometimes an attack was not directly executable in our test platform—for example, because it used a program that does not exist in OpenBSD, or because it was specific to some other architecture. However, if the workings of the attack were clear enough, we did a "simulated testing" of the attack by studying its properties and determining whether any of the existing detectors would react to that attack if it were attempted against a system instrumented with ESP.

For example, another one of the attacks examined was against the **scoadmin** program [125] in the Unixware operating system. The affected program does not exist in OpenBSD, but the exploit was clear enough to show that it worked because **scoadmin** followed a symbolic link in the `/tmp` directory. By this reasoning, we could determine that the attack would have been detected by the ESP-SYMLINK-OPEN detector.

After testing, each attack was classified in one or more of the following categories (each category has a letter code associated with it):

Detected (D): The attack was detected by one or more of the existing detectors. In this case, we recorded the names of the detectors that reacted to the attack.

Detected if successful (DS): In some cases, the attack itself was not detected, but its effects would be if the attack were to be successful. In these cases, we also recorded which detectors would be triggered by the successful attack.

For example, the attack against **cron** mentioned before was not immediately detected by any of the existing detectors. However, on success the attack would have created a root-owned set-UID copy of a shell, and this action would trigger the ESP-

BADMODE-ROOT-FILE detector, so we classify this attack as "detected if successful."

Detectable with modifications to existing detectors (DM): Some attacks were not detected by any of the existing detectors, but a reasonably small change to one of them would be sufficient to make the attack be detected. We considered as "reasonably small" changes that involved tuning some parameter of the detector, or slightly extending their functionality. In this case, we recorded the detector to which the changes would have to be made, and what those changes would be.

For example, one of the entries reviewed was a web-based attack that used a "dot-dot" (`../`) path to access files outside the normal web document directories, but with the variation that parts of the string were encoded in their hexadecimal representations to bypass checks at the web server (for example, `../` could be encoded as `.%2e%2f`, where 0x2E and 0x2F are the ASCII codes for a dot and a slash respectively). This attack was not immediately detected by the existing ESP-URI-DOTDOT detector, but a small addition to make it "unescape" the strings before checking them would enable it to detect the attack.

Detectable with creation of new detectors (DC): Some attacks were not detected by the existing detectors, but they could be by creating a new one. When the new detector would be a generic one—so that it would be able to detect multiple attacks and not only the one under testing—we considered this change as acceptable, because it provides for detection possibilities beyond the attack that prompted its creation. In this case, we recorded the type of detector to create, its conditions for triggering, and a proposed name for it.

For example, one of the attacks tested was a web-based buffer overflow, but using a long string in one of the HTTP headers included in a request (instead of being a long URL), so the existing ESP-LONGURL detector did not react to it. However, by implementing a similar detector called ESP-HTTP-HDR-OVERFLOW, which performs length checks on HTTP request headers, this attack (and possibly others) could be detected.

Detectable if successful with modifications to existing detectors (SM):
This is similar to the DM category, but for the case in which the modifications to an existing detector would cause the attack to be detected only if successful.

This is a possible category, but during the test no attacks were assigned to it.

Detectable if successful with creation of new detectors (SC): This is similar to the DC category, but for the case in which a new generic detector could be created to detect a successful attack.

Only one entry was found in this category during the testing. It led to the creation of the ESP-PRIV-ESCALATION detector, which has the potential to detect many buffer overflow and race condition attacks in which a process acquires elevated privileges.

Not detectable (ND): An attack was considered in this category when the only way to detect it would have been to create a new specific detector for it. Creating a specific detector does not provide any future benefits (possibility for detection of other attacks other than the current one), so it was not considered as an acceptable change for our purposes.

For example, one entry reviewed consisted of an attack against the **xfs** (X font server) in certain versions of XFree86, which would crash when fed a long random string, causing a denial-of-service attack. For detecting this attack, it would be necessary to implement a new specific detector in the **xfs** code. Therefore we consider it as not detectable.

An entry can belong to any of these categories and can also belong to both DS and DC (DSDC) or DS and DM (DSDM). This occurs when an attack is detectable if successful but it could also be detected by either creating or modifying a detector.

Table 5.4
Number of attacks in each category for the four batches examined during the detection tests. The "Total" column shows the counts for the whole test. The TD and TDM categories represent the sum of the other fields in each section, and correspond to "Total number of attacks detected" and "Total number of attacks detectable with changes" respectively.

	Batches				
Category	**#1**	**#2**	**#3**	**#4**	**Total**
Non-applicable	20	17	22	18	77
Applicable	20	20	20	20	80
D	6	9	8	9	32
DS	1	3	0	2	6
TD (D+DS)	7	12	8	11	38
DM	6	4	3	0	13
DC	3	2	3	0	8
SC	0	1	0	0	1
TDM (DM+DC+SC)	9	7	6	0	22
DSDC	1	2	0	0	3
ND	5	3	6	9	23

The testing was divided in four batches of 20 attacks. After every batch, all the changes recorded for detectors in categories DM, DC and SC were applied, so after each batch all the entries in those categories would belong to category D.

5.2.2 Results from the detection test

In total, 157 attacks were examined, of which 80 were applicable. Of these, 47 were done with real testing, and 33 with simulated testing.

The number of attacks in each category for each one of the batches and for the whole test are shown in Table 5.4. The total categories (TD, TDM and ND) are displayed also in Figure 5.6. In this chart, we can see that the number of attacks detected was consistently over 30% in each of the batches. When counting the attacks that were detected after making modifications to the detectors, the number was consistently over 50%.

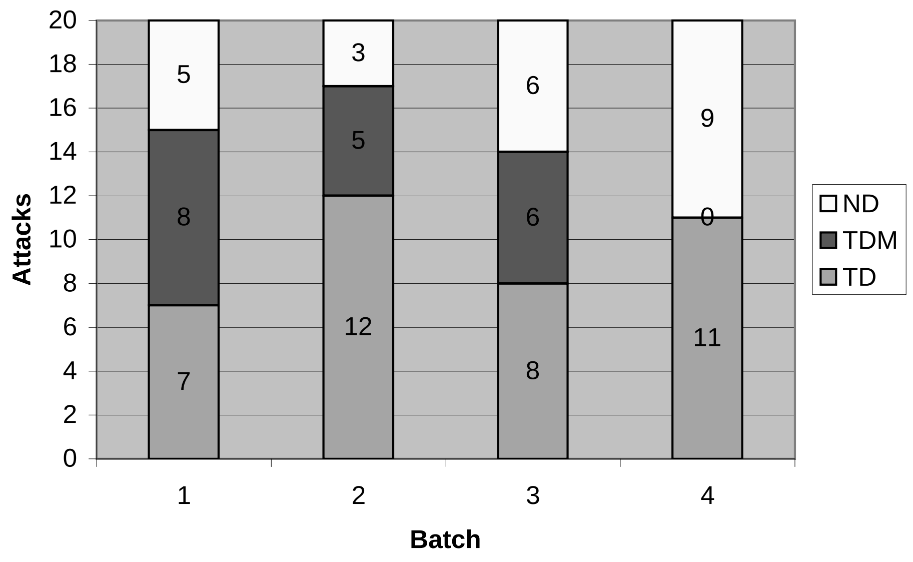


Figure 5.6. Total of attacks marked as “detected”, “detected with modifications” and “not detected” for each one of the batches of the detection test.

Keeping in mind that each batch incorporates the changes made after the previous batch, it is also of interest to analyze the total results at the beginning and at the end. This is, if all the 80 attacks had been applied to the original detectors, how many would have been detected? By comparing this with the number of attacks detected at the end of the test (after all the changes were made), we can observe the impact that the changes had in the detection capabilities. Figure 5.7 shows these numbers graphically. We can see that even without any modifications, the original ESP detectors would have been able to detect 35% of the new attacks (41.2% if we count the ones in group DS). After making the changes, the detection rate went up to 62.5% (71.2% with DS).

Original detection capabilities

Of the 80 attacks exercised during this test, 33 would have been detected by the ESP detectors as originally implemented. Table 5.5 shows the distribution of the detectors involved. Table 5.6 shows the distribution of the Krsul categories assigned to each attack (see Appendix C).

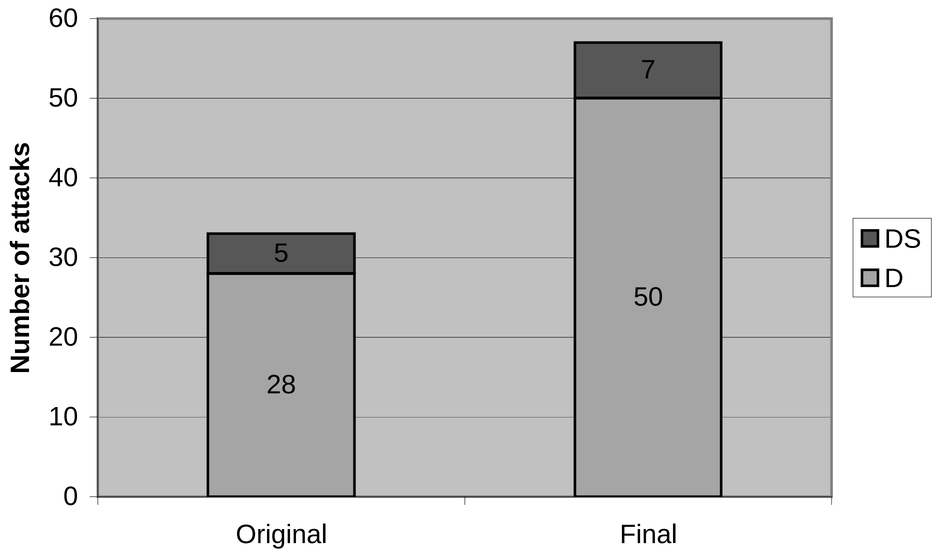


Figure 5.7. Total number of attacks that would have been detected by the original detectors, and by the detectors after the changes.

Table 5.5
Distribution of original ESP detectors that responded to the attacks in the detection test. The percentages are with respect to the total number of attacks in the test (80).

Detector	Attacks detected	% of Total
ESP-SYMLINK-OPEN	13	16.25
ESP-BADMODE-ROOT-FILE	4	5.00
ESP-FTP-CMD-OVERFLOW	4	5.00
ESP-LONGURL	3	3.75
ESP-URI-DOTDOT	3	3.75
ESP-ARGS-LEN	2	2.50
ESP-ENV-LEN	1	1.25
ESP-FILE-INTEGRITY	1	1.25
ESP-PORTSCAN	1	1.25
ESP-SNMP-EMPTY-PACKET	1	1.25
ESP-TMP-SYMLINK	1	1.25

Table 5.6
Distribution of categories in the Krsul classification for the attacks to which the original ESP detectors responded. See Appendix C for the definitions of each category. The percentages are with respect to the total number of attacks in the test (80).

Category	**Attacks detected**	**% of Total**
2-12-2-1	14	17.50
2-2-1-1	6	7.50
2-12-2-2	3	3.75
3	3	3.75
2-2-1-4	1	1.25
2-3-2-1	1	1.25
2-4-1-1	1	1.25
2-5-1-1	1	1.25
2-7-1-4	1	1.25
2-8-1-1	1	1.25
2-10-2-1	1	1.25

Looking at these tables it is possible to see some relationships. For example, the most successful detector is ESP-SYMLINK-OPEN, which triggers when a symbolic link is accessed in a temporary directory. Coupled with the high occurrence of attacks in category 2-12-2-1 (which refers to programs assuming that a file path refers to a valid temporary file), it is an indicator of the high occurrence of the so-called "bad symlink" attacks, in which a program can be tricked into following a symbolic link placed by the attacker to modify or read system files.

Also of interest is the high occurrence of attacks in category 2-2-1-1 (programs assuming that user input is at most of a certain length) that corresponds in general to buffer overflow attacks. These attacks are detected mainly by the ESP-ARGS-LEN, ESP-ENV-LEN, ESP-LONGURL and ESP-FTP-CMD-OVERFLOW detectors.

We implemented detectors for 130 out of 815 entries in the CVE database (see Sections 4.3 and 4.13), corresponding to 15.9% of the entries, both applicable and non-applicable. Those detectors were able to detect 38 of the total 157 entries (both applicable and non-applicable) examined in the detection test, corresponding to a 24.2% detection rate. These numbers are encouraging because we can expect that by implementing detectors for more CVE entries, a larger number of generic detectors could be designed and implemented, providing even larger detection capabilities for new attacks.

Effects of changes on detection capabilities

The changes done to the detectors during this test are listed in Table 5.7 according to the detector they affected. In total, 65 executable statements were added or modified in these changes, and the changes caused an increase of 30% in the detection capabilities of the detectors.

Tables 5.8 and 5.9 list the distribution of detectors and Krsul classifications for the detection capabilities of the final detectors (after the modifications).

Figure 5.8 shows the percentage of attacks in each Krsul category that occurred in the test, and the percentage of those attacks that were detected by the original and the final detectors. Also, to make it easier to see the differences between the original and final detection capabilities by type of vulnerability, Figure 5.9 shows the percentage of attacks detected by the original and final detectors, plotted as x, y coordinates. In this plot, those points farther above the identity line represent the categories with the most improvement as a result of the changes.

One notable change is the increase in the detection of attacks from categories 2-12-2-2 and 2-11-1-1 (both of which correspond to programs assuming that a path name given by the user is in valid space for the application). This can be attributed to the improvements to the ESP-URI-DOTDOT detector, as well as the creation of ESP-FTP-CMD-DOTDOT.

We can also see that there was a 100% detection rate for category 2-12-2-1 (corresponding to “bad symlink” attacks), which shows the effectiveness of the corresponding detector (ESP-SYMLINK-OPEN).

