

Host-Based Data Exfiltration Detection via System Call Sequences

Brian Jewell¹, Justin Beaver²

¹Tennessee Technological University, Cookeville, TN, U.S.

²Oak Ridge National Laboratory, Oak Ridge, TN, U.S.

bcjewell21@tntech.edu

beaverjm@ornl.gov

Abstract: The host-based detection of malicious data exfiltration activities is currently a sparse area of research and mostly limited to methods that analyze network traffic or signature based detection methods that target specific processes. In this paper we explore an alternative method to host-based detection that exploits sequences of system calls and new collection methods that allow us to catch these activities in real time. We show that system-call sequences can be found to reach a steady state across processes and users, and explore the viability of new methods as heuristics for profiling user behaviors.

Keywords: Data exfiltration, Data security, Intrusion detection, System call

1. INTRODUCTION

A successful attack on an organization involving the theft of sensitive data can be devastating. *Data exfiltration* is the term used to describe this type of theft and can be defined as the unauthorized transfer of information from a computer system. Data exfiltration attacks represent a tremendous threat to both government entities and commercial enterprises. Government organizations maintain repositories for sensitive and classified information, and breaches into protected systems or leaks into the public domain can have implications that threaten national security. Commercial enterprises manage complex levels of proprietary tools and data that, if compromised, could endanger the financial security of their institutions and/or their customers. Recent studies find that information leaks are the most prevalent security threat for organizations [0] and that in recent years attackers have exfiltrated more than 20 terabytes of data, much of which was sensitive, from the U.S. Department of Defense and Defense Industrial Base organizations, as well as civilian government organizations [0]. Valuable intellectual property and other sensitive information are often vulnerable to theft by insiders and outsiders alike and typically go unnoticed until the attackers have caused irreparable harm.

Despite the threat, the approach to defending against data exfiltration attacks is surprisingly unsophisticated. Off-the-shelf intrusion detection systems (IDS) monitor for known malicious network signatures at the system boundary. These systems are relied upon to flag potential network breaches, which are then typically investigated manually (often these are guided analyses that leverage custom-built scripts) in order to trace potential unauthorized activities. Unfortunately, the model of perimeter defense leaves attackers free to navigate, investigate, and exfiltrate information if the perimeter is breached undetected. Even recent anomaly based detection systems (Forrest 2008)(Kang 2005) that claim zero-day vulnerability detection concentrate on identifying changes in process behavior instead of attempting to profile a specific user.

Host intrusion detection systems (HIDS) are software programs that run on each computer host in a network and attempt to detect malicious events in the operation of the host. Commercial virus protection packages (McAfee 2003) are examples of HIDS and monitor system services, registry changes, and check individual files for signatures of known malicious programs. We approach the detection of data exfiltration attacks as a HIDS. Once the boundary defense is breached, it is from the individual hosts that a malicious user will explore file systems, package data, and export it to an outside network. We postulate that, given insight into the activities of individual users and processes on a given host, acts of unauthorized data exfiltration can be discriminated from normal user/process behaviors.

Our hypothesis hinges on the availability of low-level data that reflects the operation of processes on a computer host. We propose to achieve this insight into the computer's operation through the monitoring of *system calls*, which are low-level process interactions with the host computer's operating system. System calls provide a window into what all processes and users on a host machine are executing, regardless of how they are interacting with the machine. In addition, they provide more fidelity in identifying individual actions than a process monitor.

In this paper, we propose the use of a method by which unique sequences of system calls, managed at the process/user level, are the basis for discriminating normal and anomalous behaviors by users for use as an exfiltration detection agent. We then evaluate this model to fit our 3 criteria for a viable detection agent.

- *Tractable*- The chosen method must be able to run in real time while having negligible effect on a system as experienced by the end user.
- *Environmentally Neutral*- Our method must also be portable and adapt to any environment.
- *Responsive*- Lastly, our ideal method should reliably report on data exfiltration behaviors.

