

# Using internal sensors and embedded detectors for intrusion detection\*

Florian Kerschbaum   Eugene H. Spafford   Diego Zamboni

Center for Education and Research in Information Assurance and Security  
1315 Recitation Building  
Purdue University  
West Lafayette, IN 47907-1315  
{kerschf , spaf , zamboni}@cerias.purdue.edu

July 24, 2001

## Abstract

We introduce the concept of using internal sensors to perform intrusion detection in computer systems. We show its practical feasibility and discuss 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. Our implementation shows that it is possible to build both specific (specialized for a certain intrusion) and generic (able to detect different types of intrusions) detectors.

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

The field of intrusion detection has received increasing attention in recent years. One reason for this is the explosive growth of the Internet and the large number of networked systems that exist in all types of organizations. The increase in the number of networked machines has led to an

increase in unauthorized activity [12], not only from external attackers, but also from internal sources, such as disgruntled employees and people abusing their privileges for personal gain [59].

In the last few years multiple intrusion detection systems have been developed, both in the commercial and academic sectors. These systems use various approaches to detecting 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. 18, 45] or per-host data collection and analysis components [e.g. 30, 58]. In their implementation, all of these approaches are subject to similar problems.

First, they continuously use additional resources in the system they are monitoring even though there are no intrusions occurring, because the components of the intrusion detection system have to be running all the time.

Second, 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.

Finally, the information used by the intrusion detection system is usually obtained from audit trails or from packets on a network. Data have to traverse a longer path from their 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 those audit trails, which in many cases results in misinterpretations or missed events.

Even systems designed to use software agents [e.g. 4, 70] suffer from the problems mentioned, because in practice they have implemented agents as separate processes running on each host.

---

\*Portions of this work were supported by sponsors of CERIAS.

In this document, we present a classification of data collection mechanisms for intrusion detection. We propose an architecture for intrusion detection which addresses the problems mentioned above, and describe a prototype implementation. We describe case studies for two specific types of attacks, and present some performance results for our implementation.

## 1.1 Intrusion Detection

Intrusion detection is 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’)” [49]. We add to this definition the identification of *attempts* to use a computer system without authorization or to abuse existing privileges. Our working definition matches the one given by Heady et al. [29]:

### DEFINITION 1: INTRUSION

Any set of actions that attempt to compromise the integrity, confidentiality, or availability of a 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 paper we use the terms *attack* and *intrusion* interchangeably.

The definition of the word *intrusion* in an english dictionary [50] does not include the concept of an insider abusing his or her privileges to perform unauthorized actions, 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.

An *intrusion detection system* is a computer system (possibly a combination of software and hardware) that attempts to perform intrusion detection, as defined above. Most intrusion detection systems try to perform their task in real time [49], but 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. 39] or because they are geared for forensic analysis [24, 78].

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. 15, 69] 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.

In some of the definitions given below we use the term *monitored component* as follows:

### DEFINITION 2: 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.

## 1.2 Data analysis structure

Intrusion detection systems are usually classified as host-based or network-based [49]. 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.

However, this classification only addresses the general source from which data are collected by the intrusion detection system, and not the specific mechanisms used, or how and where they are processed. To make this distinction clear, we use the terms *host-based data collection* and *network-based data collection* instead.

With respect to how the data analysis components are distributed, we classify intrusion detection systems as centralized or distributed, as follows [70]:

### DEFINITION 3: 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.

Some existing intrusion detection systems that we classify as centralized are IDES [45], IDIOT [18, 40], NADIR [32] and NSM [30].

### DEFINITION 4: 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.

Some existing intrusion detection systems that we classify as distributed are DIDS [68], GrIDS [71], EMERALD [58] and AAFID [70].

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. In the last few years, an increasing number of distributed intrusion detection systems has been designed and built [e.g. 6, 34, 58, 70, 71].

### 1.3 Coupling between data collection and data analysis

Data collection and data analysis are usually considered as two distinct steps of an intrusion detection system. 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 [2] and intrusion detection systems [57].

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.

## 2 Data collection architectures

The performance of an intrusion detection system can only be as good—in terms of accuracy, reliability, and

efficiency—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.” [60, p. 107]

With the goal of better understanding the characteristics that make data collection mechanisms suitable for intrusion detection, in this section we provide two conceptual (centralized/distributed and direct/indirect) and two practical (host/network-based and external/internal) classifications of data collection mechanisms. We discuss the advantages and disadvantages of each one of them.

### 2.1 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 [2].

#### DEFINITION 5: 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 6: 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, in the same sense as in Section 1.2.

Both distributed and centralized data collection have been widely used in existing intrusion detection systems [82]. A report by Axelsson [2] 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 our purposes, 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.

## 2.2 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` [77] 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 7: DIRECT MONITORING

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

### DEFINITION 8: 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 [79] 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 [41] 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 [48].