In a similar fashion, Figure 5.10 shows the percentage of attacks detected by each detector before and after the changes, and Figure 5.11 plots the “before” and “after” percentages as x, y coordinates. We can see that ESP-URI-DOTDOT was the detector with the largest improvement. This is mostly the result of the changes in the way encoded characters are examined and to the large occurrence of attacks of type 2-12-2-2 and 2-11-1-1 (which this detector corresponds to). We can also see a considerable improvement in ESP-BADURLS

Table 5.7
Changes made during the detection test of the ESP implementation. Also listed is the ESAM count for each change. Note that the ESP-SNMP-EMPTY-PACKET detector has an ESAM count of zero because no changes were made to the code, it simply was renamed. Detectors marked with a "*" were created as part of the changes.

Detector	Description	ESAM
ESP-URI-DOTDOT	Make it check the whole HTTP request and not only the URI part. Also make it look for DOS-style directory separators (\) in addition to Unix-style (/).	2
ESP-ARGS-LEN	Reduce threshold.	1
ESP-ENV-LEN	Reduce threshold.	1
ESP-BADURLS	Add five new strings for detecting attacks.	5
ESP-BADURLS	Make it unescape the string before processing it. If any escaped characters remain after the first pass, unescape again, to catch both single- and double-encoded malicious characters.	41
*ESP-HTTP-HDR-OVERFLOW	Created. Triggers when the length of a header in an HTTP request exceeds a certain threshold or contains NOP characters.	7
*ESP-FAILED-ROOT-CHOWN	Created. Triggers when a failed attempt to change the ownership of a file to **root** occurs.	2
*ESP-PRIV-ESCALATION	Created. Triggers when "escalation of privilege" occurs, defined as the execution of a non-set-UID program by a **root** set-UID program that has not dropped its privileges.	2
*ESP-FTP-CMD-DOTDOT	Created. Triggers when a command sent to the FTP server uses "`..`" or "`...`" (valid in Windows) in a way that would attempt to access files outside the directory tree of the FTP server.	4
*ESP-SNMP-EMPTY-PACKET	Renamed. This detector already existed as CVE-2000-0221, but during the test it was seen that it is able to detect multiple attacks, so it was renamed as a generic detector.	0

Table 5.8
Distribution of ESP detectors that responded to attacks after the changes made during the detection test. The percentages are with respect to the total number of attacks in the test (80).

Detector	**Attacks detected**	**% of Total**
ESP-SYMLINK-OPEN	13	16.25
ESP-URI-DOTDOT	9	11.25
ESP-BADURLS	6	7.50
ESP-FTP-CMD-DOTDOT	5	6.25
ESP-ARGS-LEN	4	5.00
ESP-BADMODE-ROOT-FILE	4	5.00
ESP-FTP-CMD-OVERFLOW	4	5.00
ESP-ENV-LEN	3	3.75
ESP-FAILED-ROOT-CHOWN	3	3.75
ESP-LONGURL	3	3.75
ESP-PRIV-ESCALATION	2	2.50
ESP-FILE-INTEGRITY	1	1.25
ESP-HTTP-HDR-OVERFLOW	1	1.25
ESP-PORTSCAN	1	1.25
ESP-SNMP-EMPTY-PACKET	1	1.25
ESP-TMP-SYMLINK	1	1.25

Table 5.9
Distribution of categories in the Krsul classification for the attacks detected after the changes. See Appendix C for the definitions of each category. The percentages are with respect to the total number of attacks in the test (80).

Category	**Attacks detected**	**% of Total**
2-12-2-1	14	17.50
2-12-2-2	12	15.00
2-2-1-1	7	8.75
3	6	7.50
2-11-1-1	5	6.25
2-3-2-1	3	3.75
2-5-1-1	3	3.75
2-2-1-3	2	2.50
2-2-1-4	2	2.50
2-1-3-1	1	1.25
2-4-1-1	1	1.25
2-7-1-4	1	1.25
2-8-1-1	1	1.25
2-10-2-1	1	1.25

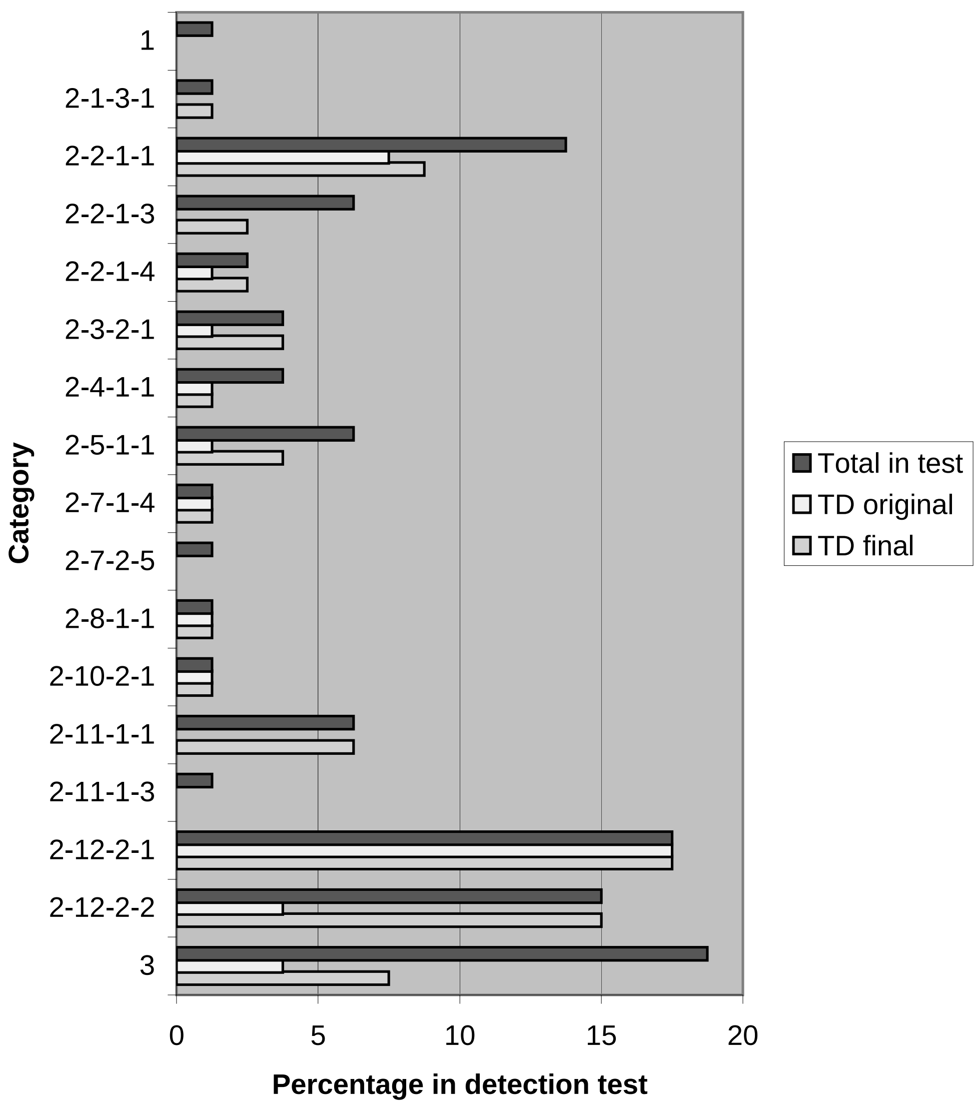


Figure 5.8. Distribution of attacks in the detection test by Krsul categories. The "Total in test" bars represent the percentage of all the applicable attacks in the detection test (80) that belong to each category. The "TD original" bars represents the percentage of attacks that were detected by the original detectors, and the "TD final" bars represent the percentage that were detected by the final detectors.

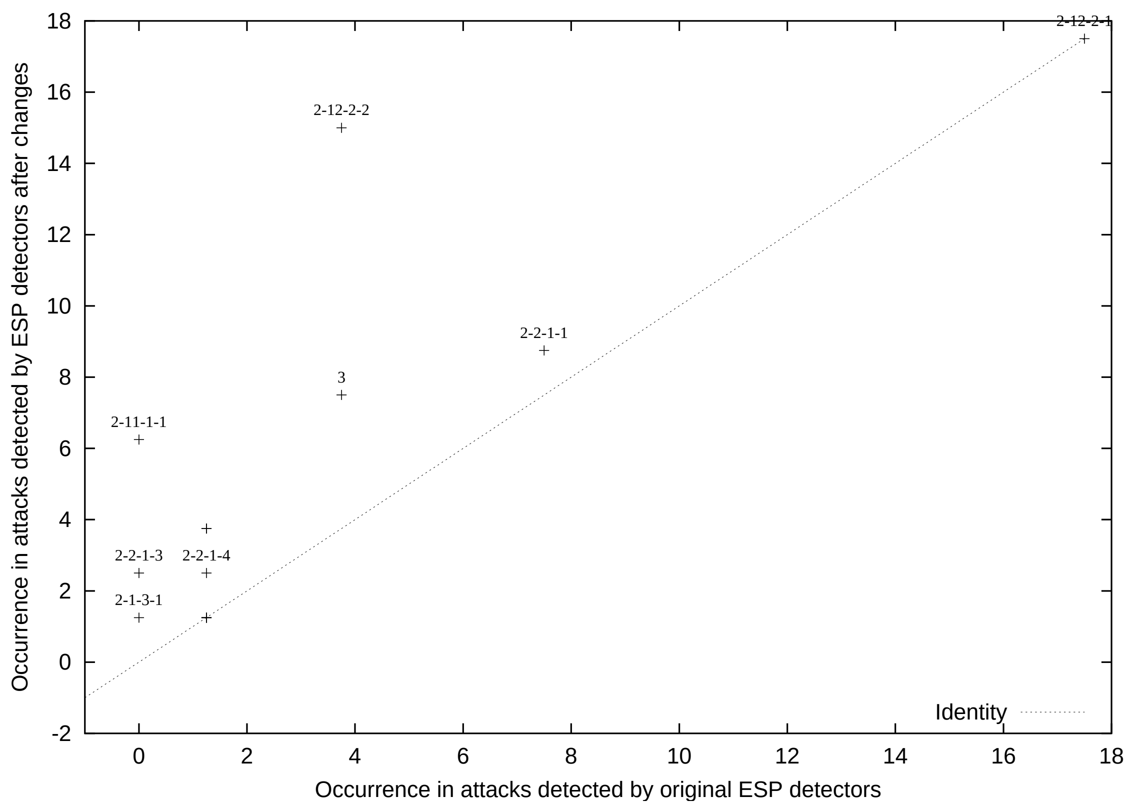


Figure 5.9. Percentages of attacks detected (by Krsul category) by the original and final detectors, plotted as x, y coordinates. Unlabeled points correspond to multiple categories with the same coordinates. Points farther above the identity line represent categories that saw the largest increases in detection as a result of the changes.

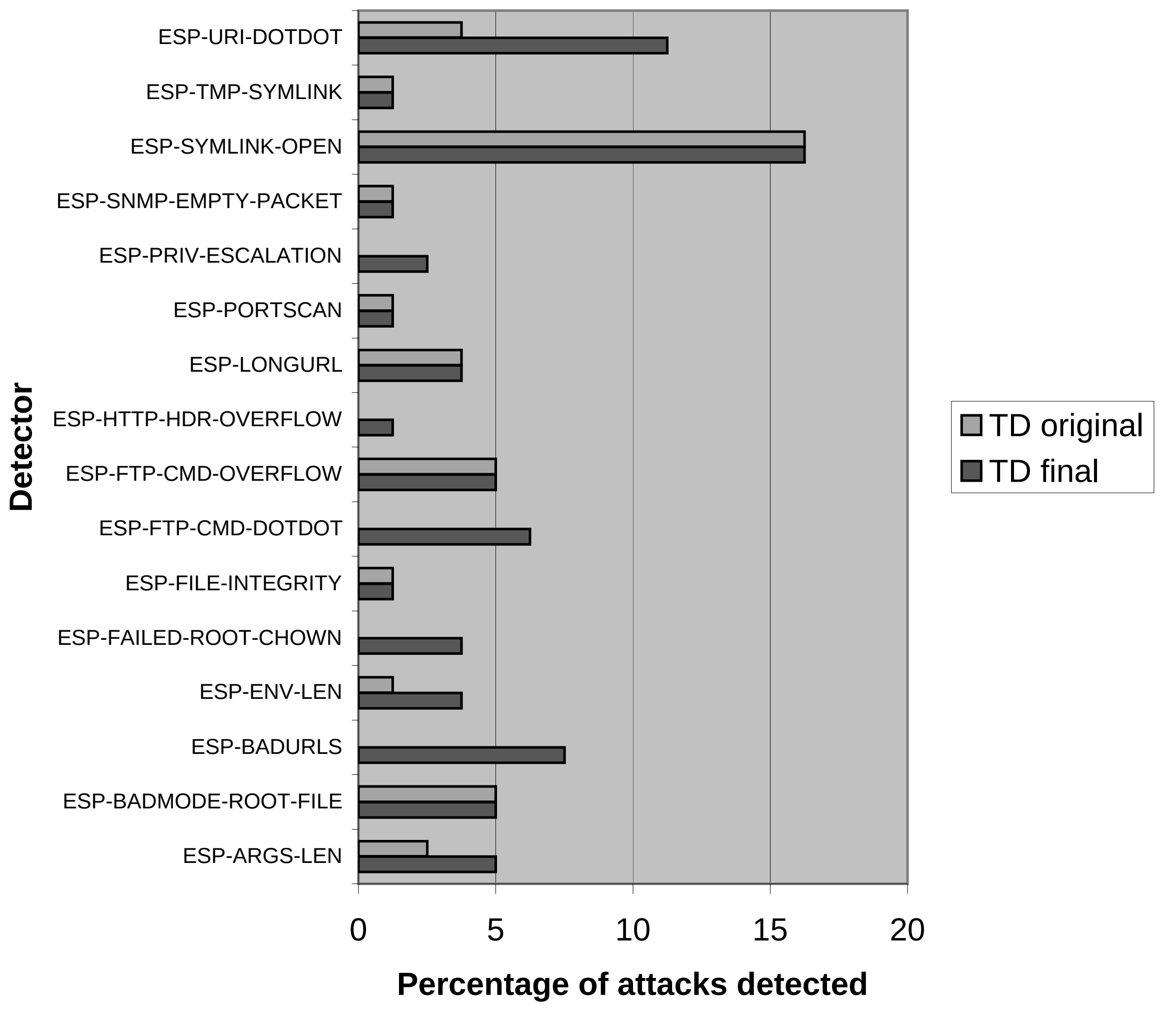


Figure 5.10. Percentage of attacks detected by each detector, before (TD original) and after (TD final) the changes made during the detection test. The percentages are expressed with respect to the total number of applicable attacks in the test (80).

