Detecting Unknown Massive Mailing Viruses Using Proactive Methods

**** This is a preprint of a paper to appear in RAID 2004 conference ****
**** Please do not distribute ****

Ruiqi Hu and Aloysius K. Mok

Department of Computer Sciences,
University of Texas at Austin, Austin, Texas 78712, USA.
Email:{hurq,mok}@cs.utexas.edu

Abstract. The detection of unknown viruses is beyond the capability of many existing virus detection approaches. In this paper, we show how proactive customization of system behaviors can be used to improve the detection rate of unknown malicious executables. Two general proactive methods, behavior skewing and cordoning, and their application in BESIDES, a prototype system that detects unknown massive mailing viruses, are presented.

Keywords: virus detection; malicious executable detection; intrusion detection.

1 Introduction

Two major approaches employed by intrusion detection systems (IDSs) are misuse-based detection and anomaly-based detection. Because misuse-based detection relies on known signatures of intrusions, misuse-based IDSs cannot in general provide protection against new types of intrusions whose signatures are not yet cataloged. On the other hand, because anomaly-based IDSs rely on identifying deviations from the "normal" profile of the protected system's behavior, they are prone to reporting false-positives unless the normal profile of the protected system is well characterized.

In either case, detection must be performed as soon as an intrusion occurs and certainly before the intruder can cause harm and/or hide its own tracks. In this paper, we present the paradigm of PAIDS (ProActive Intrusion Detection System) and describe an application of PAIDS that can detect some classes of intrusions without knowing *a priori* their signatures and does so with very small false positives. PAIDS also provides a way to dynamically trade off the time it takes to detect an intrusion and the damage that an intruder can cause, and can therefore tolerate intrusions to an extent. To achieve these advantages, PAIDS exploits two general techniques: *behavior skewing* and *cordoning*.

Traditionally, the security of a computer system is captured by a set of security policies. A complete security policy should classify each behavior of a system

as either legal or illegal. In practice, however, specifications of security policies often fail to scale [1] and are likely to be incomplete. Given a security policy, we can instead partition the set of all behaviors of a system into three subsets: (S1) Legal behaviors, (S2) Illegal behaviors and (S3) Unspecified behaviors, corresponding to whether a behavior is consistent, inconsistent or independent of the security policy. (Alternatively, think of a security policy as the axioms of a theory for establishing the legality of system behaviors.) In this context, behavior skewing refers to the modification of a security policy P into P' such that the subset of legal behaviors remains unchanged under P', but some of the behaviors that are in the subset (S3) under P are illegal under P'. By implementing detectors that can recognize the enlarged subset (S2), behavior skewing can catch intruders that would otherwise be unnoticed under the unmodified policy P. The speed of detection depends on the length of the prefix of the illegal behavior that distinguishes it from the legal behaviors. To contain the damage caused by an intrusion, we seek ways to isolate the system components that are engaged in the illegal behavior, hopefully until we can distinguish it from the legal behaviors. By virtualizing the environment external to the system components that are engaged in the illegal behavior, cordoning prevents the system from permanent damage until the illegal behavior can be recognized.

In this paper, we illustrate an application of the PAIDS paradigm by BESIDES, a tool for detecting massive mailing viruses. We have applied behavior skewing and cordoning techniques to the NT-based Windows operating systems (Windows 2000 and Windows XP). Inasmuch as behavior skewing and cordoning are general techniques, we specialize them to certain types of system behaviors for efficient implementation. Specifically, we use behavior skewing to customize the security policy upon the use of certain information items on the protected system. An *information item* can be any logical entity that carries information, e.g., a filename, an email address, or a binary file. Behavior skewing is accomplished by customizing the access control mechanism that governs the access to the information items. In a similar vein, cordoning is applied to critical system resources whose integrity must be ensured to maintain system operability. Specifically, we provide mechanisms for dynamically isolating the interactions between a malicious process and a system resource so that any future interaction between them will not affect other legal processes' interaction with the resource. This is achieved by replacing the original resource with a virtual resource the first time a process accesses a system resource. The cordoning mechanism guarantees for each legal process that its updates to critical system resources are eventually committed to the actual system resources, If a malicious executable is detected, an execution environment (a cordon) consisting of virtual system resources is dynamically created for the malicious executable and its victims. Their previous updates to the actual system resources can be undone by performing recovery operations on those resources, while their future activities can be monitored and audited. Depending on the nature of the system resources, cordoning can be achieved through methods such as cordoning-in-time and cordoning-in-space.

The rest of this paper is organized as follows: Section 2 is a brief discussion of related works. Section 3 gives the details of how behavior skewing and cordoning work. Section 4 discusses the implementation of BESIDES, a prototype system we have implemented for detecting massive-mailing viruses. Section 5 presents the experiments we have performed with BESIDES and the analysis of the experimental results. Section 6 discusses some future directions.

2 Related Work

Virus detection is closely related to the more general topic of malicious executable detection [2, 3]. Traditional malicious executable detection solutions use signature-based methods, where signatures from known malicious executables are used to recognize attacks from them [4]. Security products such as virus scanners are examples of such applications. One of the focuses in signature-based methods research is the automatic generation of high quality signatures. Along this line of work, Kephart and Arnold developed a statistical method to extract virus signatures automatically [5]. Heuristic classifiers capable of generating signatures from a group of known viruses were studied in [6]. Recently, Schultz et al. examined how data mining methods can be applied to signature generation [7] and built a binary filter that can be integrated with email servers [8]. Although proved to be highly effective in detecting known malicious executables, signature-based methods are unable to detect unknown malicious executables. PAIDS explores the possibility of addressing the latter with a different set of methods, the proactive methods. In PAIDS, signatures of malicious behaviors are implicitly generated during the behavior skewing stage and they are later used in the behavior monitoring stage to detect malicious executables that perform such behaviors.

Static analysis techniques that verify programs for compliance with security properties have also been proposed for malicious executable detection. Some of them focus on the detection of suspicious symptoms: Biship and Dilger showed how file access race conditions can be detected dynamically [9]. Tesauro et al. used neural networks to detect boot sector viruses [10]. Another approach is to verify whether safe programming practices are followed. Lo et al. proposed to use "tell-tale signs" and "program slicing" for detecting malicious code [11]. These approaches are mainly used as preventive mechanisms and the approach used in PAIDS focuses more on the detection and tolerance of malicious executables.