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.

Few existing intrusion detection systems use some form of direct monitoring [82]. 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.

### 2.3 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 9: 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 10: 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. Examples of this type of actions are executing an invalid command, or accessing a network service and providing it malformed 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 [6]. Therefore, attacks on the network infrastructure can also be considered as acting on 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 [36] could be detected at the ICMP layer in the host by looking for the occurrence of a large number of ECHO\_RESPONSE packets.

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 8). In general, it is advisable to use host-based data collection because it constitutes a form of direct monitoring, which results in some specific advantages, such as the possibility of monitoring encrypted traffic, and the avoidance of insertion and evasion attacks Ptacek and Newsham [61]. Network-based data collection also has some advantages, including ease of deployment.

The relationship between the traditional host-based/network-based classification of intrusion detection

systems [49] and the types of monitoring described in Section 2.2 is as follows: 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 [82] in an attempt to obtain the most complete view of the hosts being monitored.

### 2.4 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 11: 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 12: 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 [77] to obtain process information on a Unix system would 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 [42] 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 [56]. 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. Table 1 lists 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 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. For this reason, the base overhead associated with the creation of a separate process is avoided.

**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.

Despite their advantages, few existing intrusion detection systems use internal sensors [82], and most of those were designed for detecting specific types of attacks. 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.

## 2.5 Experiences in building a distributed intrusion detection system

AAFID [70] 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 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 2.2.

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** [77], **netstat** [75] or **df** [73]), 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

Advantages	Disadvantages
<b>External sensors</b>	
<ul style="list-style-type: none"> <li>• Easily modified, added or removed from a host.</li> <li>• Can be implemented in any programming language that is appropriate for the task.</li> </ul>	<ul style="list-style-type: none"> <li>• 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.</li> <li>• 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).</li> <li>• Can potentially be disabled or modified by an intruder.</li> <li>• Added performance impact because the sensors are separate components—processes, threads, or loaded libraries—possibly running continuously.</li> <li>• Limited access to information because they depend on existing mechanisms (such as user-level commands or system calls) to obtain it.</li> </ul>
<b>Internal sensors</b>	
<ul style="list-style-type: none"> <li>• Minimum delay between the generation of the information and its use because it can be obtained at its source.</li> <li>• 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.</li> <li>• Cannot be easily disabled or modified because they are not separate processes.</li> <li>• 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.</li> <li>• 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.</li> <li>• Because they are implemented as part of the program they are monitoring, they can access any information that is necessary for their task.</li> </ul>	<ul style="list-style-type: none"> <li>• Their implementation requires access to the source code of the program that needs to be monitored.</li> <li>• 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.</li> <li>• Need to be implemented in the same language as the program they are going to monitor.</li> <li>• If designed or implemented incorrectly, they can severely harm the performance or the functionality of the program they are part of.</li> <li>• Harder to update or modify and to port to different operating systems, or even to different versions of the same program.</li> <li>• Reduced portability, because the sensors depend on the specifics of the code where they are implemented.</li> </ul>

Table 1: Advantages and disadvantages of external and internal sensors.

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 [42].

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 limitations described in Section 2.4. For this reason, we started further exploration of the use of internal sensors for doing intrusion detection.

### 3 Related work

The lack of adequate audit data for intrusion detection was documented by Price [60], 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 [19, 20] 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. [25] 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 [69], 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 [21]. 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 [23] 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 [67], who discussed its advantages from a theoretical standpoint. Internal sensors are an ideal tool for performing application-specific intrusion detection 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 [42] 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.

Few intrusion detection systems have been developed using internal sensors. CylantSecure [80] 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 [35] and Openwall [54] projects have developed kernel patches for Linux [5] that prevent certain operations defined as “dangerous.” These patches add checks that constitute internal sensors specifically tuned for preventing those operations.

Another example of the use of internal sensors is FormatGuard [17]. This is a specialized tool for detecting and preventing format-string-based buffer overflows [51, 65]. 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.

## 4 Embedded detectors

To investigate the use of internal sensors for intrusion detection, we have defined an architecture for intrusion detection called ESP that uses embedded detectors for distributed, localized data reduction.

### DEFINITION 13: 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.

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.

### 4.1 How embedded detectors work

Figure 1 shows an example of a simple embedded detector. The code on the left is potentially vulnerable to a buffer-overflow attack [1] 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 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. See also the discussion in Section 5.1.2.

### 4.2 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. 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. In some cases, the internal sensor is clearly differentiable in the code. For example, a detector for port scans [27] 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 [8] 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. It is also possible that a single detector collects data from multiple sensors.