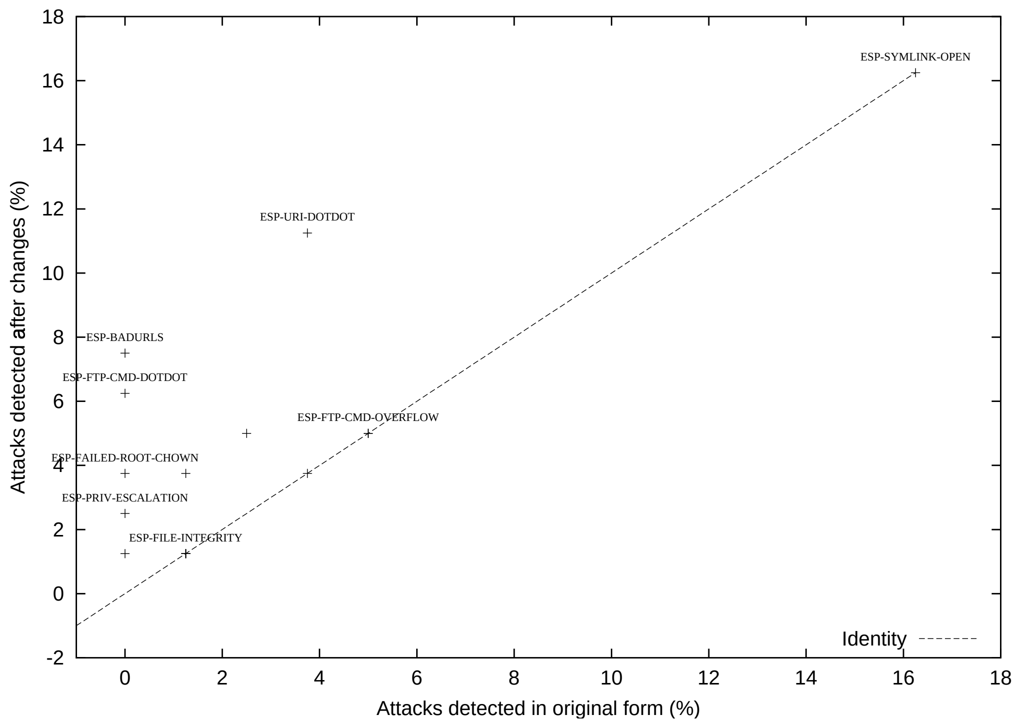


Figure 5.11. Percentage of attacks detected by each detector before and after the changes made during the detection test, plotted as x, y coordinates. Not all the points are labeled because of space considerations. Points farther above the identity line represent the detectors that saw the most improvement after the changes.

(because of the addition of new patterns), and of some of the generic detectors created as part of the changes.

5.2.3 Comments about the detection test

To evaluate the similarity of the attacks used in the detection test to the types of attacks for which the ESP detectors were implemented, we plotted each category using its occurrence in the ESP detectors and in the test set as x, y coordinates, as shown in Figure 5.12. We can see that most points are close to the identity line (shown for reference), indicating that the two distributions are indeed similar. The Pearson correlation (r) for these points is 0.673, confirming the strong linear correlation. A linear regression of the points results in the following formula:

$$y = 1.051x + 0.0694,$$

which is close to the identity line, providing a third confirmation of the similarity between the two distributions.

This similarity suggests the validity of using random drawing from the CVE as a guide for the implementation of the ESP detectors, because it resulted in detectors for a population of attacks similar to those encountered in “the real world.”

Particularly similar (close to the identity line) are categories 2-2-1-1, 2-5-1-1, 2-3-2-1 (all three of which correspond to buffer overflow attacks), 2-2-1-3 and 2-2-1-4 (corresponding to programs failing to check the form of user-provided input). Others, such as 2-12-2-1 (mostly symlink-based attacks) and 2-12-2-2 (mostly “dot-dot” attacks) have a larger representation in the test set than in the detectors implemented, but correspond to some of the most effective generic detectors that were implemented (see Tables 5.5 and 5.8).

Assuming that the set of applicable attacks in the detection test is representative of the new attacks that continuously appear in the real world, we can make some predictions about the detection capabilities of the ESP detectors (this assumption should be evaluated in future work by sampling sets of attacks that appeared during different periods of time).

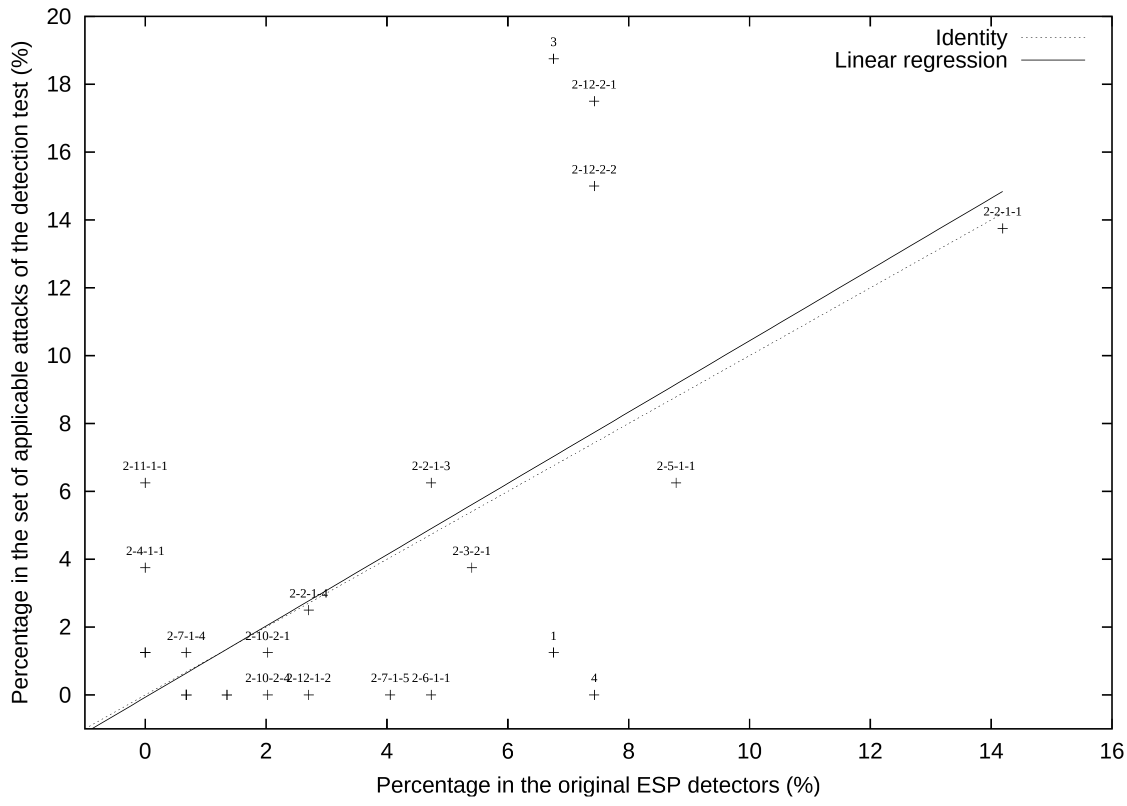


Figure 5.12. Comparison of the distribution of vulnerability types in the ESP detectors and in the set of applicable attacks found during the detection test. Points close to the identity line represent categories that have a similar representation in both distributions. Points without labels correspond to multiple categories with the same percentage in both distributions. Also plotted is the line corresponding to a linear regression of the points. The Pearson correlation of these points is 0.637, indicating a strong linear correlation.

Table 5.10
Confidence intervals for the detection rates of ESP detectors, computed using the percentages of detection before and after the changes made during the detection test. The "original" and "final" designators refer to the ESP detectors as originally implemented (at the start of the test) and after the modifications listed in Table 5.7, respectively.

Measure	Proportion measured in test	95% Conf. interval
D original	0.3500	0.2392–0.4608
TD original	0.4125	0.2984–0.5266
D final	0.6250	0.5127–0.7373
TD final	0.7125	0.6071–0.8179

As a first approach to these predictive capabilities, we can use the standard formula for computing a 95% confidence interval on a proportion p:

$$\text{95\% Confidence interval} = p \pm \left(1.96\sqrt{\frac{p(1-p)}{N}} + \frac{0.5}{N}\right).$$

The 95% confidence intervals for the percentages of detection before and after the changes made to the detectors are shown in Table 5.10. Further possibilities for prediction of detection capabilities are described in Section 6.3.

6. CONCLUSIONS, SUMMARY AND FUTURE WORK

6.1 Conclusions

Throughout their history, intrusion detection systems have been designed with different algorithms and structures for detection. They started by being host-based [40], later evolved into network-based systems [e.g. 60], and in the later years they have tended towards a distributed combination of the two [e.g. 112]. However, during all that evolution, the sources of information used by intrusion detection systems have remained essentially unchanged: audit trails and network traffic. A few notable exceptions have used other sources of information [e.g. 65, 76], but even those were designed with specific applications in mind and using only a limited set of data.

The data sources used by most intrusion detection systems to date have one main limitation: they reflect the behavior of the system being monitored, but are separate from it. Because of this, we consider them as indirect data sources. These data sources have limitations in the timeliness, completeness and accuracy of the data that they provide. They make the intrusion detection system vulnerable to attacks on several fronts, including reliability and denial-of-service. Even when direct data sources have been used by some intrusion detection systems [e.g. 137], the sensors used to collect the information have been external to the objects being monitored; therefore still subject to multiple forms of attack.

The problems and limitations that have been encountered in intrusion detection systems have indicated that the best place to collect data about the behavior of a program or system is at the point where the data is generated or used.

In this dissertation, we have proposed an architecture based on using internal sensors built into the code of the programs that are monitored and are able to extract information from the places in which it is generated or used. Furthermore, by expanding those internal sensors with decision-making logic, we showed an application of embedded detectors to build an intrusion detection system. If necessary, this intrusion detection system can operate without any external components (components that are separate from the ones being monitored), other than those necessary to read the alerts produced.

To demonstrate the feasibility of this architecture and to learn more about its needs and capabilities, we described a specific implementation in the OpenBSD operating system. In its original form, this implementation detects 130 specific attacks, and through the experience acquired with those, 20 generic detectors were implemented that have the capability of detecting previously unknown attacks, as demonstrated by the detection experiments performed.

This dissertation provides an architectural and practical framework in which future study of internal sensors and embedded detectors in intrusion detection can be based. Other research projects are already using this framework for the study of novel techniques for both host-based and distributed intrusion detection [43, 55], and considerable further study on the use of internal sensors is possible.

Both Thesis Hypotheses (Section 1.4) were shown to be true. First, it is possible to build an intrusion detection system using internal sensors while maintaining fidelity, reliability, and reasonable resource usage; although in this last respect, we concluded that the specifics are dependent on the implementation. Second, internal sensors can be used to detect not only known attacks for which they are specifically designed, but also new attacks by looking at generic indicators of malicious activity.

This dissertation also provides a classification of data source types for intrusion detection systems, and a description of the characteristics and types of internal sensors and embedded detectors. Furthermore, its implementation provides specific insight into the types of data that are more useful in the detection of attacks using internal sensors, and into places in a system where those sensors can best be placed to make them efficient and reliable.

Although the implementation cost of the prototype described in this dissertation was high, the advantages of having internal sensors available on a system are numerous. We expect that our work will provide guidance and encouragement for their integration in future systems since their design, which will lead to considerable reductions in their implementation cost and in their impact on performance and size.

In a sense, our approach differs little from the error-checking and careful programming that should exist in any well-maintained system. But the use of code embedded into the operating system and the programs not only for prevention of problems, but for generation of data, is an important step in the development of intrusion detection systems that can provide complete and accurate information about the behavior of a host. The collection of data, even if it is about actions that do not constitute an immediate risk, may provide information about other attacks against which hosts are not protected yet.

6.2 Summary of main contributions

- Provided a classification of data sources for intrusion detection, and of the mechanisms used for accessing them.

- Described the properties of internal sensors in their use for intrusion detection, and an architecture for building intrusion detection systems based on internal sensors and specialized sensors named *embedded detectors*.

- Implemented a prototype intrusion detection system using the architecture described, showing that it is possible to use internal sensors to perform intrusion detection while maintaining most of the desirable properties described in Section 1.2.2.

- Showed that through the implementation of a "large enough" number of specific detectors, it was possible to identify patterns and concepts to be used in generic detectors, confirming the notion that attacks against computer systems tend to cluster around certain vulnerabilities. Once these core problems are identified, it is possible to build generic detectors for them.

- Showed that the prototype implemented is able to detect previously unknown attacks through the use of properly designed generic detectors.