Dynamic monitoring techniques such as sandboxes represent another approach that contributes to malicious executable detection. They essentially implement alternative reference monitoring mechanisms that observe software execution and enforce additional security policies when they detect violations [12]. The range of security policies that are enforcible through monitoring were studied in [13, 14, 15] and more general discussion on the use of security policies can be found in [1]. The behavior monitor mechanism used in PAIDS adopts a similar approach. Sandboxing is a general mechanism that enforces a security policy by executing processes in virtual environments (e.g., Tron [16], Janus

[17], Consh [18], Mapbox [19], SubDomain [20], and Systrace [21]). Cordoning is similar to a light-weight sandboxing mechanism whose coverage and time parameters are customizable. However, cordoning emphasizes the virtualization of individual system resources, while traditional sandboxing mechanisms focus on the virtualization of the entire execution environment (e.g., memory space) of individual processes. More importantly, sandboxing usually provides little tolerance toward intrusions, while cordoning can tolerate misuse of critical system resources as explained in Section 3.2.

Deception tools have long been used as an effective way of defeating malicious intruders. Among them, Honeypot (and later Honeynet) is a vulnerable system (or network) intentionally deployed to lure intruders' attentions. They are useful for studying the intruders' techniques and for assessing the efficacy of system security settings [22, 23]. Traditional Honeypots are dedicated systems that are configured the same way as (or less secure than, depending on how they are used) production systems so that the intruders have no direct way to tell the difference between the two. In the context of the Honeypot, no modification is ever performed on production systems. The latest advances such as virtual Honeypots [24, 25] that simulate physical Honeypots at the network level still remain the same in this regard. Recently, the concept of Honeypots was generalized to Honeytokens—"an information system resource whose value lies in unauthorized or illicit use of that resource" [26]. So far, few implementation or experimental results have been reported (among them, a Linux patch that implements Honeytoken files was made available early 2004 [27]). The Honeytoken concept comes the closest to PAIDS. However, the proactive methods that PAIDS explores, such as behavior skewing and cordoning, are more comprehensive and systematic than Honeytokens. We note that the implementation of our IDS tool BESIDES was well on its way when the Honeytoken concept first appeared in 2003.

System behavior modifications are gaining more and more interest recently. Somayaji and Forrest applied "benign responses" to abnormal system call sequences [28]. The intrusion prevention tool LaBrea is able to trap known intruders by delaying their communication attempts [29]. The virus throttles built by Williamson et al. [30, 31, 32] utilized the temporal locality found in normal email traffics and were able to slow down and identify massive mailing viruses as they made massive connection attempts. Their success in both intrusion prevention and toleration confirms the effectiveness of behavior modification based methods. However, the modifications performed in all these methods do not attempt to modify possibly legal behaviors since that may incur false positives, while the behavior skewing method in PAIDS takes a further step and converts some of the legal but irrelevant behaviors into illegal behaviors. No false positives are induced in this context.

Intrusion detection is a research area that has a long history [33, 34]. Traditionally, IDSs have been classified into following categories: Misuse-based IDSs look for signatures of known intrusions, such as a known operation sequence that allows unauthorized users to acquire administrator privileges [35, 36]. This is very similar to the signature-based methods discussed earlier in this section.

Anomaly-based IDSs detect intrusions by identifying deviations from normal profiles in system and network activities. These normal profiles can be studied from historical auditing data that are collected during legitimate system executions [34, 37, 38, 39]. Specification-based IDSs look for signatures of known legitimate activities that are derived from system specifications [40]. These IDSs differ from misused-based IDSs in that activities that do not match any signature are treated as intrusions in the former, but are regarded as legitimate ones in the latter. Hybrid IDSs integrate different types of IDSs as subsystems [41]. The limitations of different subsystems can be compensated and a better overall quality can be improved.

Many techniques devised in IDSs are applicable to malicious executable detection and vice versa. For example, in an effort to apply the specification-based approach to malicious executable detection, Giffin et al. showed how malicious manipulations that originated from mobile code can be detected [42]. Christodorescu and Jha proposed an architecture that detects malicious binary executables even in the presence of obfuscations [43]. One common goal of all IDSs (anomaly-based IDSs in particular) is to generate a profile using either explicit or implicit properties of the system that can effectively differentiate intrusive behaviors from normal behaviors. A wide variety of system properties, ranging from low-level networking traffic statistics and system call characteristics to high-level web access patterns and system resource usages, have been studied in literature. McHugh and Gates reported a rich set of localities observed in web traffic and pointed out such localities can be used to distinguish abnormal behaviors from normal behaviors [44]. None of them considers the possibility of modifying system security settings for intrusion detection purposes, which is explored in PAIDS. The proactive methods proposed in PAIDS modify the definition of normal behaviors through skewing the security policy in anticipation of the likely behavior of intruders. By doing so, PAIDS implicitly generates profiles that have small false positive rates, are easy to understand and configure, and are highly extensive.

3 Methodology

3.1 Behavior Skewing

As illustrated in Fig.1, behaviors are high-level abstractions of system activities that specify their intentions and effects, but ignore many of the implementation details of the actual operations they perform. We refer to the part of security assignments of behaviors that are explicitly specified to be legal (or illegal) as the *legal behavior set* (LBS) (or the *illegal behavior set* (IBS)). The security assignments of behaviors that are not explicitly specified either intentionally or unintentionally are denoted as the *unspecified behavior set* (UBS). The UBS consists of behaviors that either 1) the user considers to be irrelevant to her system's security or 2) the user is unaware of and unintentionally fails to specify their security assignments; the default security policy will apply to the behaviors in the set UBS. *Behavior skewing* refers to the manipulation of the security policy

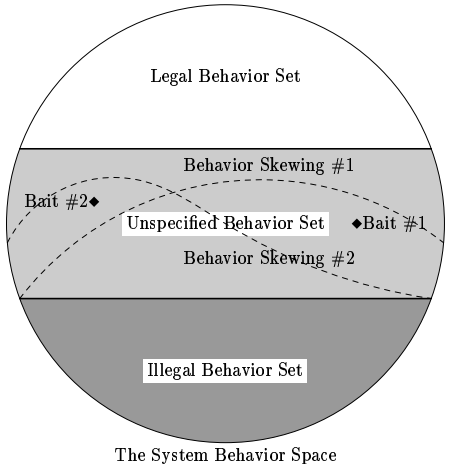


Fig. 1. System States and Behavior Skewing

that customizes the security assignments of the behaviors in the set UBS that are implicitly assigned to be legal by default. The goal of behavior skewing is to create enough differences in the legal behaviors among homogeneous systems so that malicious executables are made prone to detection.