### 4.3 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:

### DEFINITION 14: STATELESS EMBEDDED DETECTOR

A detector that bases its decisions solely on information present in the program at the time of evaluation, or that

<pre> 1 char buf[256]; 2 strcpy(buf, getenv("HOME")); </pre> <p style="text-align: center;">Code before inserting detector</p>		<pre> 1 char buf[256]; 2 { 3     if (strlen(getenv("HOME"))&gt;255) { 4         log_alert("buffer overflow"); 5     } 6 } 7 strcpy(buf, getenv("HOME")); </pre> <p style="text-align: center;">Code after inserting detector</p>
--	--	--

Figure 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.

can be obtained from the system at the moment it is needed.

**DEFINITION 15: 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 4.2.

## 5 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

The main class of components, which we describe in this paper, is the internal sensors and embedded detectors, which is used for direct monitoring of a host and for localized analysis of the data.

### 5.1 Distinguishing characteristics

The use of internal sensors gives intrusion detection systems implemented based on the ESP architecture some unique characteristics with respect to how they collect and process data, and also impacts some of their operational features.

#### 5.1.1 Types of data observed

ESP 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 2.2. When compared to other intrusion detection techniques that observe program behavior [e.g. 33, 34], ESP has the advantage of being able to observe the internal data and state of the program and not only its externally observable behavior.

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. It also makes it possible for detectors to use the existing defense mechanisms of the monitored component (for example, if the program already looks for malicious activity) and combine them with detection.

#### 5.1.2 Types of detectable attacks

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). We used this feature for exploring the detection of intrusions over multiple architectures and platforms.

Figure 2 shows code similar to the one in Figure 1, but this code is not vulnerable to a buffer overflow because the `strcpy` function is being used. However, the same detector can be added to this code as shown on the right side of Figure 2. This example shows how embedded detectors can exist even in code that is not vulnerable to the attacks for which the detectors look.

### 5.1.3 Tighter coupling between event collection and event analysis

As mentioned in Section 1.3, data collection and data analysis have traditionally been two 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 5.1.1.

### 5.1.4 Intrusion detection at the application and operating system level

Application-based intrusion detection systems [67] 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.

### 5.1.5 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.

### 5.1.6 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. It is also conceivable that the intrusion detection system could also stop the intrusions before they cause any damage.

### 5.1.7 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.

### 5.1.8 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.

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.

## 6 Implementation

In this section, we describe the implementation details for our ESP prototype, that uses embedded detectors for intrusion detection.

### 6.1 Specific and generic detectors

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

#### DEFINITION 16: SPECIFIC DETECTOR

An embedded detector designed to detect one specific attack.

<pre> 1 char buf[256]; 2 strncpy(buf, getenv("HOME"), 3         sizeof(buf)); </pre>		<pre> 1 char buf[256]; 2 { 3     if (strlen(getenv("HOME"))&gt;255) { 4         log_alert("buffer overflow"); 5     } 6 } 7 strncpy(buf, getenv("HOME"), 8         sizeof(buf)); </pre>
Code before inserting detector		Code after inserting detector

Figure 2: 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.

#### DEFINITION 17: 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.

Our 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.

## 6.2 Sources of information

The detectors we have implemented map directly to entries in the CVE (Common Vulnerabilities and Exposures) database [13, 47]. The CVE is not a taxonomy or a classification scheme and is used only as a fairly complete and recognized list of known vulnerabilities and attacks. Linking each detector to a CVE entry facilitates discussion and reference, and ensures that no repeated detectors are implemented. For the work reported in this document, version 20000712 of the database was used. This version was published on July 12 of 2000 and contained 815 entries.

Information about the attacks themselves, including exploits, was gathered from common sources on the Internet [e.g. 7, 55, 63, 66, 81].

## 6.3 Implementation platform

The detectors in our prototype have been implemented in the OpenBSD 2.7 operating system [53]. This 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 [46, 72] about the internals of the kernel.

- The source code is managed and distributed as a single directory tree. The source tree of OpenBSD 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 line-by-line code security audit process. Most of the security problems for which detectors are implemented have already been fixed in OpenBSD. Looking at the security patches and at the change log for each file makes it easier to locate the portions of code where the problems existed, and helps in determining where the detectors for each attack have 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 exist, it is easier to try attacks against the instrumented system without worrying about the adverse effects they could have on the host.

As described in Section 5.1.2, although OpenBSD is used as the implementation platform, we can 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 are 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.

## 6.4 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 Balasubramanian et al. [3], but decided against them primarily because of the overhead they require and because they are geared towards exchanging messages between separate processes.

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.

Access to the `esp_log` system call and some other utility functions is provided through a library we implemented for this purpose, called `libesp`.

## 6.5 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 modify any data used by the program, nor alter its flow in any way. We refer to this guideline as “the prime directive for detectors” [52].

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 [64] attacks—to avoid adding unnecessary code to the system.

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.

## 6.6 Naming and measuring detectors

Each detector is given a unique identifier. For detectors inspired by CVE entries, this identifier is the corresponding CVE name. 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. For example, “ESP-PORTSCAN” and “ESP-TMP-SYMLINK” are valid identifiers.