The following sections are organized as follows: Section 2 provides a review of similar work. Section 3 formalizes the methodology we used to categorize normal behavior and collect a profile from the system call data traces. Section 4 evaluates our method according to the 3 criteria we set, and Section 5 gives a detailed account of our results, conclusions, and ideas for future work.

2. RELATED WORK

The detection of data exfiltrations has been a recent focus of cyber security research. Exfiltration detection is a difficult problem due to the wide range of methods available, and the subtlety with which it can be performed [1]. Current IDS systems are mostly concerned with intrusion attempts, although there are extrusion detection systems that are commercially available (e.g., [2]). Like network-based IDSs, these are primarily signature-based solutions that perform network traffic analysis through custom hardware. Many more advanced data analysis approaches have been proposed, including clustering of network traffic for anomaly detection [3], the application of statistical and signal processing methods to outbound traffic for signature identification [4], and the application of data mining techniques [5] to network data. These approaches yielded varying degrees of success, but inevitably were plagued with base-rate fallacy [6] issues or a narrow problem focus.

However, when we look at previous work on host-based IDSs, there is some inspiration for host-based data exfiltration detection. In 1996, Forrest, et al, proposed a host-based intrusion detection method based on the monitoring of system calls (Forrest 1996). This early work was inspired by the human immune system's ability to recognize what cells are part of the host organism (it's self) or foreign (non-self). They used this principle in developing their own methodology for constructing a "sense of self" for Unix based systems using available system trace data. This methodology used *lookahead pairs*- sets containing pairs of system calls formed by the originating system call and the one that follows it with spacing 1, 2, 3, .. k. These pairs were used to form a database of normal process behavior (or self), and used to monitor for previously unfound patterns, that were then tagged as anomalous (or non-self). While their results were only preliminary, they did show that a stable signature of normal process behavior could be constructed using very simple methods.

Many other approaches have been taken since to model the behavior of processes using system calls, including the use of Hidden Markov Models (HMM) (Gao 2006), neural networks (Endler 1998), k-nearest neighbors (Liao 2002), and Bayes models (Kosoresow 1997). These models were all developed in hopes of producing more accurate models while reducing false positives. However, this comes at a high computational cost. The most notable advantage of Forrest's model was the ability to track processes for anomalous behavior at the application layer of each individual host in real time at a very low computational cost.

Forrest, et al, later improved upon their earlier work in (Forrest 2008) by introducing another simple model that is suitable for real-time detection dubbed sequence time-delay embedding (stide), and again involved the enumeration of system call sequences. However, this time their method used contiguous sequences of fixed length to form a database of normal behavior. They also introduce a new modification to their method called sequence time-delay embedding with frequency threshold (t-stide). This method explored the hypothesis that sequences with very low occurrence rates in training data are suspicious. They tested these methods against 2 popular machine-learning methods. One based on RIPPER - a rule learning system developed by William Cohen (Cohen 1995) that was later adapted by Lee et al (Lee 1998)(Lee 1999) to learn rules to predict system calls and find anomalies,

and the other based on HMMs as used in (Gao 2006). While they weren't able to show that it performed the other methods, they did conclude that it performed comparably to more complicated methods.

Our work is most closely paralleled by that of Forrest and leverages host-based system call information to detect anomalous user behaviors. Unlike previous work, we seek to implement and adapt this approach as an analysis process that is user centric and that meets our three criteria for a data exfiltration agent.

3. METHODOLOGY

Our model for data exfiltration detection focuses on the analysis of system calls used in a host's operation and hinges on observations similar to that found in previous works by Forrest et al (Forrest 2008). This section justifies the use of sequences of system calls as a mechanism for defining normal behaviors in Section 3.1, discusses variants in optimizing system call sequences in Section 3.2, and compares these variants for use as data exfiltration detectors in Section 3.3.