Specifically, behavior skewing customizes a security policy regarding the use of certain information items in a system. Behavior skewing creates its own access control mechanism that describes the use of information items since many of the information items are not system resources and are thus not protected by native security mechanisms. The customization is performed on *information domain*s, which are sets of information items that are syntactically equivalent but semantically different. For example, all text files in a target system form an information domain of text files. Specifically, behavior skewing reduces the access rights to existing items that are specified by the default access rights and creates new information items in an information domain with reduced access rights.

Figure 1 shows two possible behavior skewing instances. We emphasize that although different skewing mechanisms produce different security settings, they all preserve the same LBS. The modified security policy generated by a behavior skewing is called the *skewed security policy* (or *skewed policy* in short). The default security policy is transparent to the user and cannot be relied upon by the user to specify her intentions. Otherwise, additional conflict resolution mechanisms may be required.

After behavior skewing is completed, the usage of information items is monitored by a *behavior monitoring* mechanism that detects violations of the skewed policy. Any violation of the skewed policy triggers an intrusion alert. It should be noted that the monitoring mechanism does not enforce the skewed policy,

instead it simply reports violations of the skewed policy to a higher-level entity (e.g., the user or an IDS) that is responsible for handling the reported violations.

3.2 Cordoning

Although behavior skewing and monitoring make malicious executables more prone to detection, actual detection cannot happen before the malicious executables have their chance to misbehave. Hence, there is a need for additional protection mechanisms to cope with any damage the malicious executables may incur before they are eventually detected. Among them, the recovery of system states is an obvious concern. In this section, we illustrate another proactive method, *cordoning*, that can be used to recover states of selected system resources. Existing system recovery solutions, such as restoring from backup media or from revertible file systems, usually perform bulk recovery operations, where the latest updates to individual system resources containing the most recent work of a user may get lost after the recovery. Cordoning addresses this problem by performing the recovery operation individually.

In general, cordoning is a mechanism that allows dynamic partial virtualization of execution environments for processes in a system. It is performed on *critical system resources* (CSRs), objects in the system whose safety are deemed critical to the system's integrity and availability (e.g., executables, network services, and data files, etc.). The cordoning mechanism converts a CSR (also called an *actual* CSR) to a *recoverable* CSR (or a *cordoned* CSR) that can be recovered to a known safe state by dynamically creating a virtual CSR (called the *current* CSR), and granting it to processes that intend to interact with the actual CSR. The current CSR provides the same interface as the actual CSR. The underlying cordoning mechanism ensures all updates to the current CSR are eventually applied to the actual CSR during normal system execution.

When a malicious executable is detected by the behavior monitor, its updates to all cordoned CSRs can be recovered by performing the corresponding *recover* operations on the cordoned CSRs that restore the actual CSRs to known secure states (called *recovered* CSRs). The malicious executable and its *victims*—its children, as well as processes accessing system resources that have been updated by the malicious executable[1]—continue to interact with the *current CSR*s, the same set of CSRs they are using at the time of detection. However, their future updates to this set of current CSRs will not be applied to the actual CSRs and are instead subject to audition and other intrusion investigation mechanisms. *Unaffected process*es—existing processes other than the malicious executable and its victims, and all newly started processes—will use the recovered CSRs in their future interactions. The malicious executable and its victims are thus *cordoned* by a dynamically created execution environment (called a *cordon*) that consists of these current CSRs. The operations performed by the malicious executable

[1] We note that an accurate identification of victims can be as difficult as the identification of covert channels. The simple criteria we use here is a reasonable approximation although more accurate algorithms can be devised.

and its victims on CSRs are thus isolated from those performed by the unaffected processes.

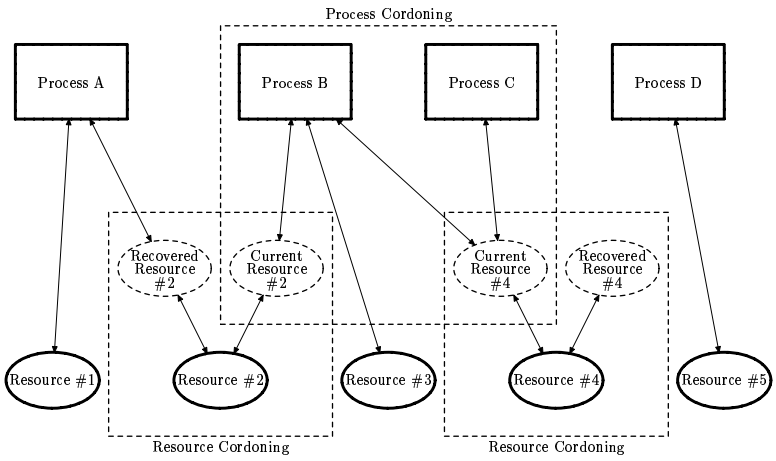


Fig. 2. A Cordoning Example

CSRs can be classified as revertible CSRs, delayable CSRs, and substitutable CSRs based on their nature. Two cordoning mechanism: cordoning-in-time and cordoning-in-space can be applied to these CSRs as described below:

A *revertible* CSR is a CSR whose updates can be revoked. For example, a file in a journaling file system can be restored to a previous secure state if it is found corrupted during a virus infection. Cordoning of a revertible CSR can be achieved by generating a *revocation list* consisting of revocation operations of all committed updates. The recovery of a revertible resource can be performed by carrying the revocation operations in the revocation list that leads to a secure state of the CSR.

Cordoning-in-time buffers operational requests on a CSR and delay their commitments until a secure state of the resource can be reached. It is thus also referred to as *delayed commitment*. Cordoning-in-time can be applied to *delayable* CSRs—CSRs that can tolerate certain delays when being accessed. For example, the network resource that serves a network transaction is delayable if the transaction is not a real-time transaction (e.g., a SMTP server). The delays on the requests can be made arbitrarily long unless it exceeds some transaction-specific time-out value. Time-outs constraint the maximum length of delays. Such a constraint, as well as others (e.g., constraints due to resource availability) may render a delayable CSR a *partially recoverable* CSR; the latter can only be recovered within a limited time interval. The recovery of a delayable resource can be performed by discarding buffered requests up to a secure state. If no such secure state can be found after exhausting all buffered requests, the CSR may not be securely recovered unless it is also revertible.

Cordoning-in-space is applied to a *substitutable* CSR—a CSR that can be replaced by another CSR (called its *substitute*) of the same type transparently. For example, a file is a substitutable CSR because any operation performed on a file can be redirected to a copy of that file. Cordoning-in-space redirects operational requests from a process toward a substitutable CSR to its substitute. The actual CSR is kept in secure states during the system execution and is updated by copying the content of its substitute only when the latter is in secure states. A substitutable CSR can be recovered by replacing it with a copy of the actual CSR saved when it is in a secure state. Where multiple substitutes exist for a CSR (e.g., multiple writer processes), further conflict resolution is required but will not be covered in this paper. We note here that the cordoning and recovery of CSRs are independent mechanisms that can be separately performed.