An important aspect of the ESP detectors is their small size, so we were interested in measuring them. 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. We used the definition of “executable statement” as provided in the Source Code Counting Rules described by Jones [37] and as implemented by Metre [44].

For example, the detector shown in the right side of Figure 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 1 has a BOCAM count of 1, because all its code is in a single contiguous block.

## 7 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. In some cases the code has been reformatted (and some of the messages changed) for reasons of space.

### 7.1 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 [21]. In this section, we describe this implementation and the results obtained.

#### 7.1.1 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 2 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 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.

#### 7.1.2 Stateless Detectors

Twelve of the 16 detectors in Table 2 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 [10] 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("LAND attack\n");
#endif
        goto drop;
    }
```

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

Table 2: 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 4.3. The ESAM and BOCAM columns indicate the sizes as defined in Section 6.6.

```

}
...
}

```

The CVE-1999-0103 (Echo-chargen denial-of-service attack [9]) detector was implemented within the `inetd` [74] 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.

SYN flooding [64] 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.

### 7.1.3 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 [11, 36] 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 [11] sends ICMP ECHO\_RESPONSE packets. Those do not differ from legitimate packets (for example, in response to a `ping` command [76]) 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 [27] 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 [38].

#### 7.1.4 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.

## 7.2 Embedded detectors for sendmail attacks

Sendmail [16] 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 [14].

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.

### 7.2.1 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 3.

In the next sections we will describe in more detail some of these detectors. As with the network detectors described in Section 7.1, 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.

### 7.2.2 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 over-

ID	Description	Type	ESAM	BOCAM
CVE-1999-0047	Buffer overflow vulnerability in sendmail 8.8.3/8.8.4	Stateful	10 <sup>1</sup>	7 <sup>1</sup>
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

<sup>1</sup> These counts include a subroutine that is shared with CVE-1999-0206 that consists of 4 executable statements.

Table 3: 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 4.3. The ESAM and BOCAM columns indicate the sizes as defined in Section 6.6.

write 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:

```

...
a->q_next = a1;
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.