3.1 Defining Normal in Sequences of System Calls

A *system call trace* or *system call sequence* is the ordered list of system calls as invoked by a process that spans the length of execution by a given user. An example system call trace for a given user might be:

“..., open, read, fstat, fstat, write, close, mmap,...”,

where “open”, “read”, “fstat”, etc. are all examples of system call executable names. All invoked user operations, whether a command line imperative or in the operation of a running program, use various combinations of system calls to complete their tasking. Even simple commands, such as a directory listing, use a sequence of multiple system calls to execute.

While there are a number of current methods to enumerate system call sequences, there is a common theme: to form a data store of traces that are used to characterize normal behaviors (also referred to as a normal profile) in a given environment. Once the data store is established, it can be used as the basis for identifying future sequences as normal (within the set) or anomalous (not included in the set). In addition, it is desirable for any automated comparison of this profile with experienced events to be computationally tractable.

Previous research (Forrest 2008) on host-based IDSs that has attempted to use system call sequences to detect anomalous behaviors has concentrated on detecting anomalies in program execution. That is, the focus of the analysis is on individual processes and their execution but did not take into account the uniqueness of each individual user. By contrast, when attempting to detect data exfiltrations, we are more interested in the behavior specific to a user. However, in order to create a normal profile that is specific to a user, it must be established that system call sequences are suitable for discriminating normal and anomalous behaviors in such a context.

Given that, experientially, user behavior seems to vary drastically depending on the task being performed at any given moment, it is necessary to support the claim that unique system call sequences for a user can be generalized. We performed an experiment in which the unique system call sequences for individual users were tracked. The results of this experiment are illustrated in Figure 1. We define a stable profile as one that plateaus at a given size (N sequences). It's the asymptotic nature of the line that makes the anomalous detection possible. That is, in a given trace, the number of sequences generated can always be observed to "step" or to plateau under normal usage and to increase suddenly when a user performs a new or unusual action. Figure 1 demonstrates that, despite varying operations by users, a normal profile can be established and characterized by a tractable (< 200) number of unique system call sequences.

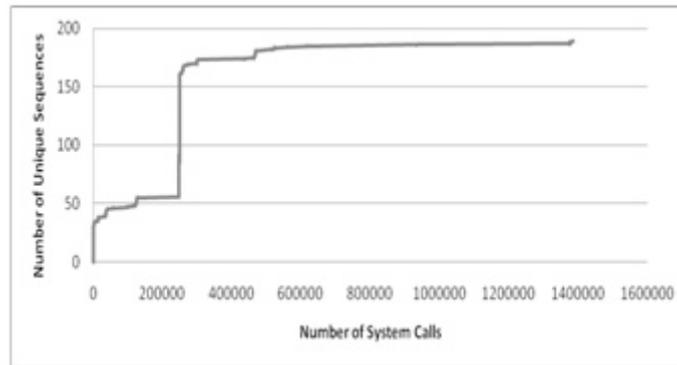


Figure 1: Number of unique system call sequences for a given user/process versus the total number of system calls.

The results in Figure 1 provide some confidence in our basis of focusing on user behaviors over individual process behavior. By organizing the host's system calls through user identifier (UID) / process name pairs, we can model specific behaviors unique to each user. This allows for the tracking of a specific user's behavior for a specific user, and it helps to expose some interesting clues to the diversity and nature of system call sequences.

3.2 Models for System Call Sequences

For our testing we implemented three different simple methods for enumerating system calls. The first of these methods is implemented very similarly to stide (Warrender 1999). The method uses a sliding window of size N across all system calls included in a trace to form the sequences. However, we have adapted the method to incorporate UID/process name pairs to create a profile of live data.