Figure 2 shows an example of cordoning on two CSRs: Resource #2 and Resource #4. Process B and Process C are identified as victims of a virus infection. They are placed in a cordon consisting of the Current Resource #2 and the Current Resource #4. The cordoning mechanism performs the recovery operations, where the Recovered Resource #2 and the Recovered Resource #4 are created. Unaffected processes that shares CSRs with the victims, e.g., Process A, continue to use Recovered Resource #2. New processes such as Process D will be given the Recovered Resource #4 when it requests to access Resource #4 in the future. The remaining system resources, such as Resource #1, #3, and #5, are not cordoned.

4 Implementation

BESIDES is a prototype system that uses behavior skewing, behavior monitoring, and cordoning to detect unknown massive mailing viruses. As illustrated in Figure 3, BESIDES consists of an *email address domain skewer (EADS)*, an *email address usage monitor (EAUM)* and a *SMTP Server Cordoner (SSC)*. The EADS is a user-mode application and the EAUM and the SSC are encapsulated in a kernel-mode driver. BESIDES is implemented on Windows 2000 and also runs on Windows XP. The main reason we choose to build a Windows based system is that Windows is one of the most attacked systems and virus attacks in particular are frequently seen there. Developing and experimenting BESIDES on Windows allows us to illustrate our approach and to show that it can be implemented on commercial operating systems.

4.1 The Email Address Domain Skewer

After BESIDES is installed, the user needs to use the EADS to skew the behavior of the email address domain, an information domain that is actively exploited by malicious executables such as massive mailing viruses. The skewing is performed by modifying the usage policy of email addresses. A typical Windows system does not restrict the use of email addresses. In particular, all email addresses can be used as email senders or recipients. The EADS changes this default usage

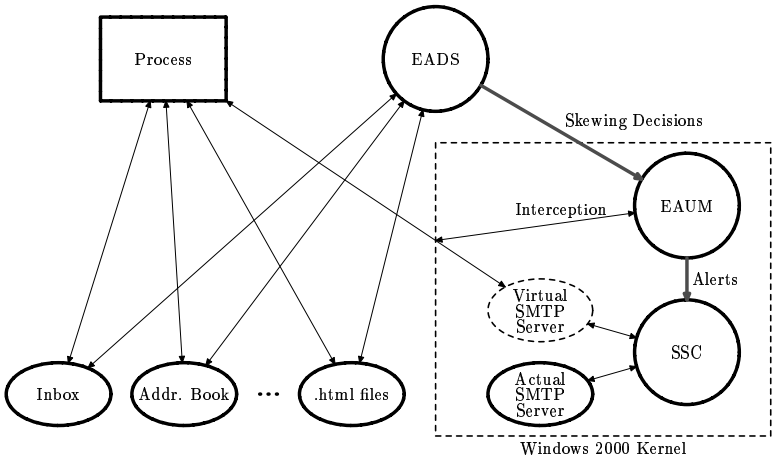


Fig. 3. The BESIDES Architecture

policy by making certain email addresses unusable in any locally composed email. The affected email addresses are referred to as *baiting address*es, or simply *bait*s. The existing email addresses are not skewed unless the user is able to determine which of them is skewable. By default, the EADS grants all subjects full usage (i.e., "allow all") of any email address except the baits. It deems any use of baits a violation of the skewed email address usage policy (i.e., "deny all") The EAUM will issue an intrusion alert whenever it detects the use of a bait in any email messages sent locally. In addition, the EADS sets access rights to certain email domains (e.g., `foo.com` in `alice@foo.com`) in baits to "deny all" as well. This makes the skewing more versatile so that even viruses manipulating email addresses before using them (e.g., Bugbear) can also be detected.

The skewing of email address domain requires the creation of enough unpredictability in the domain so that viruses are not likely to figure out whether an email address is legitimate by simply looking at the email address itself. The EADS uses both heuristic methods and randomization methods in its skewing procedure, i.e., it generates the baiting addresses either based on a set of baits the user specifies or in random.

Specifically, the EADS creates email baits in following file types: email boxes, e.g. `.eml` files; text-based files, e.g. `.HTM`, `.TXT`, and `.ASP` files, etc.; and binary files, e.g., `.MP3`, `.JPG`, and `.PDF` files, etc. Email boxes are skewed by importing baiting email messages that use baiting addresses as senders. Text-based files are skewed with newly created syntactically valid files that contain baits. Binary files are skewed with the same text files as in text-based file skewing but with modified extensions. Such baiting binary files have invalid format and cannot be processed by legitimate applications that operate on these files. This does not pose a problem because these baits are not supposed to be used by them in the first place. Many massive mailing viruses, however, are still able to discover the baits within these files because they usually access them by "brute-force" and

neglect their formats, e.g., by searching for email addresses with straightforward string matching algorithms. By default, all these baits are placed in commonly accessed system directories such as "`C:\`", and "`My Documents`". The user is allowed to pick additional directories she prefers.

4.2 The Email Address Usage Monitor

BESIDES uses system call interposition to implement the EAUM (and the SSC as well). System call interposition is a general technique that allows interception of system call invocations and has been widely used in many places [45, 46, 20, 47, 19, 17, 48]. Windows 2000 has two sets of system calls [49]: The *Win32* application programming interfaces (APIs), the standard API for Windows applications, are implemented as user-mode dynamically linked libraries (DLLs). The *native API*s (or *native system call*s) are implemented in kernel and are exported to user-mode modules through dummy user-mode function thunks in `ntdll.dll`. User-mode system DLLs use native APIs to implement Win32 APIs. A Win32 API is either implemented within user-mode DLLs or mapped to one (or a series of) native API(s). Windows 2000 implements the TCP/IP protocol stack inside the kernel [50]. At the native system call interface, application level protocol behaviors are directly observable and can thus be efficiently monitored. Transport level protocol specific data (e.g., TCP/UDP headers) are not present at this interface. This saves the additional time needed to parse them, analyze them, and then reconstruct protocol data flow states that is unavoidable in typical network based interceptors.