### 7.2.3 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()`:

```
#ifndef ESP_CVE_1999_0047
char
esp_mime_buffer_overflow(char c,
    int filled, char *msg) {
    char egg[] =
        "\xeb\x37\xe"
        (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.

## 7.2.4 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.

## 7.3 Additional detectors

After the initial case studies, a number of additional detectors were implemented. We followed a consistent methodology for their implementation:

1. Select a detector to implement. In the case of specific detectors, this corresponded to selecting an entry from the CVE database. Entries were selected at random from the CVE to ensure coverage of different types of attacks.
2. 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 6.2.
3. 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.

4. 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.
5. 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.

As a special case, we implemented a file integrity monitoring detector [22] that can provide detection capabilities for successful attacks that modify system files. This monitoring is similar to that performed by other tools [e.g. 39], but with a tighter integration into the operating system.

## 8 Detector size statistics

In total, 130 specific detectors were implemented. 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. Of the 130 specific detectors, 91 had code associated with them, and 39 were detected by one of the generic detectors.

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 6.6, 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. Figure 3 combines the ESAM and BOCAM measurements and shows the count of detectors against each combination of ESAM and BOCAM values. This graph shows that most detectors are small and non-fragmented, with 78% of the detectors being 4 ESAM or less in size.

All the detectors implemented account for 507 ESAM, resulting in an average detector length of 5.57 ESAM (this

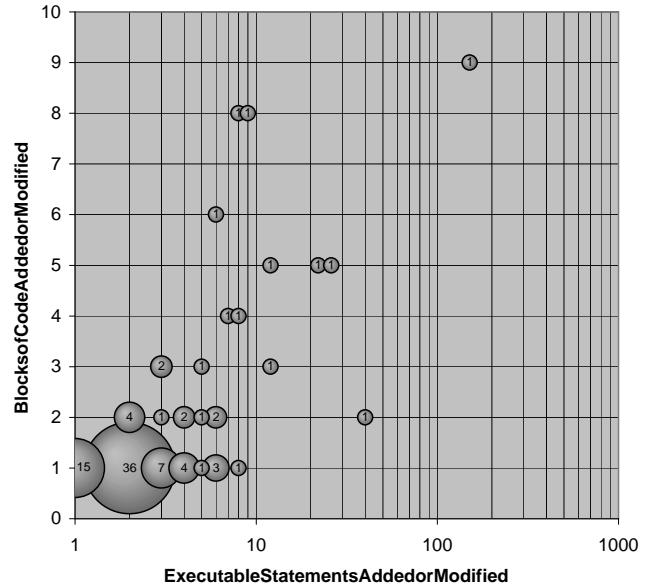


Figure 3: Graph of detector sizes by combination of the ESAM and BOCAM metrics. Bubble size represents the number of detectors that have each combination of parameters. The horizontal axis has been made logarithmic to better display the different values at the lower end of the scale.

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.

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.

## 9 Testing ESP

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.

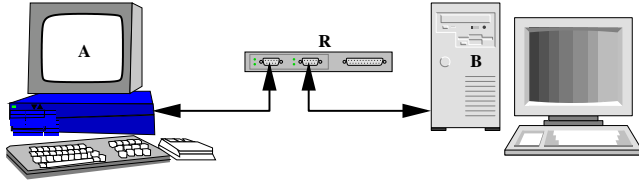


Figure 4: General setup for the performance tests of the ESP implementation. Host B is the server, host A is the client, and R represents a host acting as a transparent bridge between A and B. The three hosts are on dedicated point-to-point connections over 100Mbit/s full-duplex Ethernet.

## 9.1 Performance testing

We present performance results for the detectors described in Section 7.1 for network-based attacks. We decided to focus on this group because they are one of the largest groups of detectors in the ESP implementation (24 detectors). Also, many of these run in the kernel, so their performance is critical for overall system performance. Detectors are additional code to be executed, and because they do not interfere with their surrounding code, their main impact is additional execution time. We measured CPU utilization and compared kernels compiled with and without detectors.

The setup for the test was as shown in Figure 4: 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 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 [26] and **dummy.net** [62] for imposing constraints on the bandwidth.

The test was done using a subset of the NetPerf [31] 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.

For each value of the independent variable (bandwidth),

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** [43] command, which uses information gathered in the `stat-clock()` function within the kernel context switch [46, 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.

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 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.

### 9.1.1 Performance test results

Figure 5 shows the CPU measurements obtained in host B during the execution of the NetPerf experiment. There are 20 points at each value of the independent variable  $X$  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 difference 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 pairwise F-test was done for each value of  $X$ , and its results are shown in Table 4. 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.

The results of this experiment show that detectors have a larger impact as the amount of work that the system does increases. The detectors can have a visible impact on the CPU load of the host, particularly for high values of  $X$ . 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

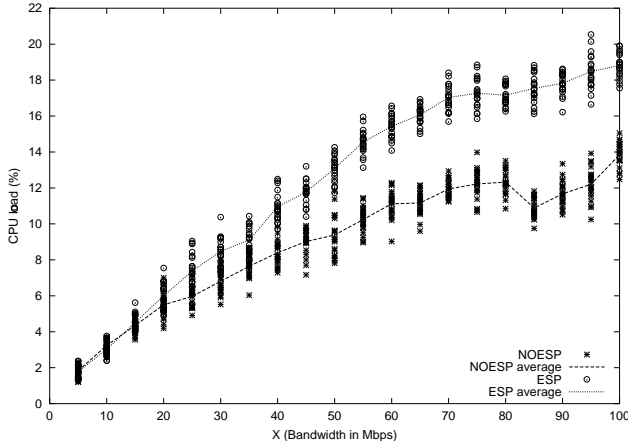


Figure 5: Plot of the CPU utilization measurements from the network performance experiment, showing the mean values for the ESP and NOESP cases.

X	Mean CPU %		Diff.	p-value
	NOESP	ESP		
5	1.83	1.79	-0.04	0.8507
10	3.24	3.02	-0.21	0.3330
15	4.35	4.51	0.16	0.4647
20	5.50	5.99	0.49	0.0270
25	5.97	7.38	1.41	< 0.0001
30	6.81	8.47	1.66	< 0.0001
35	7.64	9.10	1.46	< 0.0001
40	8.39	10.96	2.56	< 0.0001
45	9.02	11.77	2.75	< 0.0001
50	9.39	13.09	3.71	< 0.0001
55	10.24	14.54	4.31	< 0.0001
60	11.12	15.43	4.31	< 0.0001
65	11.16	16.07	4.90	< 0.0001
70	11.94	17.03	5.09	< 0.0001
75	12.22	17.27	5.05	< 0.0001
80	12.32	17.16	4.83	< 0.0001
85	10.88	17.54	6.66	< 0.0001
90	11.67	17.82	6.14	< 0.0001