We also wanted to take care in choosing an appropriate value of N to be used with our implementation. The best value for N that is used for stide and similar implementations is discussed in a number of previous works. Hofmeyr et al (Kosoresow 1997) suggest "the best sequence length to use would be 6 or slightly higher than 6." And Kymie 0 in a paper dedicated to the singular question of "Why 6?" provided evidence supporting the conjecture. However, while evaluating our own variable sequence length methods we identify another possible and more fundamental reason to pick a sequence size of 6. In Figure 2 we see the number of unique sequences present in a complete "normal" profile generated by our variable length sequence collector over one week. It is interesting to note the dramatic decrease in the number of sequences that occur with a length greater than 6. As the value of N increases we increase the accuracy of the profile generated proportionately to the percentage of system call sequences that fall under that size. However, we also increase our learning time and profile complexity by the same proportions. Therefore, for our experiments we also use 6 for the length of our sequences

The next model is designed out of the desire to avoid what appears to be a rather severe shortcoming of the windowing method. Many sequences of length 1 or 2 can be observed as repeating continuously, making a perfect fit with the windowed method requires substantial unnecessary overhead. This is also demonstrated in Figure 2. This leads us to theorize that a method utilizing a variable window length would perform better than the previous methods.

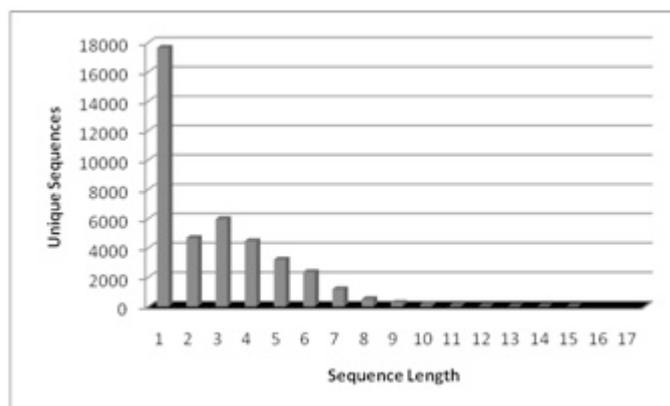


Figure 2: Number of unique sequences observed with the size N (x-axis)

While developing an approach to create variable sequences of system calls it was important to preserve the low run time complexity of the sliding window method while attempting to better model normal behavior. Thus, a simple solution was chosen. In order to construct our variable sequences we chose a subset such that sequence length is maximized while no one system call is repeated. This is implemented by constructing a sequence as calls are being traced and beginning a new sequence when a call is found to be a repeat in the current sequence.

Up to this point we ignore the additional information about each system call in building our normal profile. Thus, we implement a third method that additionally uses `errno`, `arg[0]`, and function arguments in matching sequences. The method uses the same methodology as the variable length sequences. Unique system call sequences are selected in such a way that length is maximized while no one system call is repeated in a given sequence. However, here we define a system call as `{probefunc, errno, args}`.

3.3 Comparison and Discussion

To validate our methods we tested them against each other. In Figure 3 we show the increase in number of sequences generated over one week of collection for a given UID/execname pair. From Figure 3 we observe that the variable sequence length method both finishes training of a normal sequence faster and uses fewer sequences than both of the other methods as hoped. This is likely in part due to the observation that the majority of sequences have a length less than 6 and the smaller the sequence the more that they repeat. (Refer back to Figure 2)

Other observations we can make from Figure 3 is that the variable with arguments method reaches a stable state faster than the windowed method and without an unacceptable large increase in the number of unique sequences. Again, this is most likely due to the better fitting of high frequency sequences.

The surprise comes in how poorly the windowed method performs in terms of generating a stable profile. Overall the windowed method performs well, but when the testing is stretched over the period of a week the method fails to show the same level of stabilization as the other 2 methods. Thus, for all purposes the variable method seems superior, with the addition of arguments requiring a much large database, which correlates to a lot more false positives. Since detection speed and precision are what we're interested in, we'll be using the variable method for the remainder of our testing.