The EAUM runs in kernel mode and monitors the use of email addresses in SMTP sessions. An *SMTP session* consists of all SMTP traffic between a process and an SMTP server. It starts with a "`HELO`"(or "`EHLO`") command and usually ends with a "`QUIT`" command. The EAUM registers a set of system call filters to the system call interposition module (also referred to as the *BESIDES engine*) during BESIDES initialization when the system boots up. It uses an SMTP automaton derived from the SMTP protocol [51] to simulate the progresses in both the local SMTP client and the remote SMTP server. The SMTP automaton in BESIDES is shared among all SMTP sessions. The BESIDES engine intercepts native system calls that transmit network data and invokes the system call filters registered by the EAUM. These filters extract SMTP data from network traffic, parse them to generate SMTP tokens, and then perform state transitions in the SMTP automaton that monitors the corresponding SMTP session. Each SMTP session is represented in the EAUM by a SMTP context, which is passed to the SMTP automaton as a parameter each time that particular SMTP session makes progress. The use of email addresses in a SMTP session can be monitored when the SMTP automaton enters corresponding states. Specifically, the EAUM looks for SMTP commands that explicitly uses email addresses (i.e., "`MAIL FROM:`" and "`RCPT TO:`") and validates these usage against the skewed email address usage policy specified by the EADS. If any violation is detected, the EAUM notifies the SSC with an intrusion alert because none of the baiting addresses should be used as either a recipient or sender. The use of legitimate

email addresses does not trigger any alert because their usage is allowed by the EADS. One advantage of this monitoring approach is that viruses that carry their own SMTP clients are subject to detection, while interpositions at higher levels (e.g., Win32 API wrappers) are bypassable. Misuse detection mechanisms in the form of wrappers around SMTP servers are not used since viruses may choose open SMTP relays that are not locally administrated.

4.3 The SMTP Server Cordoner

In addition to detecting massive mailing viruses, BESIDES also attempts to protect CSRs (here, SMTP servers) from possible abuses from them. A SMTP server is a delayable CSR since emails are not considered a real-time communication mechanism and email users can tolerate certain amount of delays. It is also weakly revertible (by this we mean the damage of a delivered message containing a virus can be mollified by sending a follow-up warning message to the recipient when the virus is later detected). Whenever a SMTP session is started by a process, the SSC identifies the SMTP server it requests and assigns it the corresponding virtual SMTP server (the current SMTP server). Delayed-commitment is then used to buffer the SMTP messages the process send to the virtual SMTP server. The SSC also runs in kernel-mode and shares the same SMTP automaton with the EAUM.

Specifically, the SSC intercepts SMTP commands and buffers them internally. It then composes a positive reply and has the BESIDES engine forward it to the SMTP client indicating the success of the command. After receiving such a reply, the SMTP client will consider the previous command successful and proceeds with the next SMTP command. The SSC essentially creates a virtual SMTP server for the process to interact with. The maximum time a SMTP message can be delayed is determined by the cordoning period—a user specified time-out value that is smaller than the average user tolerable delays, as well as the user specified threshold on the maximum number of delayed messages. A SMTP message is delivered (committed) to the actual SMTP server when either it is delayed more than the cordoning period or the number of delayed messages exceeds the message number threshold. After delivering a SMTP message, the SSC creates a corresponding log entry (a revocation record) in the SMTP server specific log containing the time, subject, and the recipient of the delivered message. When informed of an intrusion alert, the SSC identifies the process that is performing the malicious activity be the malicious executable. It then determines the set of victims based on the CSR access history and process hierarchy, i.e, all processes that access CSRs updated by this process and all its child processes are labeled as victims. After this, the SSC initiates the recovery operations on all cordoned CSRs they have updated. If the process that owns a SMTP session is one of the victims or the malicious executable itself, no buffered messages from that SMTP session is committed; instead they are all quarantined. All messages that are previously committed are regarded as suspicious and the SSC sends a warning message for their recipients as a weak recovery mechanism using the information saved in the delivery log entries. Since the SMTP messages sent to

Virus		BugBear	Haptime	Klez	MyDoom
Client	Detected?	Yes	Yes	Yes	Yes
	Baits Used at Detection	Addr. Book	`.htm`	`.html`	`.htm`
	Delayed Message Quarantined?	Yes	Yes	Yes	Yes
Server	Detected?(by anti-virus software)	Yes	No	Yes	No
	SMTP Message Received?	Yes	No	No	No

Table 1. Effectiveness of BESIDES (The BESIDES SSC is configured to intercept at most 10 total SMTP messages and for at most 60 seconds for each SMTP message during these experiments. The Outlook Express book is manually skewed.)

a SMTP server are independent, the order they are received does not affect the correct operation of the SMTP server [52]. Thus the actual SMTP server can be kept in a secure state even if some of the messages are dropped during the recovery operation. In the mean time, the unaffected processes are unaware of this recovery and can proceed as if no intrusion has occurred.

5 Experimental Results

We performed a series of experiments on BESIDES with viruses we collected in the wild. These experiments are performed on a closed local area network (LAN) consisting of two machines: a server and a client. BESIDES is installed on the client and its EADS is set up the same way in all experiments. The server simulates a typical network environment to the client by providing essential network services, such as DNS, Routing, SMTP, POP3, and Remote Access Service (RAS). The server is also equipped with anti-virus software (i.e., Symantec Antivirus) and network forensic tools (e.g., Ethereal, TcpDump, etc.). Evidences of virus propagation can be gathered from output of these tools as well as service logs.

Two sets of experimental results are presented in the remainder of this section. First we present the outcome when BESIDES is experimented with several actual viruses. These results demonstrate the effectiveness of behavior skewing and cordoning when they are applied in a real-world setting. The second set of results presents the performance overheads observed during normal system execution for normal system applications, including delays observed at the native system call interfaces, and those at the command-line interface. As we have expected, the overheads are within a reasonable range even though we have not perform any optimization in BESIDES.

5.1 Effectiveness Experiments

The results of our experiments with four actual viruses—BugBear [53], Haptime [54], Klez [55], and MyDoom [56]—are shown in Table 1. In all the experiments, BESIDES were able to detect the viruses being experimented. Although all these viruses attempted to collect email addresses on the client machine, their

methods were different and the actual baiting addresses they were using when BESIDES detected them also differed from each other. In all experiments, the BESIDES SSC intercepted multiple SMTP messages sent by viruses and successfully quarantined them. However, some of the virus carrying messages were found delivered before the virus was detected during the experiment with Bugbear. From the email messages received by the SMTP server from Bugbear, we found Bugbear actually manipulated either the sender or the recipient addresses before sending them. Specifically, it forms a new email address by combining the user field of one email address and the domain field of another email address. BESIDES was initially unable to detect these messages since it only considers the matches of both the user name field and the domain field as an acceptable match to a bait. It then committed these messages to the SMTP server[2]. With our experimental setups, an average two out of ten such messages were found committed to the SMTP server. We note that the actual numbers varies with the skewing manipulations. In one of our experiments, two such messages were thus committed before BESIDES detected the virus. The anti-virus software on the server detected Bugbear and Klez because they also spread over network shares, a mechanism that is not cordoned by BESIDES in all these experiments.