- Showed that it is possible to build a general-purpose intrusion detection system based on obtaining the data needed for the detection, instead of relying on the data provided by the operating system and the applications.
- Showed that it is possible for an intrusion detection system built using internal sensors to have acceptable impact on the host.
- Showed that careless implementation can cause sensors to have a large impact on the host. When using internal sensors, implementation issues are significantly more important than when using traditional external detectors.
- Collected information about the placement and types of data most frequently used by detectors; data therefore most likely to be useful in the development of future detectors and sensors.
- Identified types of vulnerabilities most frequently encountered, both through the development of the prototype and in the detection testing performed.
- Showed that internal sensors and embedded detectors can add significant detection capabilities to a system while increasing its size only by a small fraction.
- By combining concepts from source code instrumentation and intrusion detection, we showed that it is possible to build an intrusion detection system that can perform intrusion detection at multiple levels (operating system, application, network) and in multiple forms (signature-based, anomaly-based) to increase its coverage.

6.3 Future work

The work presented in this dissertation has explored the basic concepts of using internal sensors for intrusion detection by showing their feasibility. However, there is a considerable amount of work that needs to be done to further study and characterize their properties.

This dissertation focuses on the use of internal sensors in a single host. We consider ESP as a distributed intrusion detection architecture because the sensors operate independently in multiple components, but work is needed to show the feasibility and characteristics of using internal sensors in an intrusion detection system that spans multiple hosts.

Some work is already underway [55] to study the mechanisms that could be used in such a system in a way that prevents overloading of both communication channels and coordination components.

The performance tests showed that implementation decisions can severely affect the performance impact of the sensors. In this respect, practical work is necessary to perform optimization and reevaluation of the detectors. Further study is necessary to identify the factors of a detector that result in the largest processing overhead, and in the best ways of reducing those factors.

In terms of the detection capabilities of internal sensors and embedded detectors, the results presented in Section 5.2 show that they have the possibility of detecting a significant percentage of new attacks. Longer-term testing may help in fully understanding and possibly modeling their capabilities and limitations. A formal characterization of the detectors in relationship to the types of attacks encountered would provide firmer prediction capabilities, and help ensure consistency and completeness of the data provided by the detectors. A probabilistic model that links the “ease of detection” of each type of vulnerability with the expected occurrence of each category in new attacks (as implied by the percentages of detection and total occurrence of each category in Figure 5.8) could be useful in predicting the detection capabilities of embedded detectors. Such a model could also be related to the expected effect that improvements to the detectors have in the detection capabilities (as shown in Figure 5.9) to determine cost-effective policies for sensor and detector maintenance and upgrading.

We showed that by implementing detectors for a relatively small fraction of the entries in the CVE database, we were able to implement generic detectors capable of detecting a significant percentage of new attacks. Future work could explore improving the detection rate for new attacks by implementing detectors for a larger number of CVE entries.

Once a host is instrumented with internal sensors, it might be possible to explore new detection capabilities that would have been to expensive or complex to implement using traditional intrusion detection systems. One such possibility is that of *outbound intrusion detection*, in which internal sensors could provide enough information to detect malicious activity at its origin, so that the burden of intrusion detection can be placed not only on the victims, but also on the potential attackers such as the hosts at Internet Service Providers.

Erlingsson and Schneider [47] described the use of *reference monitors* that are automatically generated. In this dissertation, the internal sensors and embedded detectors are individually hand-coded. Using the information gained about the types of data that need to be collected, it may be possible to explore the possibility of automatically generating those sensors in a policy directed fashion. Another possibility would be the automatic generation of components that could be used by programmers to insert sensors and detectors in their code.

As a design decision, during this work we avoided using the embedded detectors to stop an attack once its detected. However, this is a clear application for embedded detectors because of their localization and their ability to perform early detection. Automatic reaction to intrusions has not been widely explored in practice because of the dangers it presents (a false alarm can result in the interruption or modification of legitimate activity), but embedded detectors would be an ideal mechanism for implementing it. The feasibility of this task has been shown in the implementation of the pH system [135], which uses internal sensors to perform both detection and reaction to attacks.

This dissertation has explored the feasibility of extracting information about the behavior of a computer system that is more complete and reliable than any data that had been available before to intrusion detection systems. This availability opens multiple possibilities for future exploration and research, and may lead to the design and development of more efficient, reliable and effective intrusion detection systems.

LIST OF REFERENCES

LIST OF REFERENCES

[1] ACME Labs. http_load. Website at `http://www.acme.com/software/http_load/`, 2001.

[2] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49), 1996. URL `http://www.securityfocus.com/archive/1/5667`.

[3] D. Anderson, T. Frivold, and A. Valdes. Next-generation intrusion-detection expert system (NIDES): A summary. SRI-CSL 95-07, SRI International, Menlo Park, California, May 1995. URL `http://www.sdl.sri.com/nides/reports/4sri.pdf`.

[4] Anzen. Web page at `http://www.anzen.com/products/afj/`, June 2001.

[5] Apache Software Foundation. Apache server. Website at http://www.apache.org/, 2000.

[6] Midori Asaka, Atsushi Taguchi, and Shigeki Goto. The implementation of IDA: An intrusion detection agent system. In *Proceedings of the 11th FIRST Conference*, Brisbane, Australia, June 1999. URL `http://www.ipa.go.jp/STC/IDA/paper/first.ps.gz`.

[7] Stefan Axelsson. Research in intrusion-detection systems: A survey. TR 98-17, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, December 1998. Revised August 19, 1999.

[8] Jai Sundar Balasubramaniyan, Jose Omar Garcia-Fernandez, David Isacoff, Eugene Spafford, and Diego Zamboni. An architecture for intrusion detection using autonomous agents. In *Proceedings of the 14th Annual Computer Security Applications Conference*, pages 13–24. IEEE Computer Society, December 1998.

[9] Bruce Barnett and Dai N. Vu. Vulnerability assessment and intrusion detection with dynamic software agents. In *Proceedings of the Software Technology Conference*, April 1997.

[10] Michael Beck, Harold Bohme, Mirko Dzladzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *Linux Kernel Internals*. Addison-Wesley, Reading, Massachusetts, 1996.

[11] Kirk A. Bradley, Steven Cheung, Nick Puketza, Biswanath Mukherjee, and Ronald A. Olsson. Detecting disruptive routers: A distributed network monitoring approach. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 115–124, Los Alamitos, California, May 1998. IEEE Press.

[12] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0. W3C recommendation, World Wide Web Consortium, October 2000. URL `http://www.w3.org/TR/2000/REC-xml-20001006`.

[13] BugTraq. Mailing list archive. Web page at `http://www.securityfocus.com/`, 1999–2001.

[14] Adriano M. Cansian, Aleck Zander T. de Souza, Sérgio Leugi Filho, and Edson S. Moreira. Um sistema de captura de pacotes para uso en segurança de redes. Available at `http://www.acme-ids.org/downloads/security/papers/apresentacoes/acme2.pdf`, 1999.

[15] Captus Networks. The CaptIO and CaptIO-G security solutions. Web page at `http://www.captusnetworks.com/`, June 2001.

[16] CERT Coordination Center. Denial-of-service attack via ping. CERT Advisory CA-1996-26, Computer Emergency Response Team, December 1996. URL `http://www.cert.org/advisories/CA-1996-26.html`.

[17] CERT Coordination Center. UDP port denial-of-service attack. CERT Advisory CA-1996-01, Computer Emergency Response Team, February 1996. URL `http://www.cert.org/advisories/CA-1996-01.html`.

[18] CERT Coordination Center. IP denial-of-service attacks. CERT Advisory CA-1997-28, Computer Emergency Response Team, December 1998. URL `http://www.cert.org/advisories/CA-1997-28.html`.

[19] CERT Coordination Center. Smurf IP denial-of-service attacks. CERT Advisory CA-1998-01, Computer Emergency Response Team, January 1998. URL `http://www.cert.org/advisories/CA-1998-01.html`.

[20] CERT Coordination Center. CERT/CC statistics. URL `http://www.cert.org/stats/cert_stats.html`, 2001.

[21] Steven Christey, Mann, and Hill. Development of a common vulnerability enumeration. In *Proceedings of the 2nd International Workshop on Recent Advances in Intrusion Detection (RAID99)*, West Lafayette, Indiana, September 1999. Online proceedings, available at `http://www.raid-symposium.org/raid99/`.

[22] Gary G. Christoph, Kathleen A. Jackson, Michael C. Neuman, Christine L. B. Siciliano, Dennis D. Simmonds, Cathy A. Stallings, and Joseph L. Thompson. UNICORN: Misuse detection for UNICOS^{TM}. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*. ACM Press and IEEE Computer Society Press, 1995. URL `http://www.supercomp.org/sc95/proceedings/714_GGC/SC95.HTM`.

[23] Cisco Secure Consulting. Vulnerability statistics report. Available online at `http://www.ieng.com/warp/public/778/security/vuln_stats_02-03-00.html`, 2001.

[24] Cisco Systems. Cisco PIX and CBAC fragmentation attack, September 1998. URL `http://www.cisco.com/warp/public/770/nifrag.shtml`. Field Notice.

[25] Cisco Systems. Cisco Secure Intrusion Detection. Web page at `http://www.cisco.com/warp/public/cc/pd/sqsw/sqidsz/index.shtml`, June 2001.

[26] Computer Associates. eTrust audit. Web page at `http://www3.ca.com/Solutions/Product.asp?ID=157`, June 2001.

[27] Computer Associates. eTrust intrusion detection. Web page at `http://www3.ca.com/Solutions/Product.asp?ID=163`, June 2001.

[28] Computer Incident Advisory Center. LLNL's NID distribution site. Web page at `http://ciac.llnl.gov/cstc/nid/`, June 2001.

[29] Computer Security Technology. The Kane security monitor. Web page at `http://www.cstl.com/html/info/idi/ksm.htm`, June 2001.

[30] Bryan Costales and Eric Allman. *sendmail*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, Massachusetts, second edition, 1997.

[31] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, Washington, DC, August 2001. URL `http://immunix.org/formatguard.pdf`. To be published.

[32] Mark Crosbie, Bryn Dole, Todd Ellis, Ivan Krsul, and Eugene Spafford. IDIOT—users guide. CSD-TR 96-050, COAST Laboratory, Purdue University, 1398 Computer Science Building, West Lafayette, Indiana, September 1996. URL `http://www.cerias.purdue.edu/techreports/public/96-04.ps`.

[33] Mark Crosbie and Eugene Spafford. Defending a computer system using autonomous agents. In *Proceedings of the 18th National Information Systems Security Conference*, volume II, pages 549–558, October 1995. URL `http://www.best.com/~mcrosbie/Research/NISSC95.ps`.

[34] Mark Crosbie and Gene Spafford. Active defense of a computer system using autonomous agents. Technical Report 95-008, COAST Group, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, February 1995. URL `http://www.cerias.purdue.edu/homes/spaf/tech-reps/9508.ps`.

[35] Mark Crosbie and Gene Spafford. Applying genetic programming to intrusion detection. In *Proceedings of the AAAI Fall Symposium on Genetic Programming*. AAAI, 1995. URL `ftp://ftp.cerias.purdue.edu/pub/doc/intrusion_detection/mcrosbie-spaf-AAAI-paper.ps.Z`.

[36] Cylant. Cylantsecure. Web page at `http://www.cylant.com/`, June 2001.

[37] Thomas E. Daniels and Eugene H. Spafford. Identification of host audit data to detect attacks on low-level IP vulnerabilities. *Journal of Computer Security*, 7(1): 3–35, 1999.

[38] DataLynx. DataLynx products page. Web page at `http://www.dlxguard.com/products.htm`, June 2001.

[39] Hervé Debar, Monique Becker, and Didier Siboni. Hyperview: An intelligent security supervisor. In *Proceedings of the 2nd International Conference on Intelligence in Networks*, Bordeaux, France, March 1992.

[40] Dorothy E. Denning. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, 13(2):222–232, February 1987.

[41] Digital Equipment Corporation. POLYCENTER security intrusion detector for SunOS, version 1.0. Online description at `http://www.geek-girl.com/ids/0015.html`, August 1994.

[42] Cheri Dowell and Paul Ramstedt. The ComputerWatch data reduction tool. In *Proceedings of the 13th National Computer Security Conference*, pages 99–108, Washington, DC, October 1990.

[43] James P. Early. An embedded sensor for monitoring file integrity. CERIAS TR 2001-41, CERIAS, Purdue University, West Lafayette, Indiana, March 2001.

[44] En Garde Systems, Inc. T-sight^{TM}: on target security. Web page at `http://www.EnGarde.com/software/t-sight/`, June 2001.

[45] EnteraSys. Dragon intrusion detection solutions. Web page at `http://www.enterasys.com/ids/`, June 2001.

[46] Entercept Security Technologies. Entercept 2.0: Advanced e-Server protection. Web page at `http://www.clicknet.com/products/entercept/`, June 2001.

[47] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *New Security Paradigms Workshop*, pages 87–95, Caledon Hills, Ontario, Canada, September 1999. ACM SIGSAC, ACM Press.

[48] Dan Farmer and Wietse Venema. Computer forensics analysis class handouts. Web page at `http://www.fish.com/forensics/`, August 1999. Accessed in May 2000.

[49] Stephanie Forrest, Steven Hofmeyr, Anil Somayaji, and Thomas Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128. IEEE Computer Press, 1996. URL `ftp://ftp.cs.unm.edu/pub/forrest/ieee-sp-96-unix.ps`.

[50] FreeBSD. Web page at `http://www.freebsd.org/`, July 2001.

[51] D. A. Frincke, D. Tobin, J. C. McConnell, J. Marconi, and D. Polla. A framework for cooperative intrusion detection. In *Proceedings of the 21st National Information Systems Security Conference*, pages 361–373, October 1998.

[52] Fyodor (fyodor@dhp.com). The art of port scanning. Internet `http://www.insecure.org/nmap/nmap_doc.html`, September 1997.

[53] Greg Gillion and Paul E. Proctor. The case for CentraxICE^{TM} hybrid security solution. White paper, CyberSafe Corporation, March 2001. URL `http://www.cybersafe.com/centrax/content/CentraxICE_whitepaper.pdf`.