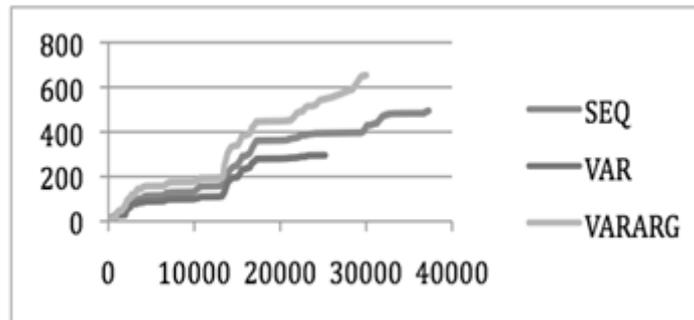


Figure 3: Three sequence collection methods (SEQ - windowed, VAR - variable, VARARG - variable with arguments) compared by number of unique sequences generated.

4. EVALUATION

We now evaluate the method prescribed in the previous section against our three criteria for our ideal exfiltration detection agent.

4.1 TRACTABLE

Perhaps the largest singular challenge encountered during the implementation of this project is the task of collecting and managing the torrent of system calls that occur during normal to heavy use of a modern computer. Each user or process action can result in hundreds of system calls and in our own experiments logging system call activity alone generates a gigabyte of data per hour. In previous works (Forrest 2008)(Kang 2005) this problem is sidestepped mainly by concentrating on individual processes/users/calls and/or using previously collected data.

Unlike prior efforts, we are interested in tracing all system calls across multiple users to track their behavior in real time, and we also desire to deploy a swift analysis of that data without noticeably degrading system performance or destabilizing the system. While researching options for this we found an existing commercial solution that meets all of our needs. Dtrace (Dtrace 2009) is a software tool that is designed specifically for low impact system call tracing for system administration and debugging. More importantly, it can be configured to collect the required system call data with negligible effect on system performance, such as timestamp, user and process identifiers, executable names, error numbers, and executable arguments.

Another challenge is the management of the collected data. Retaining and cataloging all the system calls for analysis at a later time is impractical given the observed data rate of over 1 gigabyte per hour. As we previously discussed, by collecting just the unique sequences that form our profile of normal behavior in real time we elegantly address this problem. Also as we discussed in earlier, Figure 1 shows the increase in the number of unique system calls for the 1/java pair in an hour-long system call set using our variable sequence model. This pair was chosen because of the volume of system calls that are produced while the program was actually conducting only a few functions repeatedly. Over the course of over 1.4 million system calls generated by 1/java over the course of the collection, only 196 unique sequences were recorded. It's this quick stabilization and small normal profile that along with Dtrace combine to make our implementation lightweight with very low observed impact on system performance.

4.2 Environmentally Neutral

In order to validate that we can distinguish a normal profile of one user/process apart from another regardless of environmental conditions such as the operating system or other operational conditions, we must first explore the "diversity hypothesis" similar to that put forth by Forrest et al in (Forrest 2008). Their hypothesis states that the code paths executed by a process are highly reliant upon the usage patterns of the users, configuration, and environment hence causing what is considered to be normal to differ widely from one installation to the next. While the methods used to create the sequences that Forrest et al are similar they focus solely on program execution, the same diversity should theoretically still exist between the profiles generated by our methods when per user patterns

are added as a controlling factor. In addition it may also be possible to determine the degree of impact changes such as different operating systems and varying users have upon a normal profile. We can observe this in our own testing by comparing the various collected profiles from different users and operating systems.

Table 1: Comparison of normal profiles generated by different users by platform.

	User 1B	Linux (User1)	Solaris (User1)
User 1A	0.91129591	0.16700353	0.19755409
User 1B	1	0.14119998	0.13764726
User 2	0.25793254	0.13287113	0.12885861
User 3	0.30644131	0.17470944	0.20602069

For this testing we had 3 different users (User 1A, User 2, and User 3 in Table 1) run our variable sequence collection algorithm for approximately 1 hour. All users were using Mac OSX on separate machines. In addition, we had User 1 repeat the same collection process on a separate date using Linux and Solaris operating system on different machines trying to keep behavior as similar as possible.