We observed significant hard disk accesses from Haptime when it tried to infect local files and collecting email addresses from them. All these happened before the virus start to perform massive mailing operations. This suggested that Haptime can be detected faster if BESIDES skews file access rights as well.

The outbreak of Mydoom was later than the version of BESIDES used in the experiments was completed. Thus BESIDES had no knowledge of the virus when it was experimented with it. BESIDES successfully detected the virus when it tried to send messages using baiting email addresses placed in `.htm` files. This demonstrated BESIDES's capability in detecting unknown viruses.

As some of the viruses use probabilistic methods, (e.g., Klez,) their behavior in different experiments can be different. The result shown here is thus only one possible behavior. Also, as some of the viruses use local time to decide whether particular operations are (or are not) to be performed (e.g., MyDoom does not perform massive mailing if it performs DDoS attacks, which is dependent on the system time.), we manually changed the client's system time to hasten the activation of massive mailing by the virus. We emphasize that this is done to speed up our experiments and that system time manipulation is not needed to effect detection in production.

5.2 Performance Experiments

Overall System Overheads Table 2 shows statistical results of the system call overheads observed at three native system call interface during normal system executions. Two interceptors, the pre-interceptor and the post-interceptor, are used to perform interception operations before and after the actual system call

[2] This loophole is later fixed by creating additional email usage policy on email domains and creating baiting domains in the EADS. See Section 4.1 for details.

Native System Call	NtClose()	NtCreateFile()	NtDeviceIoControlFile()
Total Execution Time	283076	1997247	16110683
Pre-Interceptor Time	108520	24033	21748
Actual System Call Time	32960	1471367	15791127
Post-Interceptor Time	23588	371826	156350
System Call Interposition Time	118008	130021	141457
Overhead (in %)	758.85%	35.74%	2.02%

Table 2. BESIDES system call overheads observed at native system call interface. All numbers are calculated from CPU performance counter values directly retrieved from the client machine (Pentium III 730MHz CPU with 128MB memory).

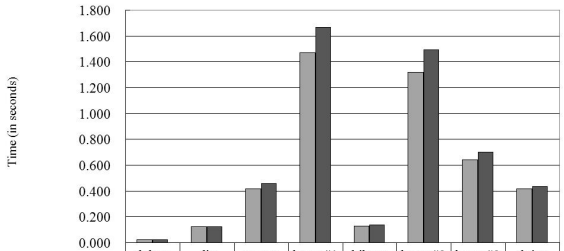
is executed. The three native system calls shown in the table are representatives of three cordoning session phases.

`NtCreateFile()` is invoked to create or open a file (including a socket) [49]. BESIDES intercepts it so that SMTP sessions can be recognized and their corresponding SMTP contexts can be created. It is the *setup phase* of a SMTP session. The post-interceptor is used to perform these operations. As can be seen from Tab.2, the overhead is a fraction of actual system call.

`NtDeviceIoControlFile()` performs an I/O control operation on a file object that represents a device [49]. Network packets are also transmitted using this system call. BESIDES intercepts this system call so that it can inspect SMTP session control and data messages. This is the *inspection phase* of a SMTP session. During the interception, SMTP data are parsed, SMTP tokens are generated, and the BESIDES EAUM SMTP automaton's state is updated. The pre-interceptor and the post-interceptor processes sent data and received data, respectively. The overhead observed is only a small percentage of the actual system call because the actual system call incurs expensive hardware I/O operations.

`NtClose()` is invoked by a user-mode process to close a handle to an object [49]. BESIDES intercepts it to terminate a SMTP session. This is called the *termination phase* of that SMTP session. The pre-interceptor releases system resources that are used to monitor this SMTP session. The actual system call is a lightweight system call and it takes much less time than the other two. The overhead observed dominates the actual system call. However, as both setup and termination phases are only performed once during a SMTP session's lifespan, their relatively high cost can be amortized by the much faster inspection phase.

Finally, it should be noted that a different type of overhead, the system call interposition overhead, exists in all system call interceptions. This overhead accounts for the mandatory overhead each intercepted system call has to pay, including extra system call lookup time, and kernel-stack setup time, etc. However, for those system calls that are not intercepted, optimized shortcuts are created in interception routines so that as little overhead as possible is generated.

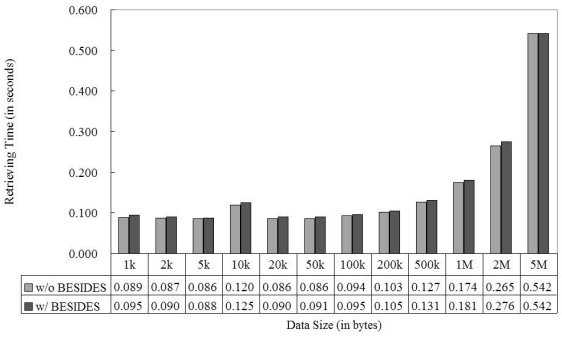


	delete	dir	mpost	latex #1	bibtex	latex #2	latex #3	dvips
w/o BESIDES	0.021	0.122	0.417	1.471	0.127	1.318	0.641	0.418
w/ BESIDES	0.022	0.124	0.456	1.666	0.135	1.492	0.702	0.436

Fig. 4. Time overhead for the latex application series

Application Specific Overheads We also measured the time overhead for several applications on the client machine where BESIDES is installed. Figure 4 shows the overheads (average values of 10 separate runs) observed on a series of applications that compile the postscript version of a draft of this paper from its `.tex` source files. The applications executed include delete (cleaning up the directory), dir (listing the directory content), mpost (building graphic `.eps` files), latex #1 (The first run of `latex`), bibtex (preparing bibliography items), latex #2 (The second run of `latex`), latex #3 (The third run of `latex`), and dvips (converting the `.dvi` file to the postscript file). Both CPU intensive and I/O intensive applications are present and this series can be regarded as a representative of applications that require no network access. These applications are only affected by the native system call interposition overhead induced by the BESIDES engine. The average time overhead observed in these experiments is around 8%. The highest increases (around 13%) occur in latex #1 and latex #2, both of which perform significant I/O operations. The lowest increases (around 1.5% and 3.3%) occur in dir and delete respectively. These are shell commands that perform simple tasks that require few system calls. We note that CPU intensive applications, e.g., dvips, suffer much smaller overheads (e.g., 4.3% for dvips). These results indicate that I/O intensive applications are likely to endure more overhead than CPU intensive applications. They conform to our expectation as essentially all I/O operations are carried out by native system calls and are thus subject to interception, while CPU intensive operations contain higher percentage of user-mode code that makes few system calls.