95	12.20	18.48	6.27	< 0.0001
100	13.85	18.82	4.97	< 0.0001

Table 4: Statistics and analysis results for data from the network performance experiment.

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 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.

Our objective in presenting these measurements is not to present absolute figures of performance, but to show our conclusion that in practice, the existence of embedded detectors in a system does not cause any unreasonable difference in performance.

## 9.2 Detection testing

The purpose of the detection test was to determine the possibility of detecting new attacks using embedded detectors. 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.

As a source of information, we monitored the BugTraq mailing list [7] for a period of slightly over one month, from May 3, 2001 to June 8, 2001. New attacks published during these period 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.

**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.

After testing, each attack was classified in one 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.

- 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 its functionality. In this case, we recorded the detector to which the changes would have to be made, and what those changes would be.
- 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.
- 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.
- 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.
- 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.

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.

### 9.3 Results from the detection test

In total, 157 attacks were examined, of which 80 were applicable to our implementation platform. 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. The total categories (TD, TDM and ND) are displayed also in Figure 6.

Because each batch incorporates the changes made after the previous batch, it is also of interest to analyze the total

Category	Batches				Total
	#1	#2	#3	#4	
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

Table 5: 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.

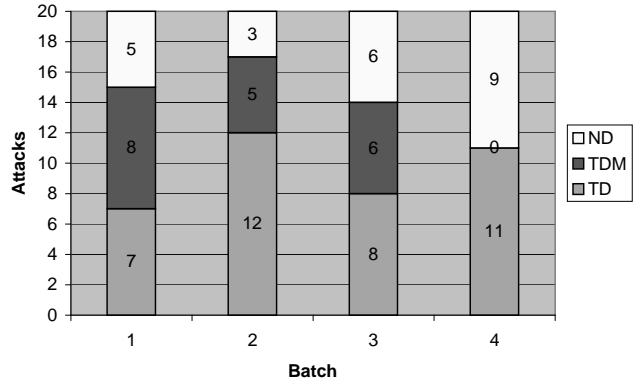


Figure 6: Total of attacks marked as “detected”, “detected with modifications” and “not detected” for each one of the batches of the detection test. In this graph, attacks belonging to the DSDC category (both DS and DC) are counted in TD.

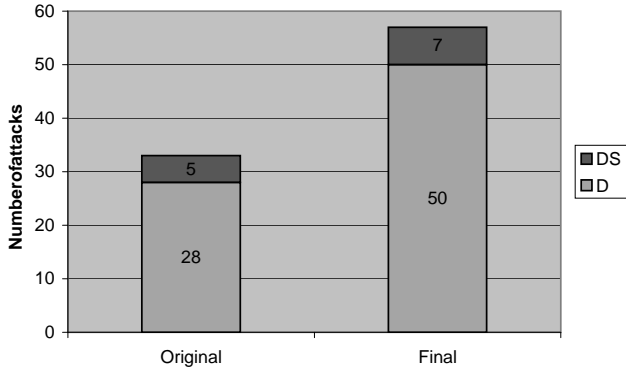


Figure 7: Total number of attacks that would have been detected by the original detectors, and by the detectors after the changes.

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 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% if we count the ones in group DS). After making the changes, the detection rate went up to 63% (71% with DS).

We implemented detectors for 130 out of 815 entries in the CVE database, corresponding to 16% 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% 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.

## 10 Conclusions

In this paper, we studied the characteristics of different methods of data collection for intrusion detection systems, and claimed that direct data collection provides more reliable and timely data for use in intrusion detection. Direct data collection can be performed using external or internal sensors. Internal sensors can provide data in the most reliable and efficient manner, while being resistant to tampering with the sensor, or to modification of the data being produced.

We proposed an architecture based on using internal sensors built into the code of the programs that are monitored, that extract information from the places in which it is gen-

erated 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.

The results of our experiments are very encouraging. Our specific embedded detectors achieve a 100% detection rate for the attacks they were designed to detect. The detectors have reasonable impact on the performance of the host and are implemented with little modifications, as shown by our code-size statistics. The detectors have been the simplest in the cases where the program already checks for the attack. This has been mostly the case for vulnerabilities that depend on implementation flaws, such as the Land and Teardrop attacks.

Embedded detectors have the following specific advantages over other mechanisms for detecting intrusions:

- They can use internal data of the program they are monitoring, which allows them to more accurately detect intrusions.
- They can use the defense mechanisms that already exist in the program being monitored, and combine them with detection.
- They can collect and provide their own state information to aid in detection.
- They can provide detailed information about the internal state of the program when the intrusion was detected, which can be helpful for further analysis of the reports.
- They can detect both successful and unsuccessful attempts to exploit vulnerabilities. Furthermore, they can detect attempts to exploit vulnerabilities that no longer exist in the program, or that never existed.

We showed a group of detectors that were added to the OpenBSD kernel for detecting network-based attacks. Because the detectors can exist at the lowest levels of the kernel, they are able to function and produce reports even when the system is under heavy load (possibly as a result of an attack). Our measurements show that the detectors have a

reasonable impact on the CPU utilization of the host being monitored, even under heavy network load.