[54] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, pages 1–13, San Jose, California, July 1996. URL http://www.cs.berkeley.edu:80/~daw/papers/janus-usenix96.ps.

[55] Rajeev Gopalakrishna. A framework for distributed intrusion detection using interest driven cooperating agents. Paper for Qualifier II examination, Department of Computer Sciences, Purdue University, May 2001.

[56] Naji Habra, Baudouin Le Charlier, Aziz Mounji, and Isabelle Mathieu. Preliminary report on Advanced Security Audit Trail Analysis on Unix (ASAX also called SAT-X). Technical report, Institut D'Informatique, FUNDP, rue Grangagnage 21, 5000 Namur, Belgium, September 1994.

[57] Hackerlab. Rx-Posix: : a very fast implementation of the Posix regexp functions. Web page at http://regexps.com/, June 2001.

[58] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell: A Desktop Quick Reference*. O'Reilly, January 2001.

[59] R. Heady, G. Luger, A. Maccabe, and M. Servilla. The Architecture of a Network Level Intrusion Detection System. Technical Report CS90-20, University of New Mexico, Department of Computer Science, August 1990.

[60] L. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A Network Security Monitor. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 296–304, May 1990. URL http://seclab.cs.ucdavis.edu/papers/pdfs/th-gd-90.pdf.

[61] *HP Praesidium Intrusion Detection System/9000 Guide*. Hewlett Packard, 3000 Hanover Street, Palo Alto, California, first edition, July 2000.

[62] Hewlett-Packard. Netperf. Website at http://www.netperf.org/, 2001.

[63] Judith Hochberg, Kathleen Jackson, Cathy Stallings, J. F. McClary, David DuBois, and Josephine Ford. NADIR: An automated system for detecting network intrusion and misuse. *Computers and Security*, 12(3):235–248, May 1993.

[64] Steven A. Hofmeyr and S. Forrest. Architecture for an artificial immune system. *Evolutionary Computation*, 8(4):443–473, Winter 2000. URL http://cs.unm.edu/~forrest/publications/hofmeyr_forrest.ps.

[65] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[66] Steven Andrew Hofmeyr. *An Immunological Model of Distributed Detection and Its Application to Computer Security*. PhD thesis, University of New Mexico, May 1999. URL ftp://coast.cs.purdue.edu/pub/doc/intrusion_detection/hofmeyer-distributed-detection.ps.gz.

[67] Gary Houston. Henry Spencer's regular expression library. Web page at http://arglist.com/regex/, June 2001.

[68] Xie Huagang. Build a secure system with LIDS. Online at http://www.lids.org/document/build_lids-0.2.html, October 2000.

[69] Craig A. Huegen. The latest in denial of service attacks: "smurfing" description and information to minimize effects. URL `http://www.pentics.net/denial-of-service/white-papers/smurf.cgi`. Accessed on January 18, 2001, February 2000.

[70] Koral Ilgun, Richard A. Kemmerer, and Phillip A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[71] Internet Security Systems. RealSecure. Web page at `http://www.iss.net/securing_e-business/security_products/intrusion_detection/`, June 2001.

[72] Capers Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, New York, New York, 1991.

[73] Y. Frank Jou, Fengmin Gong, Chandru Sargor, Shyhtsun Felix Wu, and W. Rance Cleaveland. Architecture design of a scalable intrusion detection system for the emerging network infrastructure. Technical Report CDRL A005, MCNC Information Technologies Division, Research Triangle Park, North Carolina, April 1997.

[74] Richard A. Kemmerer. NSTAT: A model-based real-time network intrusion detection system. Technical Report TRCS97-18, University of California, Santa Barbara, Computer Science, June 17, 1998. URL `ftp://ftp.cs.ucsb.edu/pub/techreports/TRCS97-18.ps`.

[75] Florian Kerschbaum, Eugene H. Spafford, and Diego Zamboni. Using embedded sensors for detecting network attacks. In *Proceedings of the 1st ACM Workshop on Intrusion Detection Systems*. ACM SIGSAC, November 2000.

[76] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29, Fairfax, Virginia, November 1994. ACM Press.

[77] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, Orlando, Florida, December 1994. IEEE Computer Society Press.

[78] Ivan Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, 1998. URL `ftp://coast.cs.purdue.edu/pub/COAST/papers/ivan-krsul/krsul-phd-thesis.ps.Z`.

[79] Sandeep Kumar and Eugene H. Spafford. A pattern matching model for misuse intrusion detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, October 1994. URL `http://www.cerias.purdue.edu/homes/spaf/tech-reps/ncsc.ps`.

[80] Sandeep Kumar and Eugene H. Spafford. A software architecture to support misuse intrusion detection. In *Proceedings of the 18th National Information Systems Security Conference*, pages 194–204. National Institute of Standards and Technology, October 1995.

[81] Benjamin A. Kuperman and Eugene H. Spafford. Generation of application level audit data via library interposition. CERIAS TR 99-11, COAST Laboratory, Purdue University, West Lafayette, Indiana, October 1998. URL `https://www.cerias.purdue.edu/techreports-ssl/public/99-11.ps`.

[82] LANguard Network Security Products. LANguard security event log monitor. Web page at `http://www.languard.com/languard/`, June 2001.

[83] William LeFebvre. *Top: display and update information about the top CPU processes*, 2001. Unix manual pages.

[84] *PRC-PRéCis^{TM} whitepaper*. Litton PRC, 1999. URL `http://www.bellevue.prc.com/precis/solution.pdf`.

[85] Paul Long. Metre v2.3. Software metrics tool available at `http://www.lysator.liu.se/c/metre-v2-3.html`, 2000. Accessed on January 18, 2001.

[86] Teresa F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P. G. Neumann, H. S. Javitz, A. Valdes, and T. D. Garvey. A real-time intrusion detection expert system (IDES) – final technical report. Technical report, SRI Computer Science Laboratory, SRI International, Menlo Park, California, February 1992.

[87] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, Reading, Massachusetts, 1996.

[88] MimeStar Intrusion Detection. SecureNet PRO. Web page at `http://www.mimestar.com/products/`, June 2001.

[89] MITRE. Common vulnerabilities and exposures. Web page at `http://cve.mitre.org/`, 1999–2000.

[90] Abha Moitra. Real-time audit log viewer and analyzer. In *Proceedings of the 4th Workshop on Computer Security Incident Handling*. Forum of Incident Response and Security Teams (FIRST), August 1992.

[91] Paolo Moroni. CERN network security monitor. In *Proceedings of the 1st International Workshop on Recent Advances in Intrusion Detection*, Louvain-la-Neuve, Belgium, September 1998. Online proceedings, available at `http://www.raid-symposium.org/raid98/Prog_RAID98/Table_of_content.html`.

[92] Abdelaziz Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection*. D.Sc. thesis, Facultés Universitaires, Notre-Dame de la Paix, Namur (Belgium), September 1997. URL `ftp://ftp.cerias.purdue.edu/pub/doc/intrusion_detection/mounji_phd_thesis.ps.Z`.

[93] Biswanath Mukherjee, Todd L. Heberlein, and Karl N. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, May/June 1994.

[94] NetBSD. Web page at `http://www.netbsd.org/`, July 2001.

[95] Network ICE Corporation. BlackICE Sentry. Web page at `http://www.networkice.com/products/blackice_sentry.html`, June 2001.

[96] Victoria Neufeldt and David B. Guralnik, editors. *Webster's New World Dictionary of American English*. Simon & Schuster, Inc., third college edition, 1988.

[97] Peter G. Neumann and Phillip A. Porras. Experience with EMERALD to date. In *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, California, April 1999. URL `http://www.sdl.sri.com/emerald/det99.ps.gz`.

[98] Tim Newsham. Format string attacks. Whitepaper, Guardent, 2000. URL `http://julianor.tripod.com/tn-usfs.pdf`.

[99] NFR Security. Overview of NFR network intrusion detection. White paper, June 2001. URL `http://www.nfr.com/products/NID/docs/NID_Technical_Overview.pdf`.

[100] Stephen Northcutt and The Intrusion Detection Team. Intrusion detection: Shadow style. Technical report, SANS Institute, 1998. URL `http://www.docshow.net/ids/shadowstyle.zip`.

[101] Okena. Stormwatch technical white paper. White paper, Okena, 2000. URL `http://www.okena.com/products/literature.htm`.

[102] Michael Okuda and Denise Okuda. *The Star Trek Encyclopedia:A Reference Guide to the Future*. Simon and Shuster Incorporated, August 1999.

[103] OpenBSD. Web page at `http://www.openbsd.org/`, July 2001.

[104] OpenBSD. The ports and packages collection. Web page at `http://www.openbsd.org/ports.html`, June 2001.

[105] Openwall Project. Linux kernel patch from the Openwall project. Web page at `http://www.openwall.com/linux/`, June 2001.

[106] Packet Storm. Web page at `http://packetstorm.securify.com`, 2000.

[107] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th Annual USENIX Security Symposium*, San Antonio, Texas, January 1998. URL `ftp://ftp.ee.lbl.gov/papers/bro-usenix98-revised.ps.Z`.

[108] Wendy W. Peng and Dolores R. Wallace. Software error analysis. NIST Special Publication 500-209, National Institute of Standards and Technology, Gaithersburg, Maryland, March 1993. URL `http://hissa.nist.gov/HHRFdata/Artifacts/ITLdoc/209/error.htm`.

[109] Charles Pfleeger. *Security in Computing*. Prentice Hall, second edition, 1997.

[110] PGP Security. Cybercop monitor. Web page at `http://www.pgp.com/products/cybercop-monitor/default.asp`, May 2001.

[111] Phil Porras, Dan Schnackenberg, Stuart Staniford-Chen, Maureen Stillman, and Felix Wu. The common intrusion detection framework architecture. Web page at `http://www.gidos.org/drafts/architecture.txt`, May 2001.

[112] Phillip A. Porras and Peter G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, pages 353–365. National Institute of Standards and Technology, October 1997.

[113] Richard Power. 1999 CSI/FBI computer crime and security survey. *Computer Security Journal*, Volume XV(2), 1999.

[114] Katherine E. Price. Host-based misuse detection and conventional operating systems' audit data collection. Master's thesis, Purdue University, December 1997. URL `http://www.cerias.purdue.edu/techreports/public/97-15.ps`.

[115] Paul Proctor. Computer misuse detection system (CMDS^{TM}) concepts. In *SAIC Science and Technology Trends I*, pages 137–145. SAIC, December 1996. URL `http://cp-its-web04.saic.com/satt.nsf/1author?OpenView`.

[116] Psionic Software. The Abacus project. Web page at `http://www.psionic.com/abacus/`, June 2001.

[117] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., January 1998.

[118] Andrew Rathmell and Lorenzo Valeri. Information warfare and the asymmetric threat: An approach to early warning. Technical report, International Center for Security Analysis, 1997. URL `http://www.icsa.ac.uk/Publications/asymmetric-nf.htm`.

[119] Recourse Technologies. ManHunt. Web page at `http://www.recourse.com/products/manhunt/hunt.html`, June 2001.

[120] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, January 1997.

[121] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the LISA'99 conference*. USENIX, November 1999. URL `http://www.snort.org/lisapaper.txt`.

[122] RootShell. Web page at `http://www.rootshell.com`, 2000.

[123] Ryan Net Works, LLC. CyberTrace intrusion detection system. Web page at `http://www.cybertrace.com/ctids.html`, June 2001.

[124] Sanctum. AppShield: Automated web application control and security. Web page at `http://www.sanctuminc.com/solutions/appshield/index.html`, June 2001.

[125] *Scoadmin: invoke SCOadmin applications or configure SCOadmin hierarchy*. Santa Cruz Operation, 2001. SCO Unixware manual page.

[126] Christoph L. Schuba, Ivan V. Krsul, Markus G. Kuhn, Eugene H. Spafford, Aurobindo Sundaram, and Diego Zamboni. Analysis of a denial of service attack on TCP. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 208–223. IEEE Computer Society Press, May 1997.

[127] Scut and Team Teso. Exploiting format string vulnerabilities. Online document at `http://julianor.tripod.com/teso-fs1-1.pdf`, March 2001.

[128] M. Sebring, E. Shellhouse, M. Hanna, and R. Whitehurst. Expert Systems in Intrusion Detection: A Case Study. In *Proceedings of the 11th National Computer Security Conference*, October 1988.

[129] Secure Worx. Defense Worx network intrusion detection system. Web page at `http://www.secure-worx.com/products/network_ids.html`, June 2001.

[130] SecurityFocus. Web page at `http://www.securityfocus.com/`, 1999–2000.

[131] Robert S. Sielken. Application intrusion detection. Technical Report CS-99-17, Department of Computer Science, University of Virginia, June 1999. URL `ftp://ftp.cs.virginia.edu/pub/techreports/CS-99-17.ps.Z`.

[132] Stephen Smaha. Haystack: An intrusion detection system. In *Proceedings of the 4th Aerospace Computer Security Applications Conference*, pages 37–44, December 1988.

[133] Steven R. Snapp, S. Smaha, D. M. Teal, and T. Grance. The DIDS (Distributed Intrusion Detection System) Prototype. In *Proceedings of the USENIX Summer 1992 Technical Conference*, pages 100–108, San Antonio, Texas, June 1992.

[134] Michael Sobirey. The intrusion detection system AID. Web page at `http://www-rnks.informatik.tu-cottbus.de/~sobirey/aid.e.html`, June 2001.

[135] Anil Somayaji and Stephanie Forrest. Automated response using system-call delays. In *Proceedings of the 9th USENIX Security Symposium*, August 2000. URL `http://cs.unm.edu/~forrest/publications/uss-2000.ps`.

[136] SourceFire Inc. OpenSnort Sensor. Web page at `http://www.sourcefire.com/html/products.html`, June 2001.