Figure 5 shows the overheads observed for a command-line based web client when it retrieves data files from the local web server. The sizes of data files used in this experiment range from 1kB to 5MB. Although the web client retrieves these data files using HTTP, its network traffic is still subject to inspection of the SMTP filters in BESIDES. The system call interposition overhead is relatively small since the web client performs few other system calls during the whole process. The two largest overheads observed are around 6.3% and 5% (at 1kB and 50kB, respectively). The two smallest overheads are 0% and 1.8% (at 5MB



	1k	2k	5k	10k	20k	50k	100k	200k	500k	1M	2M	5M
w/o BESIDES	0.089	0.087	0.086	0.120	0.086	0.086	0.094	0.103	0.127	0.174	0.265	0.542
w/ BESIDES	0.095	0.090	0.088	0.125	0.090	0.091	0.095	0.105	0.131	0.181	0.276	0.542

Fig. 5. Time overhead for the command-line web client

and 100kB, respectively). The average overhead is around 3.4%, which is close to 2.02%, the overhead observed at the `NtDeviceIoControlFile()` interface. This confirms our previous speculation that the seemingly high increase in the session setup phase and the session termination phase can be amortized by the low overhead of the session inspection phase.

6 Conclusions

This paper presents a general paradigm, PAIDS for intrusion detection and tolerance by proactive methods. We present our work on behavior skewing and cordoning, two proactive methods that can be used to create unpredictability in a system so that unknown malicious executables are more prone to be detected. This approach differs from existing ones in that such a proactive system anticipates the attacks from malicious executables and prepares itself for them in advance by modifying the security policy of a system.

PAIDS enjoys the advantage that it can detect intruders that have not been seen yet in the raw and yet PAIDS has a very low false-positive rate of detection. BESIDES is a proof-of-concept prototype using the PAIDS approach, and it can be enhanced in many directions. Obvious enhancements include skewers and cordoners for additional information domains (e.g., file access skewer) and system resources (e.g., file system cordoner). The BESIDES SSC can be augmented with more versatile handling and recovery schemes to cope with general malicious executables. We are also interested in devising more proactive methods. In general, we want to investigate to what extent we can systematically cordon off parts or even all of a system by cordoning all the protocols they use to interact with the external environment.

Finally, we would like to point out that the proactive methods we have studied are only part of the solution to the general problem of detecting unknown malicious executables. A system that is equipped with only proactive techniques are still vulnerable to new types of malicious executables that do not misuse

any of the skewed information domains or abuse the system in more subtle ways such as stealing CPU cycles from legitimate applications. PAIDS is not a cure-all in that it works only for viruses' whose route of spreading infection or damage-causing mechanism is well characterized. A comprehensive solution that consists of techniques from different areas is obviously more effective because the weaknesses of each individual technique can be compensated by the strength of others. We would like to explore how proactive methods can be integrated with such hybrid solutions.

References

[1] Blakley, B.: The Emperor's Old Armor. In: Proceedings of the ACM New Security Paradigms Workshop, Lake Arrowhead, California (1996)

[2] Cohen, F.: Computer Viruses: Theory and Experiments. Computers and Security **6** (1987) 22–35

[3] Chess, D.M., White, S.R.: An Undetectable Computer Virus. In: Proceedings of the 2000 Virus Bulletin International Conference, Orlando, Florida (2000)

[4] Gryaznov, D.: Scanners of the Year 2000: Heuristics. In: Proceedings of the 5th Virus Bulletin International Conference, Boston, Massachusetts (1999) 225–234

[5] Kephart, J.O., Arnold, W.C.: Automatic Extraction of Computer Virus Signatures. In: Proceedings of the 4th Virus Bulletin International Conference, Abingdon, England (1994) 178–184

[6] Arnold, W., Tesauro, G.: Automatically Generated Win32 Heuristic Virus Detection. In: Proceedings of the 2000 Virus Bulletin International Conference, Orlando, Florida (2000)

[7] Schultz, M.G., Eskin, E., Zadok, E., Stolfo, S.J.: Data Mining Methods for Detection of New Malicious Executables. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, Oakland, California (2001) 38–49

[8] Schultz, M.G., Eskin, E., Zadok, E., Bhattacharyya, M., Stolfo, S.J.: MEF: Malicious Email Filter — A UNIX Mail Filter that Detects Malicious Windows Executables. In: Proceedings of the Annual USENIX Technical Conference, FREENIX Track, Boston, Massechusetts (2001) 245–252

[9] Bishop, M., Dilger, M.: Checking for Race Conditions in File Accesses. Computing Systems **9** (1996) 131–152

[10] Tesauro, G., Kephart, J., Sorkin, G.: Neural Networks for Computer Virus Recognition. IEEE Expert **11** (1996) 5–6

[11] Lo, R.W., Levitt, K.N., Olsson, R.A.: MCF: A Malicious Code Filter. Computers and Security **14** (1995) 541–566

[12] Anderson, J.P.: Computer Security Technology Planning Study. Technical report, ESD-TR-73-51, U.S. Air Force Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), Bedford, Massachusetts (1972)

[13] Viswanathan, M.: Foundations for the Run-time Analysis of Software Systems. PhD thesis, University of Pennsylvania (2000)

[14] Hamlen, K.W., Morrisett, G., Schneider, F.B.: Computability Classes for Enforcement Mechanisms. Technical report, TR 2003-1908, Cornell University, Dept. of Computer Science (2003)

[15] Bauer, L., Ligatti, J., Walker, D.: More Enforceable Security Policies. In: Foundations of Computer Security, Copenhagen, Denmark (2002)