The most significant result is that profiles from User 1A and User 1B have a >90% match while profiles generated from the other 2 users did not exceed 31% when compared to User 1. This seems to confirm that there is significant variation between profiles of one user from another. Perhaps the disappointment here is that correlation between User 1A and User 1B profiles wasn't closer to 100%. However, it should be noted that most of the difference between these 2 sets was the use of a new process in the User 1B profile that wasn't present in the User 1A profile. This type of anomaly will have to be taken care of in any future implementation.

Differences in profiles among various users are expectedly severe, with the biggest difference coming from the use of different operating systems. This is perhaps unsurprising since many of the system calls that are used by Mac OSX aren't used by Linux and vice versa. The same goes for Solaris vs. the others as well. However, this does validate that any model will have to be highly adaptable to the environment and not rely on a predetermined set of signature detection algorithms. This property does however help us greatly as mimicry attacks will be extremely difficult to carry out without specific knowledge of the environment and user's behaviors.

4.3 Responsive

The last of the criteria for evaluating our chosen implementation is the ability to detect a very large variety of data exfiltrations. For this stage of testing we issued a challenge that was conducted over the course of 2 days at Oak Ridge National Laboratory (ORNL) during the summer of 2010. Participants were solicited from the lab to exfiltrate a number of files setup on one of our testing machines. All participants were asked to exfiltrate 3 files:

- A plain text file plainly labeled in a directory which everyone that participated could reach easily with their own account.
- A mock transactional database containing simulated sensitive personal financial information that was hidden within a shared location on the same machine.
- A document that was clearly labeled and had a known location but in a user directory with restricted access.

While this data set will have a number of other uses in the future, it currently gives a good view of whether it will be feasible to detect attacks in progress and give an idea of what those attacks might look like. We had originally hoped that our attacks would display some specific similarities to each other, perhaps manifesting as an increase in certain system call types or some other type of pattern.

However, we found that all of our attacks differed significantly with a wide variety of tactics deployed. Even those attacks that appeared to use the same tactics of exfiltration displayed very dissimilar system call sequence profiles. Overall there were 18 individual UIDs and over 9 gigabytes of alerts observed during the 2-day period. The size of the dataset collected in contrast to the average observed rate of approximately 2 megabytes of alerts generated under normal operation over the same time period seems to verify that our method is sufficiently sensitive to data exfiltration activities. Among the 20 observed UIDs, 8 are identifiable as successfully retrieving at least one of the files, and at least 2 retrieving all three. Observed behaviors included probing with find, escalation of privilege attempts, mass data exfiltration using sftp, and transferring the files to an USB flash drive.

The detection of many of these attacks is simplified given that they were new users on the system using a distinct UID. However, a few of the attacks were observed among root and the primary users UID lending credibility to the system's ability to detect exfiltration behaviors even when the activity is hidden amongst normal system operation and users. As for the other attacks, each incident should have generated an alarm by design, and for our immediate purposes serve to validate that our implementation is working as intended.

5. CONCLUSIONS

The accurate detection of malicious data exfiltration is a complex task that can take human experts months. However, in order to react to an attack a practical system not only needs to detect attacks autonomously, but do so in real time before files can be leaked.

The goal of this paper was to identify and test ways to approach this problem. We initially identified the main issues that separated what we needed in our implementation as opposed to previous work on HIDSs. We sought a method that would be tractable to run in real time, environmentally neutral as to perform well with any operating system or conditions, and most importantly responsive to behaviors specific to data exfiltrations. With these criteria in mind we adapted a means of host-based detection using sequences of system calls to implement a data exfiltration detection agent.

In all of our testing, we have found that data exfiltration behaviors can be successfully detected by simple means in real time with negligible effect as observed by the user. Our adaptation of system call sequence monitoring to this specific problem is promising and passed our three main evaluation criteria. The implementation was successfully run in real time and deployed across a diverse set of systems and users. We were also able to present evidence that our method detects a wide range of exfiltration related behaviors.