[137] Eugene H. Spafford and Diego Zamboni. Intrusion detection using autonomous agents. *Computer Networks*, 34(4):547–570, October 2000. URL `http://www.elsevier.nl/gej-ng/10/15/22/49/30/25/article.pdf`.

[138] Lance Spitzner. Intrusion detection for FW-1: How to know when you are being probed, November 2000. URL `http://www.enteract.com/~lspitz/intrusion.html`.

[139] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS: A graph based intrusion detection system for large networks. In *Proceedings of the 19th National Information Systems Security Conference*, volume 1, pages 361–370. National Institute of Standards and Technology, October 1996.

[140] W. Richard Stevens. *TCP/IP Illustrated*, volume 2. Addison-Wesley, 1994.

[141] *Crontab: user crontab file*. Sun Microsystems, 2001. SunOS 5.7 manual page.

[142] *Df: report filesystem disk space usage*. Sun Microsystems, 2001. SunOS 5.7 manual page.

[143] *Inetd: Internet services daemon*. Sun Microsystems, 2001. SunOS 5.7 manual page.

[144] *Netstat: show network status*. Sun Microsystems, 2001. SunOS 5.7 manual page.

[145] *Ping: send ICMP ECHO_REQUEST packets to network hosts*. Sun Microsystems, 2001. SunOS 5.7 manual page.

[146] *Ps: report process status*. Sun Microsystems, 2001. SunOS 5.7 manual page.

[147] Kymie M. C. Tan, David Thompson, and A. B. Ruighaver. Intrusion detection systems and a view to its forensic applications. Technical report, Department of Computer Science, University of Melbourne, Parkville 3052, Australia, year of publication unkown. URL `http://www.securityfocus.com/data/library/idsforensics.ps`.

[148] AXENT Technologies. AXENT technologies' NetProwler^{TM} and Intruder Alert^{TM}. White paper, Hurwitz Group, September 2000. URL `http://www.safecomms.com/pdf/symantec/ita_hurwitzreport_wp.pdf`.

[149] Touch Technologies, Inc. INTOUCH INSA - network security agent. Web page at `http://www.ttisms.com/tti/nsa_www.html`, June 2001.

[150] H. S. Vaccaro and G. E. Liepins. Detection of anomalous computer session activity. In *Proceedings of the 1989 IEEE Symposium on Research in Security and Privacy*, pages 280–289, 1989.

[151] Wietse Venema. TCP WRAPPER: Network monitoring, access control and booby traps. In USENIX Association, editor, *Proceedings of the 3rd UNIX Security Symposium*, pages 85–92, Berkeley, California, September 1992. USENIX.

[152] G. Vigna and R.A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999. URL `http://www.cs.ucsb.edu/~rsg/pub/1999_vigna_kemmerer_jcs99.ps.gz`.

[153] Gregory B. White, Eric A. Fisch, and Udo W. Pooch. Cooperating security managers: A peer-based intrusion detection system. *IEEE Network*, pages 20–23, January/February 1996.

[154] Scott M. Wimer. Cylantsecure^{TM}: A scientific approach to security. Whitepaper, Cylant Technology, Inc., 2001. URL `http://www.cylant.com/whitepapers/cs-scientific.shtml`.

[155] X-Force. Web page at `http://xforce.iss.net`, 2000.

[156] Jiahai Yang, Peng Ning, X. Sean Wang, and Sushil Jajodia. CARDS: A distributed system for detecting coordinated attacks. In *Proceedings of IFIP TC11 16th Annual Working Conference on Information Security*, pages 171–180, August 2000. URL `http://ise.gmu.edu/~pning/sec2000.ps`.

APPENDICES

Appendix A: Detectors and Sensors Implemented

The tables in this appendix list all the detectors and sensors that were implemented in the original implementation phase of the ESP project (before the modifications that resulted from the testing phase described in Section 5.2).

The columns in these tables have the following meanings:

N: Reference number of the detector, starting with an "S" for specific detectors and with a "G" for generic detectors.

ID: Identifier of the detector.

Vuln: For specific detectors, vulnerable operating system or program.

S: Whether the detector is stateful: "Yes" (Stateful) or "No" (Stateless).

Src: Types of data sources used: "Net" (Network data), "SysState" (System state), "User" (User-provided data), "FileSys" (File system state), "App" (Application data), "Prog" (Program state) and "SysInfo" (System information).

Dir: Implementation source directory.

Class: Corresponding category in the taxonomy proposed by Krsul [78] (see Appendix C).

Det: Reference number of detectors that cover the current one, if any.

Imp: Reference number of detectors that implement the current one, if any.

E: ESAM (Executable Statements Added or Modified) metric for the detector.

B: BOCAM (Blocks of Code Added or Modified) metric for the detector.

Not all fields apply to all the detectors. When not applicable, a field is left empty or with the string "n/a".

Table A.1: List of specific detectors implemented.

N	ID	Vuln	S	Src	Dir	Class	Det	Imp	E	B
S1	CVE-1999-0013	(m)ssh	No	FileSys	`ssh`	2-12-2-2	G18, G21		2	1
	Stolen credentials from SSH clients via ssh-agent program, allowing other local users to access remote accounts belonging to the ssh-agent user.									
S2	CVE-1999-0016	(m)IP	No	Net	`netinet`	2-10-1-1	G20		2	1
	Land IP denial of service.									
S3	CVE-1999-0022	(m)rdist	No			2-5-1-1	G1			
	Local user gains root privileges via buffer overflow in rdist, via expstr() function.									
S4	CVE-1999-0026	Irix	No			2-5-1-1	G1			
	Root privileges via buffer overflow in pset command on SGI IRIX systems.									
S5	CVE-1999-0027	Irix	No	User	`mt`	2-5-1-1	G1		2	1
	Root privileges via buffer overflow in eject command on SGI IRIX systems.									
S6	CVE-1999-0032	(m)lpr	No	User	`lpr`	2-5-1-1	G1		2	1
	Buffer overflow in BSD-based lpr package allows local users to gain root privileges.									
S7	CVE-1999-0039	Irix	No		`httpd`	2-9-1-3		G3		
	Arbitrary command execution using webdist CGI program in IRIX.									
S8	CVE-1999-0046	(m)rlogind	Yes	Net	`rlogind`	2-3-2-1	G6		5	2
	Buffer overflow of rlogin program using TERM environmental variable.									
S9	CVE-1999-0047	(m)sendmail	Yes	Prog, User	`sendmail`	3			10	7
	Buffer overflow vulnerability in sendmail 8.8.3/8.8.4.									
S10	CVE-1999-0049	Irix	No			2-12-2-1	G19, G21			
	Csetup under IRIX allows arbitrary file creation or overwriting.									
S11	CVE-1999-0052	(m)IP	No	Net	`netinet`	2-10-2-2			2	1
	IP fragmentation denial of service in FreeBSD allows a remote attacker to cause a crash.									
S12	CVE-1999-0053	FreeBSD	No	Net, SysState	`netinet`	2-10-4-1	G20		2	1
	TCP RST denial of service in FreeBSD.									

(continued on next page)

Table A.1: List of specific detectors implemented (continued).

N	ID	Vuln	S	Src	Dir	Class	Det	Imp	E	B
S13	CVE-1999-0057	(m)vacation	No	User	`vacation`	2-2-1-3			2	1
	Multiple vendor vacation(1) vulnerability.									
S14	CVE-1999-0068	(m)PHP	No		`httpd`	2-12-1-1		G3		
	CGI PHP mylog script allows an attacker to read any file on the target server.									
S15	CVE-1999-0070	(m)test-cgi	No		`httpd`	2-2-1-4		G3		
	Test-cgi program allows an attacker to list files on the server.									
S16	CVE-1999-0071	(m)httpd	No	Net	`httpd`	2-6-1-1			2	1
	Apache httpd cookie buffer overflow for versions 1.1.1 and earlier.									
S17	CVE-1999-0074	(m)TCP	Yes	Net	`netinet`	1		G13	12	3
	Listening TCP ports are sequentially allocated, allowing spoofing attacks.									
S18	CVE-1999-0077	(m)TCP	No	Net, SysState	`netinet`	1			2	1
	Predictable TCP sequence numbers allow spoofing.									
S19	CVE-1999-0093	AIX	No	FileSys, SysState	`nslookup`	2-12-1-2			3	1
	AIX nslookup command allows local users to obtain root access by not dropping privileges correctly.									
S20	CVE-1999-0095	(m)sendmail	No	User	`sendmail`	4			1	1
	Debug command in sendmail.									
S21	CVE-1999-0096	(m)sendmail	No	User	`sendmail`	4			6	2
	Sendmail decode aliases can be used to overwrite files.									
S22	CVE-1999-0103	(m)inetd	No	Net	`inetd`	4			8	4
	Echo and chargen, or other combinations of UDP services, can be used in tandem to flood the server, a.k.a. UDP bomb or UDP packet storm.									
S23	CVE-1999-0108	Irix	No			2-5-1-1	G1			
	The printers program in IRIX has a buffer overflow that gives root access to local users.									

(continued on next page)

Table A.1: List of specific detectors implemented (continued).

N	ID	Vuln	S	Src	Dir	Class	Det	Imp	E	B
S24	CVE-1999-0116	(m)TCP	No	SysState	`netinet`	1			2	1
	Denial of service when an attacker sends many SYN packets to create multiple connections without ever sending an ACK to complete the connection, aka SYN flood.									
S25	CVE-1999-0128	(m)ICMP	No	Net	`netinet`	2-10-2-1	G11		3	1
	Oversized ICMP ping packets can result in a denial of service, aka Ping of Death.									
S26	CVE-1999-0129	(m)sendmail	No	FileSys	`sendmail`	2-12-1-2			2	1
	Sendmail group permissions vulnerability.									
S27	CVE-1999-0130	(m)sendmail	Yes	SysState	`sendmail`	2-7-2-3			3	3
	Sendmail Daemon Mode vulnerability.									
S28	CVE-1999-0131	(m)sendmail	No	Prog	`sendmail`	2-6-1-1	G10		1	1
	Sendmail GECOS buffer overflow and resource starvation.									
S29	CVE-1999-0132	(m)vi	No			2-12-2-1	G19, G21		1	
	Expreserve, used in vi and ex, allows local users to overwrite arbitrary files and gain root access.									
S30	CVE-1999-0135	Solaris	No			2-12-2-1	G19, G21			
	Admintool in Solaris allows a local user to write to arbitrary files and gain root access.									
S31	CVE-1999-0137	Linux	No	User	`dip`	2-5-1-1	G1		2	1
	The dip program on many Linux systems allows local users to gain root access via a buffer overflow.									
S32	CVE-1999-0153	Windows	No	Net	`netinet`	unknown			3	1
	Windows 95/NT out of band (OOB) data denial of service through NETBIOS port, aka WinNuke.									
S33	CVE-1999-0157	Cisco PIX	No		`netinet`	2-10-2-2	S11	S11	2	1
	Cisco PIX firewall and CBAC IP fragmentation attack results in a denial of service.									
S34	CVE-1999-0158	Windows	No			2-2-1-3, 2-12-2-2	G22			
	Cisco PIX firewall manager (PFM) on Windows NT allows attackers to connect to port 8080 on the PFM server and retrieve any file whose name and location is known.									
S35	CVE-1999-0164	Solaris	No			2-7-1-4	G17, G16, G21			
	A race condition in the Solaris ps command allows an attacker to overwrite critical files.									

(continued on next page)

Table A.1: List of specific detectors implemented (continued).

N	ID	Vuln	S	Src	Dir	Class	Det	Imp	E	B
S36	CVE-1999-0177	Windows	No		`httpd`	2-12-1-2		G3		
	The uploader program in the WebSite web server allows a remote attacker to execute arbitrary programs.									
S37	CVE-1999-0183	Linux	No	User	`tftpd`	2-12-2-2			2	1
	Linux implementations of TFTP would allow access to files outside the restricted directory.									
S38	CVE-1999-0204	(m)sendmail	Yes	Net	`sendmail`	2-4-1-4			4	2
	Identd attack.									
S39	CVE-1999-0206	(m)sendmail	Yes	Prog, User	`sendmail`	3			9	8
	MIME buffer overflow in sendmail 8.8.0 and 8.8.1.									
S40	CVE-1999-0214	SunOS	No	Net, SysState	`netinet`	unknown			2	1
	Denial of service by sending forged ICMP unreachable packets.									
S41	CVE-1999-0219	Windows	No			2-2-1-1	G8		2	1
	Buffer overflow in Serv-U FTP server when user performs a cwd to a directory with a long name.									
S42	CVE-1999-0239	(m)FastTrack Web Server	No		`httpd`	2-2-1-3		G3		
	Netscape FastTrack Web server lists files when a lowercase get command is used instead of an uppercase GET.									
S43	CVE-1999-0259	(m)cfingerd	No	User	`fingerd`	2-2-1-3			2	1
	Cfingerd lists all users on a system via search.**@target.									
S44	CVE-1999-0260	(m)jj	No		`httpd`	2-2-1-4		G3		
	The jj CGI program allows command execution via shell metacharacters.									
S45	CVE-1999-0265	(m)ICMP	No	Net	`netinet`	unknown			8	1
	ICMP redirect messages may crash or lock up a host.									
S46	CVE-1999-0267	(m)NCSA httpd	No	App	`httpd`	2-2-1-1		G3	2	1
	Buffer overflow in NCSA HTTP daemon v1.3 allows remote command execution.									

(continued on next page)

Table A.1: List of specific detectors implemented (continued).