Of course, there are issues that would the practical deployment of intrusion detection systems based on internal sensors. These include the implementation difficulty of sensors, the lack of tools and mechanisms for making it simple for developers to incorporate sensors and detectors into their programs, the lack of formal guidelines to indicate where sensors need to be located and how they should be implemented, and the possibility of severely impacting the performance and reliability of a host through deficient implementation. These issues should be explored and addressed in future work.

We believe that the use of internal sensors in general (and embedded detectors as a special case) holds promise for the development of efficient, compact and resilient intrusion detection systems.

## 11 Future work

The work presented in this paper 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.

Our work has focused 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 [28] 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 detectors can have a reasonable but still significant impact on the performance of the hosts. In this respect, practical work is necessary to explore optimization techniques. 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 9.2 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 different types of vulnerabilities with the expected occur-

rence of each type in new attacks 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 to determine cost-effective policies for sensor and detector maintenance and upgrading.

Erlingsson and Schneider [23] described the use of *reference monitors* that are automatically generated. In our work, the internal sensors and embedded detectors have been individually hand-coded. 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 developers to insert sensors and detectors in their programs.

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 [69], which uses internal sensors to perform both detection and reaction to attacks.

We have 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.

## References

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

- [4] Bruce Barnett and Dai N. Vu. Vulnerability assessment and intrusion detection with dynamic software agents. In *Proceedings of the Software Technology Conference*, April 1997.
- [5] Michael Beck, Harold Bohme, Mirko Dzladzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *Linux Kernel Internals*. Addison-Wesley, Reading, MA, USA, 1996.
- [6] 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, May 3–6 1998. IEEE Press.
- [7] BugTraq. Mailing list archive. Web page at <http://www.securityfocus.com/>, 1999–2001.
- [8] 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>.
- [9] 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>.
- [10] 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>.
- [11] 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>.
- [12] CERT Coordination Center. CERT/CC statistics. URL [http://www.cert.org/stats/cert\\_stats.html](http://www.cert.org/stats/cert_stats.html), 2001.
- [13] Steven Christey, Mann, and Hill. Development of a common vulnerability enumeration. Workshop RAID99, September 1999.
- [14] Cisco Secure Consulting. Vulnerability statistics report. Available online at [http://www.ieng.com/warp/public/778/security/vuln\\_stats\\_02-03-00.html](http://www.ieng.com/warp/public/778/security/vuln_stats_02-03-00.html), 2001.
- [15] Cisco Systems. Cisco Secure Intrusion Detection. Web page at <http://www.cisco.com/warp/public/cc/pd/sqsw/sqidsz/index.shtml>, June 2001.
- [16] Bryan Costales and Eric Allman. *sendmail*. O’Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, second edition, 1997.
- [17] 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.
- [18] 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, IN 47907-1398, September 1996. URL <http://www.cerias.purdue.edu/techreports/public/96-04.ps>.
- [19] Mark Crosbie and Eugene H. Spafford. Defending a computer system using autonomous agents. In *Proceedings of the 18th National Information Systems Security Conference*, volume II, pages 549–558, Oct 1995. URL [ftp://ftp.cerias.purdue.edu/pub/doc/intrusion\\_detection/mcrosbie-spaf-NISC-paper.ps.Z](ftp://ftp.cerias.purdue.edu/pub/doc/intrusion_detection/mcrosbie-spaf-NISC-paper.ps.Z).
- [20] 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](ftp://ftp.cerias.purdue.edu/pub/doc/intrusion_detection/mcrosbie-spaf-AAAI-paper.ps.Z).
- [21] 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.
- [22] James P. Early. An embedded sensor for monitoring file integrity. CERIAS TR 2001-41, CERIAS, Purdue University, West Lafayette, IN, March 2001.
- [23] 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.
- [24] Dan Farmer and Wietse Venema. Computer forensics analysis class handouts. Web page at <http://www.fish.com/forensics/>, August 1999. Accessed in May 2000.
- [25] 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>.
- [26] FreeBSD. Web page at <http://www.freebsd.org/>, July 2001.
- [27] Fyodor (fyodor@dhp.com). The art of port scanning. Internet [http://www.insecure.org/nmap/nmap\\_doc.html](http://www.insecure.org/nmap/nmap_doc.html), September 1997.
- [28] 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.
- [29] 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.
- [30] 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>.
- [31] Hewlett-Packard. Netperf. Website at <http://www.netperf.org/>, 2001.
- [32] 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.
- [33] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6: 151–180, 1998.
- [34] 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/hofmeyr-distributed-detection.ps.gz](ftp://coast.cs.purdue.edu/pub/doc/intrusion_detection/hofmeyr-distributed-detection.ps.gz).