[16] Berman, A., Bourassa, V., Selberg, E.: TRON: Process-Specific File Protection for the UNIX Operating System. In: Proceedings of the 1995 Annual USENIX Technical Conference, New Orleans, Lousianna (1995) 165–175
[17] Goldberg, I., Wagner, D., Thomas, R., Brewer, E.A.: A Secure Environment for Untrusted Helper Applications. In: Proceedings of the 6th USENIX Security Symposium, San Jose, California (1996)
[18] Alexandrov, A., Kmiec, P., Schauser, K.: Consh: A Confined Execution Environment for Internet Computations. In: Proceedinges of the Annual USENIX Technical Conference, New Orleans, Louisianna (1998)
[19] Acharya, A., Raje, M.: MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Application. In: Proceedings of the 9th USENIX Security Symposium, Denver, Colorado (2000)
[20] Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P., Gligor, V.: Subdomain: Parsimonious Server Security. In: Proceedings of the 14th Systems Administration Conference, New Orleans, Louisianna (2000)
[21] Provos, N.: Improving Host Security with System Call Policies. In: Proceedings of the 12th USENIX Security Symposium, Washington, DC (2003) 257–272
[22] Honeypots, Intrusion Detection, Incident Response: http://www.honeypots.net/.
[23] The Honeynet Project: http://project.honeynet.org/.
[24] Provos, N.: A Virtual Honeypot Framework. In: Proceedings of the 13th USENIX Security Symposium, San Diego, California (2004)
[25] Jiang, X., Xu, D.: Collapsar: A VM-Based Architecture for Network Attack Detention Center. In: Proceedings of the 13th USENIX Security Symposium, San Diego, California (2004)
[26] Spitzner, L.: Honeytokens: The Other Honeypot (2003) http://www.securityfocus.com/infocus/1713.
[27] Pontz, B.: Honeytoken. CERT Honeypot Archive (2004) http://cert.uni-stuttgart.de/archive/honeypots/2004/01/msg00059.html.
[28] Somayaji, A., Forrest, S.: Automated Response Using System-Call Delays. In: Proceedings of the 9th USENIX Security Symposium, Denver, Colorado (2000)
[29] LaBrea Sentry IPS: Next Generation Intrusion Prevention System: http://www.labreatechnologies.com/.
[30] Williamson, M.M.: Throttling Viruses: Restricting propagation to defeat malicious mobile code. In: Proceedings of the 18th Annual Computer Security Applications Conference, Las Vegas, Nevada (2002)
[31] Twycross, J., Williamson, M.M.: Implementing and Testing a Virus Throttle. In: Proceedings of the 12th USENIX Security Symposium, Washington, DC (2003)
[32] Williamson, M.M.: Design, Implementation and Test of an Email Virus Throttle. In: Proceedings of the 19th Annual Computer Security Applications Conference, Las Vegas, Nevada (2003)
[33] Anderson, J.P.: Computer Security Threat Monitoring and Surveillance. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania (1980)
[34] Denning, D.E.: An Intrusion–Detection Model. In: IEEE Transactions on Software Engineering. Volume SE-13:(2). (1987) 222–232
[35] Porras, P.A., Kemmerer, R.A.: Penetration State Transition Analysis – A Rule-Based Intrusion Detection Approach. In: 8th Annual Computer Security Applications Conference, San Antonio, Texas (1992) 220–229
[36] Kumar, S.: Classification and Detection of Computer Intrusions. PhD thesis, Purdue University (1995)
[37] Lunt, T.F.: Detecting Intruders in Computer Systems. In: Proceedings of the Conference on Auditing and Computer Technology. (1993)

[38] Javitz, H.S., Valdes, A.: The NIDES Statistical Component: Description and Justification. Technical report, SRI International, Computer Science Laboratory, Menlo Park, California (1993)
[39] Anderson, D., Lunt, T.F., Javitz, H., Tamaru, A., Valdes, A.: Detecting Unusual Program Behavior Using the Statistical Components of NIDES. Technical report, SRI-CSL-95-06, SRI International, Computer Science Laboratory, Menlo Park, California (1995)
[40] Wagner, D., Dean, D.: Intrusion Detection via Static Analysis. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, Oakland, California (2001) 156–169
[41] Neumann, P.G., Porras, P.A.: Experience with EMERALD to Date. In: Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring, Santa Clara, California (1999) 73–80
[42] Giffin, J.T., Jha, S., Miller, B.P.: Detecting Manipulated Remote Call Streams. In: Proceedings of the 11th USENIX Security Symposium, San Francisco, California (2002)
[43] Christodorescu, M., Jha, S.: Static Analysis of Executables to Detect Malicious Patterns. In: Proceedings of the 12th USENIX Security Symposium, Washington, DC (2003)
[44] McHugh, J., Gates, C.: Locality: A New Paradigm for Thinking about Normal Behavior and Outsider Threat. In: Proceedings of the ACM New Security Paradigms Workshop, Ascona, Switzerland (2003)
[45] Jain, K., Sekar, R.: User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In: Proceedings of the Network and Distributed System Security Symposium, San Diego, California (2000) 19–34
[46] Ghormley, D.P., Petrou, D., Rodrigues, S.H., Anderson, T.E.: SLIC: An Extensibility System for Commodity Operating Systems. In: Proceedinges of the Annual USENIX Technical Conference, New Orleans, Louisianna (1998) 39–52
[47] Fraser, T., Badger, L., Feldman, M.: Hardening COTS Software with Generic Software Wrappers. In: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California (1999) 2–16
[48] Ko, C., Fraser, T., Badger, L., Kilpatrick, D.: Detecting and Countering System Intrusions Using Software Wrappers. In: Proceedings of the 9th USENIX Security Symposium, Denver, Colorado (2000)
[49] Nebbett, G.: Windows NT/2000 Native API Reference. 1st edn. MacMillan Technical Publishing (2000)
[50] Solomon, D.A., Russinovich, M.E.: Inside Microsoft Windows 2000. 3rd edn. Microsoft Press (2000)
[51] Postel, J.B.: Simple Mail Transfer Protocol (1982) http://www.ietf.org/rfc/rfc0821.txt.
[52] Russell, D.L.: State Restoration in Systems of Communicating Processes. IEEE Transactions on Software Engineering **SE6** (1980) 133–144
[53] Liu, Y., Sevcenco, S.: W32.bugbear@mm. Symantec Security Response (2003) http://securityresponse.symantec.com/avcenter/venc/data/w32.bugbear@mm.html.
[54] Sevcenco, S.: Vbs.haptime.a@mm. Symantec Security Response (2004) http://securityresponse.symantec.com/avcenter/venc/data/vbs.haptime.a@mm.html.
[55] Gudmundsson, A., Chien, E.: W32.klez.e@mm. Symantec Security Response (2003) http://securityresponse.symantec.com/avcenter/venc/data/w32.klez.e@mm.html.

[56] Ferrie, P., Lee, T.: W32.mydoom.a@mm. Symantec Security Response (2004) http://securityresponse.symantec.com/avcenter/venc/data/w32.novarg.a@mm.html.