N	ID	Vuln	S	Src	Dir	Class	Det	Imp	E	B
S47	CVE-1999-0273	Solaris	Yes	Net	`telnetd`	unknown			6	2
	Denial of service through Solaris 2.5.1 telnet by sending D̂ characters.									
S48	CVE-1999-0281	Windows	No	Net	`httpd`	2-2-1-1	G12		2	1
	Denial of service in IIS using long URLs.									
S49	CVE-1999-0299	FreeBSD	No	Net	`net`	2-6-1-1	G9		1	1
	Buffer overflow in FreeBSD lpd through long DNS hostnames.									
S50	CVE-1999-0304	(m)kernel	No			3	S51			
	Mmap function in BSD allows local attackers in the kmem group to modify memory through devices.									
S51	CVE-1999-0323	(m)kernel	No	SysState	`vm`	3			2	2
	FreeBSD mmap function allows users to modify append-only or immutable files.									
S52	CVE-1999-0324	HP-UX	No			2-7-1-5	G19, G21			
	Ppl program in HP-UX allows local users to create root files through symlinks.									
S53	CVE-1999-0340	Linux	No	User	`cron`	2-3-2-1	G6, G2		1	1
	Buffer overflow in Linux Slackware crond program allows local users to gain root access.									
S54	CVE-1999-0373	(m)Super	No			2-3-2-1, 2-5-1-1	G6, G1			
	Buffer overflow in the Super utility in Debian Linux and other operating systems allows local users to execute commands as root.									
S55	CVE-1999-0377	(m)inetd	No			1	S77			
	Process table attack in Unix systems allows a remote attacker to perform a denial of service by filling a machine's process tables through multiple connections to network services.									
S56	CVE-1999-0386	Windows	No	Net	`httpd`	2-2-1-3		G3		
	Microsoft Personal Web Server and FrontPage Personal Web Server in some Windows systems allows a remote attacker to read files on the server by using a nonstandard URL.									
S57	CVE-1999-0396	(m)TCP	Yes	SysState	`netinet`	2-4-2-1		G13	3	2
	A race condition between the select() and accept() calls in NetBSD TCP servers allows remote attackers to cause a denial of service.									

(continued on next page)

Table A.1: List of specific detectors implemented (continued).

N	ID	Vuln	S	Src	Dir	Class	Det	Imp	E	B
S58	CVE-1999-0404	Windows	No			2-2-1-1	G15			
	Buffer overflow in the Mail-Max SMTP server for Windows systems allows remote command execution.									
S59	CVE-1999-0414	Linux	No	Net	`netinet`	2-10-2-3			3	1
	In Linux before version 2.0.36, remote attackers can spoof a TCP connection and pass data to the application layer before fully establishing the connection.									
S60	CVE-1999-0439	(m)procmail	No	Net, User	`procmail`	2-7-2-1			1	1
	Buffer overflow in procmail before version 3.12 allows remote or local attackers to execute commands via expansions in the procmailrc configuration file.									
S61	CVE-1999-0442	Solaris	No			2-1-4-1, 4	G2, G7			
	Solaris ff.core allows local users to modify files.									
S62	CVE-1999-0471	Windows	No		`httpd`	3		G3		
	The remote proxy server in Winroute allows a remote attacker to reconfigure the proxy without authentication through the cancel button.									
S63	CVE-1999-0474	Windows	No			2-12-2-2	G22			
	The ICQ Webserver allows remote attackers to use .. to access arbitrary files outside of the user's personal directory.									
S64	CVE-1999-0478	(m)sendmail	No	User	`sendmail`	2-2-1-1			1	1
	Denial-of-Service attack using excessively long headers, or a very large number of them.									
S65	CVE-1999-0484	OpenBSD	No	User	`ping`	3			2	1
	Buffer overflow in OpenBSD ping.									
S66	CVE-1999-0513	(m)ICMP	Yes	Net, SysState	`netinet`	4, 1			22	5
	ICMP messages to broadcast addresses are allowed, allowing for a Smurf attack that can cause a denial of service.									

(continued on next page)

Table A.1: List of specific detectors implemented (continued).

N	ID	Vuln	S	Src	Dir	Class	Det	Imp	E	B
S67	CVE-1999-0514	(m)UDP	Yes	Net, SysState	`netinet`	4, 1			12	5
	UDP messages to broadcast addresses are allowed, allowing for a Fraggle attack that can cause a denial of service by flooding the target.									
S68	CVE-1999-0676	Solaris	No			2-12-2-1	G19, G21			
	Sdtcm_convert in Solaris 2.6 allows a local user to overwrite sensitive files via a symlink attack.									
S69	CVE-1999-0693	(m)CDE	No			2-3-2-1	G6			
	Buffer overflow in TT_SESSION environment variable in ToolTalk shared library allows local users to gain root privileges.									
S70	CVE-1999-0703	(m)kernel	No	FileSys	`kern`	1			2	2
	OpenBSD, BSDI, and other Unix operating systems allow users to set chflags and fchflags on character and block devices.									
S71	CVE-1999-0704	(m)amd	No	App	`amd`	2-2-1-1			4	2
	Buffer overflow in Berkeley automounter daemon (amd) logging facility provided in the Linux am-utils package and others.									
S72	CVE-1999-0708	(m)cfingerd	No			2-6-1-1	G10			
	Buffer overflow in cfingerd allows local users to gain root privileges via a long GECOS field.									
S73	CVE-1999-0710	(m)squid	No		`httpd`	4		G3		
	The RedHat squid program installs cachemgr.cgi in a public web directory, allowing remote attackers to use it as an intermediary to connect to other systems.									
S74	CVE-1999-0731	(m)KDE	Yes	Prog	`kde`	3			3	3
	The KDE klock program allows local users to unlock a session using malformed input.									
S75	CVE-1999-0735	(m)KDE	No			2-12-2-1	G19, G21			
	KDE K-Mail allows local users to gain privileges via a symlink attack in temporary user directories.									

(continued on next page)

Table A.1: List of specific detectors implemented (continued).

N	ID	Vuln	S	Src	Dir	Class	Det	Imp	E	B
S76	CVE-1999-0744	(m)FastTrack Web Server	No			2-2-1-1	G12			
	Buffer overflow in Netscape Enterprise Server and FastTrack Server allows remote attackers to gain privileges via a long HTTP GET request.									
S77	CVE-1999-0746	(m)inetd	No	Net, Prog	`inetd`	1, 4			4	1
	A default configuration of in.identd in SuSE Linux waits 120 seconds between requests, allowing a remote attacker to conduct a denial of service.									
S78	CVE-1999-0761	FreeBSD	No	FileSys	`gen`	2-7-1-6			2	2
	Buffer overflow in FreeBSD fts library routines allows local user to modify arbitrary files via the periodic program.									
S79	CVE-1999-0763	NetBSD	No	Net, SysState	`netinet`	3			1	1
	NetBSD on a multi-homed host allows ARP packets on one network to modify ARP entries on another connected network.									
S80	CVE-1999-0764	NetBSD	No	Net, SysState	`netinet`	3			1	1
	NetBSD allows ARP packets to overwrite static ARP entries.									
S81	CVE-1999-0769	(m)crond	No	User	`cron`	2-3-2-3			6	1
	Modification of sendmail arguments using MAILTO in a crontab file.									
S82	CVE-1999-0771	Windows	No			2-12-2-2	G22			
	The web components of Compaq Management Agents and the Compaq Survey Utility allow a remote attacker to read arbitrary files via a .. (dot dot) attack.									
S83	CVE-1999-0773	Solaris	No			2-5-1-1	G1			
	Buffer overflow in Solaris lpset program allows local users to gain root access.									
S84	CVE-1999-0789	AIX	No			2-2-1-1	G8			
	Buffer overflow in AIX ftpd in the libc library.									

(continued on next page)

Table A.1: List of specific detectors implemented (continued).

N	ID	Vuln	S	Src	Dir	Class	Det	Imp	E	B
S85	CVE-1999-0806	Solaris	No			2-5-1-1	G1			
	Buffer overflow in Solaris dtprintinfo program.									
S86	CVE-1999-0834	(m)rsaref	No	User	`rsaref`	2-2-1-1			8	8
	Buffer overflow in RSAREF2 via the encryption and decryption functions in the RSAREF library.									
S87	CVE-1999-0842	Windows	No			2-12-2-2	G22			
	Symantec Mail-Gear 1.0 web interface server allows remote users to read arbitrary files via a .. (dot dot) attack.									
S88	CVE-1999-0859	Solaris	No	FileSys	`arp`	2-12-1-1			1	1
	Solaris arp allows local users to read files via the -f parameter, which lists lines in the file that do not parse properly.									
S89	CVE-1999-0865	Windows	No		`httpd`	2-2-1-1	G12		2	1
	Buffer overflow in CommuniGatePro via a long string to the HTTP configuration port.									
S90	CVE-1999-0893	SCO OpenServer	No			2-12-2-1	G19, G21			
	UserOsa in SCO OpenServer allows local users to corrupt files via a symlink attack.									
S91	CVE-1999-0896	(m)RealServer	No		`httpd`	2-2-1-1			2	1
	Buffer overflow in RealNetworks RealServer administration utility allows remote attackers to execute arbitrary commands via a long username and password.									
S92	CVE-1999-0931	Windows	No		`httpd`	2-2-1-1	G12		2	1
	Buffer overflow in Mediahouse Statistics Server allows remote attackers to execute commands.									
S93	CVE-1999-0953	(m)WWWBoard	No		`httpd`	4		G3		
	WWWBoard stores encrypted passwords in a password file that is under the web root and thus accessible by remote attackers.									
S94	CVE-1999-0959	Irix	No			2-12-2-1	G19, G21			
	IRIX startmidi program allows local users to modify arbitrary files via a symlink attack.									

(continued on next page)

Table A.1: List of specific detectors implemented (continued).

N	ID	Vuln	S	Src	Dir	Class	Det	Imp	E	B
S95	CVE-1999-0969	Windows	No	Net	`netinet`	unknown			4	1
	The Windows NT RPC service allows remote attackers to conduct a denial of service using spoofed malformed RPC packets which generate an error message that is sent to the spoofed host, potentially setting up a loop, aka Snork.									
S96	CVE-1999-0976	(m)sendmail	No	App, SysState	`sendmail`	2-12-1-2			1	1
	Senmdail allows users to reinitialize the alias database, then corrupt the alias database by interrupting sendmail.									
S97	CVE-1999-0986	(m)IP	No	SysState	`netinet`	2-10-2-4			1	1
	The ping command in Linux 2.0.3x allows local users to cause a denial of service by sending large packets with the -R (record route) option.									
S98	CVE-1999-0996	Windows	No		`httpd`	2-2-1-1	G12		2	1
	Buffer overflow in Infoseek Ultraseek search engine allows remote attackers to execute commands via a long GET request.									
S99	CVE-1999-1010	(m)ssh	No	User	`ssh`	3			5	3
	An SSH 1.2.27 server allows a client to use the “none” cipher, even if it is not allowed by the server policy.									
S100	CVE-2000-0003	UnixWare	No			2-3-2-1	G6			
	Buffer overflow in UnixWare rtpm program allows local users to gain privileges via a long environmental variable.									
S101	CVE-2000-0011	Windows	No			2-2-1-1	G12			
	Buffer overflow in AnalogX SimpleServer:WWW HTTP server allows remote attackers to execute commands via a long GET request.									
S102	CVE-2000-0014	Windows	No		`httpd`	2-2-1-3		G3		
	Denial of service in Savant web server via a null character in the requested URL.									

(continued on next page)

Table A.1: List of specific detectors implemented (continued).

N	ID	Vuln	S	Src	Dir	Class	Det	Imp	E	B
S103	CVE-2000-0015	Ascend CascadeView	No			2-12-2-1	G19, G21			
	CascadeView TFTP server allows local users to gain privileges via a symlink attack.									
S104	CVE-2000-0029	UnixWare	No			2-12-2-1	G19, G21			
	UnixWare pis and mkpis commands allow local users to gain privileges via a symlink attack.									
S105	CVE-2000-0051	(m)Allaire Spectra	No		`httpd`	4		G3		
	The Allaire Spectra Configuration Wizard allows remote attackers to cause a denial of service by repeatedly resubmitting data collections for indexing via a URL.									
S106	CVE-2000-0144	Axis Network Document Server	No			2-12-2-2	G22			
	Axis 700 Network Scanner does not properly restrict access to administrator URLs, which allows users to bypass the password protection via a .. (dot dot) attack.									
S107	CVE-2000-0148	(m)MySQL	No	User	`mysql`	2-2-1-2			1	1
	MySQL 3.22 allows remote attackers to bypass password authentication and access a database via a short check string.									
S108	CVE-2000-0165	(m)Delegate	No		`httpd`	2-2-1-1		G3		
	The Delegate application proxy has several buffer overflows which allow a remote attacker to execute commands.									
S109	CVE-2000-0169	Windows	No		`httpd`	2-2-1-4		G3		
	Batch files in the Oracle web listener ows-bin directory allow remote attackers to execute commands via a malformed URL that includes '?&'.									
S110	CVE-2000-0170	Linux	No	User	`man`	2-3-2-1	G6		6	1
	Buffer overflow in the man program in Linux allows local users to gain privileges via the MANPAGER environmental variable.									

(continued on next page)

Table A.1: List of specific detectors implemented (continued).