- [35] Xie Huagang. Build a secure system with LIDS. Online at [http://www.lids.org/document/build\\_lids-0.2.html](http://www.lids.org/document/build_lids-0.2.html), October 2000.
- [36] 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.
- [37] Capers Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, New York, NY, 1991.
- [38] 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.
- [39] 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.
- [40] Sandeep Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue University, West Lafayette, IN 47907, 1995. URL <ftp://coast.cs.purdue.edu/pub/COAST/papers/sandeep-kumar/kumar-intdet-phddiss.ps.Z>.
- [41] 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.
- [42] 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, IN, October 1998. URL <https://www.cerias.purdue.edu/techreports-ssl/public/99-11.ps>.
- [43] William LeFebvre. *Top: display and update information about the top CPU processes*, 2001. Unix manual pages.

- [44] 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.
- [45] T. 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, CA, February 1992.
- [46] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.BSD Operating System*. Addison-Wesley, Reading, MA, USA, 1996.
- [47] MITRE. Common vulnerabilities and exposures. Web page at <http://cve.mitre.org/>, 1999–2000.
- [48] 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](ftp://ftp.cerias.purdue.edu/pub/doc/intrusion_detection/mounji_phd_thesis.ps.Z).
- [49] Biswanath Mukherjee, Todd L. Heberlein, and Karl N. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, May/June 1994.
- [50] Victoria Neufeldt and David B. Guralnik, editors. *Webster's New World Dictionary of American English*. Simon & Schuster, Inc., third college edition, 1988.
- [51] Tim Newsham. Format string attacks. Whitepaper, Guardent, 2000. URL <http://julianor.tripod.com/tn-usfs.pdf>.
- [52] Michael Okuda and Denise Okuda. *The Star Trek Encyclopedia: A Reference Guide to the Future*. Simon and Shuster Incorporated, August 1999.
- [53] OpenBSD. Web page at <http://www.openbsd.org/>, July 2001.
- [54] Openwall Project. Linux kernel patch from the Openwall project. Web page at <http://www.openwall.com/linux/>, June 2001.
- [55] Packet Storm. Web page at <http://packetstorm.securify.com>, 2000.
- [56] Wendy W. Peng and Dolores R. Wallace. Software error analysis. NIST Special Publication 500-209, National Institute of Standards and Technology, Gaithersburg, MD, March 1993. URL <http://hissa.nist.gov/HHRFdata/Artifacts/ITLdoc/209/error.htm>.
- [57] 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.
- [58] 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.
- [59] Richard Power. 1999 CSI/FBI computer crime and security survey. *Computer Security Journal*, Volume XV(2), 1999.
- [60] 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>.
- [61] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., January 1998.
- [62] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, January 1997.
- [63] RootShell. Web page at <http://www.rootshell.com>, 2000.
- [64] 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.
- [65] Scut and Team Teso. Exploiting format string vulnerabilities. Online document at <http://julianor.tripod.com/teso-fs1-1.pdf>, March 2001.
- [66] SecurityFocus. Web page at <http://www.securityfocus.com/>, 1999–2000.
- [67] Robert S. Sielken. Application intrusion detection. Technical Report CS-99-17, Department of Computer Science, University of Virginia, June 9 1999. URL

<ftp://ftp.cs.virginia.edu/pub/techreports/CS-99-17.ps>.Z.

- [68] Steven R. Snapp, James Brentano, Gihan V. Dias, Terrance L. Goan, L. Todd Heberlein, Che-lin Ho, Karl N. Levitt, Biswanath Mukherjee, Stephen E. Smaha, Tim Grance, Daniel M. Teal, and Doug Mansur. DIDS (distributed intrusion detection system) - motivation, architecture, and an early prototype. In *Proceedings of the 14th National Computer Security Conference*, pages 167–176, Washington, DC, October 1991. National Institute of Standards and Technology.
- [69] 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>.
- [70] 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>.
- [71] 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.
- [72] W. Richard Stevens. *TCP/IP Illustrated*, volume 2. Addison-Wesley, 1994.
- [73] *Df: report filesystem disk space usage*. Sun Microsystems, 2001. SunOS 5.7 manual page.
- [74] *Inetd: Internet services daemon*. Sun Microsystems, 2001. SunOS 5.7 manual page.
- [75] *Netstat: show network status*. Sun Microsystems, 2001. SunOS 5.7 manual page.
- [76] *Ping: send ICMP ECHO\_REQUEST packets to network hosts*. Sun Microsystems, 2001. SunOS 5.7 manual page.
- [77] *Ps: report process status*. Sun Microsystems, 2001. SunOS 5.7 manual page.
- [78] 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 unknown. URL <http://www.securityfocus.com/data/library/idsforensics.ps>.
- [79] 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, CA, USA, September 1992. USENIX.
- [80] Scott M. Wimer. Cylantsecure™: A scientific approach to security. Whitepaper, Cylant Technology, Inc., 2001. URL <http://www.cylant.com/whitepapers/cs-scientific.shtml>.
- [81] X-Force. Web page at <http://xforce.iss.net>, 2000.
- [82] Diego Zamboni. *Using Internal Sensors for Computer Intrusion Detection*. PhD thesis, Purdue University, West Lafayette, IN, August 2001.

## 12 Acknowledgments

Portions of this work were supported by the various sponsors of CERIAS; that support is gratefully acknowledged.

We would like to thank Christopher Telfer for performing the unsupervised tests described in Section 7.1.4.