However, questions still exist as to whether we can detect these type of behaviors fast and accurately enough to prevent data exfiltration. More work is obviously needed in correlating suspicious behavior into individual attacks, and further testing of our methods against known attacks is warranted to determine long-term performance.

ACKNOWLEDGEMENTS

This work was supported in part by the Lockheed Martin Corporation. The views and conclusions contained in this document are those of the authors. This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

REFERENCES

- Axelsson, S. (2000) "The Base-Rate Fallacy and the Difficulty of Intrusion Detection." ACM Transactions on Information and System Security, Vol. 3 No. 3, pp. 186-205.
- Cohen, W.W. (1995) Fast effective rule induction. In *Machine Learning: the 12th International Conference*. Morgan Kaufmann.
- Coleman, K.G. (2008) "Data Exfiltration." [online], <http://it.tmcnet.com/topics/it/articles/37876-data-exfiltration.htm>.
- Dtrace (2009), [online], <http://www.oracle.com/technetwork/systems/dtrace/dtrace/index.html>.

Endler, D. (1998) Intrusion detection: applying machine learning to solaris audit data. In Proc. of the IEEE Annual Computer Security Applications Conference, pages 268–279. Society Press.

Fidelis Security Systems, (2009) "Fidelis Extrusion Prevention System". [online], <http://www.fidelissecurity.com/>.

Forrest, S. et al (1996) A sense of self for UNIX processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, Los Alamitos, CA, IEEE Computer Society Press.

Forrest, S. et al (2008) "The Evolution of System-call Monitoring", *2008 Annual Computer Security Applications Conference*.

Gao, D. et al (2006) Behavioral distance measurement using hidden markov models. In D. Zamboni and C. Kruegel, editors, *Research Advances in Intrusion Detection*, LNCS 4219, pages 19–40, Berlin Heidelberg, Springer-Verlag.

Ghosh, A. and Schwartzbard, A. (1999) A study in using neural networks for anomaly and misuse detection. In *Proceedings of the 8th USENIX Security Symposium*.

Giani, A. et al (2004) "Data Exfiltration and Covert Channels." In *Proceedings of the SPIE 2004 Defense and Security Symposium*.

Hooper, E. (2009) "Intelligent Strategies for Secure Complex Systems Integration and Design, Effective Risk Management and Privacy." In *Proceedings of the 3rd Annual IEEE International Systems Conference*.

Kang, D. et al (2005) "Learning Classifiers for Misuse and Anomaly Detection Using a Bag of System Calls Representation", *Proceedings of the 2005 Workshop on Information Assurance and Security*, 2005.

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

Kymie, M.C.T and Maxion, R. (2002) "Why 6? Defining the Operational Limits of stide, an Anomaly-Based Intrusion Detector."

Lee, W. et al (1997) Learning patterns from UNIX process execution traces for intrusion detection. In *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 50–56. AAAI Press.

Lee, W. and Stolfo, S.J. (1998) Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*.

(Liao 2002) Y. Liao and V. R. Vemuri. Use of k-nearest neighbor classifier for intrusion detection. *Computers & Security*, 21(5):439–448, 2002.

Liu, Y. et al (2009) "SIDD: A Framework for Detecting Sensitive Data Exfiltration by an Insider Attack." In *Proceedings of the 42nd Hawaii International Conference on System Sciences*, 2009.

McAfee (2003), [online], <http://www.mcafee.com/us/>.

Richardson, R. (2007) CSI Computer Crime and Security Survey, [online], <http://icmpnet.com/v2.gocsi.com/pdf/CSISurvey2007.pdf>.

Sans Institute. (2010) "20 Critical Security Controls, Critical Control 15: Data Loss Prevention." [online], <http://www.sans.org/critical-security-controls/control.php?id=15>

Warrender, C. et al (1999) "Detecting Intrusions Using System Calls: Alternative Data Models." In *1999 IEEE Symposium on Security and Privacy*.