N	ID	Vuln	S	Src	Dir	Class	Det	Imp	E	B
S111	CVE-2000-0174	(m)StarOffice	No			2-12-2-2	G22			
	StarOffice StarScheduler web server allows remote attackers to read arbitrary files via a .. (dot dot) attack.									
S112	CVE-2000-0175	(m)StarOffice	No		`httpd`	2-2-1-1	G12		2	1
	Buffer overflow in StarOffice StarScheduler web server allows remote attackers to gain root access via a long GET command.									
S113	CVE-2000-0208	(m)htdig	No		`httpd`	2-2-1-4		G3		
	The htdig (ht://Dig) CGI program htsearch allows remote attackers to read arbitrary files by enclosing the file name with backticks (‘) in parameters to htsearch.									
S114	CVE-2000-0210	Solaris	No			2-12-2-1	G19, G21			
	The lit program in Sun Flex License Manager (FlexLM) follows symlinks, which allows local users to modify arbitrary files.									
S115	CVE-2000-0221	Nortel Networks Nautica Router	No		`netinet`	2-10-2-1			3	1
	The Nautica Marlin bridge allows remote attackers to cause a denial of service via a zero length UDP packet to the SNMP port.									
S116	CVE-2000-0260	Windows	No		`httpd`	2-2-1-1		G3		
	Buffer overflow in the dvwssr.dll DLL in Microsoft Visual Interdev 1.0 allows users to cause a denial of service or execute commands, aka the “Link View Server-Side Component” vulnerability.									
S117	CVE-2000-0262	Windows	No			2-12-2-2	G22			
	The AVM KEN! ISDN Proxy server allows remote attackers to cause a denial of service via a malformed request.									
S118	CVE-2000-0279	BeOS	No	Net	`netinet`	2-10-2-1			3	1
	BeOS allows remote attackers to cause a denial of service via malformed packets whose length field is less than the length of the headers.									

(continued on next page)

Table A.1: List of specific detectors implemented (continued).

N	ID	Vuln	S	Src	Dir	Class	Det	Imp	E	B
S119	CVE-2000-0337	Solaris	No	User	`X11`	2-5-1-1	G1		4	1
	Buffer overflow in Xsun X server in Solaris 7 allows local users to gain root privileges via a long -dev parameter.									
S120	CVE-2000-0350	Windows	No	Net	`httpd`	4			2	1
	A debugging feature in NetworkICE ICEcap 2.0.23 and earlier is enabled, which allows a remote attacker to bypass the weak authentication and post unencrypted events.									
S121	CVE-2000-0391	(m)Kerberos5	No			2-5-1-1	G1			
	Buffer overflow in krshd in Kerberos 5 allows remote attackers to gain root privileges.									
S122	CVE-2000-0392	(m)Kerberos5	No			2-5-1-1	G1			
	Buffer overflow in ksu in Kerberos 5 allows local users to gain root privileges.									
S123	CVE-2000-0397	Windows	No		`httpd`	1		G3		
	The EMURL web-based email account software encodes predictable identifiers in user session URLs, which allows a remote attacker to access a user's email account.									
S124	CVE-2000-0405	(m)AntiSniff	No	Net	`antisniff`	2-6-1-1			2	2
	Buffer overflow in L0pht AntiSniff allows remote attackers to execute arbitrary commands via a malformed DNS response packet.									
S125	CVE-2000-0408	Windows	No		`httpd`	2-7-1-2		G3		
	IIS 4.05 and 5.0 allow remote attackers to cause a denial of service via a long, complex URL that appears to contain a large number of file extensions, aka the Malformed Extension Data in URL vulnerability.									
S126	CVE-2000-0411	(m)FormMail	No		`httpd`	1		G3		
	Matt Wright's FormMail CGI script allows remote attackers to obtain environmental variables via the env_report parameter.									
S127	CVE-2000-0418	Cayman	No			2-10-2-4	G11			
	The Cayman 3220-H DSL router allows remote attackers to cause a denial of service via oversized ICMP echo (ping) requests.									

(continued on next page)

Table A.1: List of specific detectors implemented (continued).

N	ID	Vuln	S	Src	Dir	Class	Det	Imp	E	B
S128	CVE-2000-0436	Windows	No			2-12-2-2	G22			
	MetaProducts Offline Explorer 1.2 and earlier allows remote attackers to access arbitrary files via a .. (dot dot) attack.									
S129	CVE-2000-0453	(m)XFree86	No	Net	X11	2-2-1-2			2	1
	XFree86 3.3.x and 4.0 allows a user to cause a denial of service via a negative counter value in a malformed TCP packet that is sent to port 6000.									
S130	CVE-2000-0460	(m)KDE	No			2-3-2-1	G6			
	Buffer overflow in KDE kdesud on Linux allows local uses to gain privileges via a long DISPLAY environmental variable.									

Table A.2: List of generic detectors implemented.

N	ID	S	Src	Dir	Class	Det	Imp	E	B
G1	ESP-ARGS-LEN	No	User	`kern`	2-5-1-1			2	1
	Generate an alert when any command argument is longer than a certain length.								
G2	ESP-BADMODE-ROOT-FILE	No	FileSys	`ufs`	n/a			7	4
	Generate an alert when a root-owned file becomes world-writable, SUID or SGID.								
G3	ESP-BADURLS	No	Net	`httpd`	n/a			40	2
	Generic detector for attacks based on specific URLs and for URL-based buffer overflows.								
G4	ESP-BOOT	No	SysState	`kern`	n/a			1	1
	Generate a message every time the system boot.								
G5	ESP-COLLECT-ARG-ENV-DATA	Yes	SysState	`kern`	n/a			26	5
	Collects and reports maximum and average length of command arguments and environment variables seen so far. It also reports the number of elements of each type included in the measurement.								
G6	ESP-ENV-LEN	No	SysState	`kern`	2-3-2-1			6	1
	Generate an alert when any environment variable is longer than a certain length.								
G7	ESP-FILE-INTEGRITY	No	FileSys	`kern`	n/a				
	Detector that monitors file integrity and triggers when a monitored file is modified.								
G8	ESP-FTP-CMD-OVERFLOW	No	User	`ftpd`	2-2-1-1			4	1
	Generate an alert when an FTP command is received that is too long or that contains invalid characters.								
G9	ESP-GETNAMEINFO	No	Net	`net`	2-6-1-1			1	1
	Detector for hostname-based buffer overflow attacks.								
G10	ESP-LONGGECOS	No	SysState	`gen`	2-6-1-1			2	1
	Generate an alert when the GECOS field in a user entry is longer than a certain threshold.								
G11	ESP-LONGICMP	No	Net	`netinet`	2-10-2-4			3	1
	Detector that generates an alert when an ICMP packet is received that is over a certain length.								
G12	ESP-LONGURL	No	Net	`httpd`	2-2-1-1			2	1
	Generic detector for attacks based on sending extremely long URLs to a web server.								

(continued on next page)

Table A.2: List of generic detectors implemented (continued).

N	ID	S	Src	Dir	Class	Det	Imp	E	B
G13	ESP-PORTSCAN	Yes	Net, SysState	`netinet`	n/a			151	9
	Different types of TCP port scanning, including full scans, SYN scans, FIN, XMAS and NULL scans.								
G14	ESP-SHUTDOWN	No	SysState	`kern`	n/a			5	1
	Generate a message every time the system shuts down.								
G15	ESP-SMTP-CMD-OVERFLOW	No	Net	`sendmail`	2-2-1-1			2	1
	Detect attempts at buffer overflow attacks on SMTP commands sent to sendmail.								
G16	ESP-SYMLINK-CHMOD	No	App, FileSys	`kern`	2-7-1-5	G21		2	1
	Detect attempts at changing the permissions (using the chmod() system call) of a file through a symlink in a public directory.								
G17	ESP-SYMLINK-CHOWN	No	App, FileSys	`kern`	2-7-1-5	G21		2	1
	Detect attempts at changing the owner or group (using the chown() system call) of a file through a symlink in a public directory.								
G18	ESP-SYMLINK-CONNECT	No	FileSys	`kern`	2-7-1-5	G21		2	1
	Detect attempts at connecting to a UNIX socket (using the connect() system call) that is a symlink in a public directory.								
G19	ESP-SYMLINK-OPEN	No	App, FileSys	`kern`	2-7-1-5, 2-12-2-1	G21		2	1
	Detect attempts to open files for writing or creating inside some world-writable directories, when the file already exists and is a symbolic link.								
G20	ESP-TCP-DROPPED-PACKETS	No	SysState	`netinet`	n/a			1	1
	A detector that triggers when an incoming TCP packet is dropped for any reason.								
G21	ESP-TMP-SYMLINK	No	FileSys	`kern`	2-7-1-5, 2-12-2-1			2	1
	Generate an alert when a symlink in a temporary directory is accessed for any reason.								

(continued on next page)

Table A.2: List of generic detectors implemented (continued).

N	ID	S	Src	Dir	Class	Det	Imp	E	B
G22	ESP-URI-DOTDOT	No		`httpd`	2-2-1-3, 2-12-2-2		G3		
	Use of ”../” in URLs to access files outside the normal space of web documents.								
G23	ESP-URI-NOPS	No		`httpd`	2-2-1-1		G3		
	Presence of NOP characters in an http request, which is usually a sign of a buffer overflow attempt.								

Appendix B: The ESP Library

The ESP library (`libesp`) was developed to provide a central wrapper around several functions that can be useful for embedded detectors and programs that read their messages. The functions in the library (except for the `esp_log()` system call and the `esp_logf()` function) are only available to user-level processes because a library cannot be linked against the Unix kernel.

The following functions exist in the current version of the ESP library:

esp_log: This is the direct interface to the `esp_log` system call. It takes a string pointer as argument and writes it to the message buffer.

esp_logf: Interface to the `esp_log` system call that knows how to handle format strings so that variable data can be written to the message buffer.

esp_open: Opens the `/dev/esplog` device for reading, so that a process can read messages produced by detectors.

esp_close: Closes the `/dev/esplog` device.

esp_gets: Reads the next message from the `/dev/esplog` device. In the current implementation, detector messages are produced as lines of ASCII text. However this interpretation may change in the future.

esp_match_char: This utility function performs a character-by-character comparison against a pattern. It is used by several detectors to perform comparisons needed to detect specific buffer overflow attacks.

esp_count_char: Counts how many times a specific character occurs in a string. It is used mostly through `esp_count_nops()`.

esp_count_nops: Counts the number of times the code for a NOP operation occurs in a string. NOP codes almost invariable occur in strings that are intended to cause a buffer overflow. Their occurrence is used as a strong heuristic for detecting some attacks.

esp_longest_char_seq: Returns the length of the longest contiguous sequence of a specific character that appears in a string.

esp_strcasestr: Similar to the standard `strstr()` function, it looks for the occurrence of a string inside another, but it does the search in a case-insensitive manner.

esp_strhead: It returns a buffer containing the "head" of a given string, up to a length specified by the user.

esp_check_path_level: Determines if a given path will try to go past the root directory when considered relative to a specific directory.

esp_path_level: Counts the level (starting with zero for the root directory) at which a given path will end up when considered relative to a specific directory.

Appendix C: A Taxonomy of Software Vulnerabilities

The taxonomy of software vulnerabilities proposed by Krsul [78] is used in this dissertation as one mechanism for classifying detectors and attacks. The categories used in this dissertation are listed in Table C.1 for reference. The classes listed *in italics* correspond to classes added to this taxonomy during the development of the ESP prototype, and are described in Section 4.13.6. A full listing and description of the taxonomy is outside the scope of this document but can be found in its original source.

Table C.1: Categories from the Krsul taxonomy used in this dissertation.

(1) Design faults			
(2) Environmental assumptions			
	(2-1) Running program	(2-1-3) Environment	(2-1-3-1) is `system()` safe
		(2-1-4) User running the program	(2-1-4-1) user is root or administrator
	(2-2) User input	(2-2-1) Content	(2-2-1-1) is at most x
			(2-2-1-3) matches regular expression
			(2-2-1-4) is free of shell metacharacters
	(2-3) Environment variable	(2-3-2) Content	(2-3-2-1) length is at most x
			(2-3-2-3) matches regular expression
	(2-4) Network stream	(2-4-1) Content	(2-4-1-1) is at most x
			(2-4-1-4) *matches a regular expression*
		(2-4-2) *Socket*	(2-4-2-1) *is the same object as x*
	(2-5) Command line parameters	(2-5-1) Content	(2-5-1-1) length is at most x
	(2-6) System library	(2-6-1) Return	(2-6-1-1) length is at most x
	(2-7) File	(2-7-1) Name	(2-7-1-2) is a valid file name
			(2-7-1-4) is the same object as x
			(2-7-1-5) is final
			(2-7-1-6) *length is at most x*
		(2-7-2) Content	(2-7-2-1) length is at most x
			(2-7-2-3) is a known program
			(2-7-2-5) is of a known type
	(2-8) Directory	(2-8-1) Name	(2-8-1-1) length is at least x
	(2-9) Program string	(2-9-1) Content	(2-9-1-3) is free of shell metacharacters
	(2-10) Network IP packets	(2-10-1) Source address	(2-10-1-1) *is different than destination address*

(continued on next page)

Table C.1: Categories from the Krsul taxonomy used in this dissertation (continued).

		(2-10-2) Data segment	(2-10-2-1) length is at least x
			(2-10-2-2) *is a proper fragment*
			(2-10-2-3) *corresponds to a fully established connection*
			(2-10-2-4) *length is at most x*
		(2-10-4) *TCP seq. number*	(2-10-4-1) *is in proper sequence*
	(2-11) Directory, running program	(2-11-1) Dir. name, running program privileges, name of user that ran the program	(2-11-1-1) is in valid user space for the user that invoked the program.
			(2-11-1-3) user that invoked the program can create files in the directory
	(2-12) File, running program	(2-12-1) File perms., running program privileges, user that ran the program	(2-12-1-1) User that invoked the program can read the file
			(2-12-1-2) User that invoked the program can write to the file
		(2-12-2) File name, running program privileges, user that ran the program	(2-12-2-1) is a valid temporary file
			(2-12-2-2) is in valid user space for the user that invoked the program.
(3) Coding faults			
(4) Configuration errors			

VITA

VITA

Diego Zamboni was born on December 3, 1970 in Corrientes, Argentina. He received his Bachelor's Degree in Computer Engineering from the National Autonomous University of Mexico (UNAM) in 1995, and his Masters Degree in Computer Science from Purdue University in 1998. While at UNAM, he was in charge of the security for the Unix machines at the Supercomputing Department. He also established the University's Computer Security Area, one of the first Computer Security Incident Response Teams in Mexico. In July of 2001, he was awarded the first Josef Raviv Memorial Postdoctoral Fellowship from IBM.