A SCALED, IMMUNOLOGICAL APPROACH TO ANOMALY COUNTERMEASURES

Combining pH with cfengine

Kyrre M. Begnum
Faculty of Engineering, Oslo University College, Norway
Kyrre.Begnum@iu.hio.no

Mark Burgess
Faculty of Engineering, Oslo University College, Norway
Mark.Burgess@iu.hio.no

Abstract: We discuss the combination of two anomaly detection models, the Linux kernel module pH and cfengine, in order to create a multi-scaled approach to computer anomaly detection with automated response. By examining the time-average data from pH, we find the two systems to be conceptually complementary and to have compatible data models. Based on these findings, we build a simple prototype system and comment on how the same model could be extended to include other anomaly detection mechanisms.

1. Introduction

Computer Immunology is an approach to integrity management, based on the notion that computer systems are healthy when their behaviour is free of anomalous occurrences[16, 3]. The onus falls on researchers to define what 'anomaly free' means, or conversely what is normal for a system. This can be done in several ways.

Commonly one supposes that systems are normal when they exhibit medium term stability, i.e. stability on a time scale at which users experience the system[4]. Health or stability is thus related to ones idea of *policy*. Long term changes, such as policy revisions, can occur and short term changes are occurring all the time. Normality is a statistical concept, which accrues over time, and computer immunology is a form of computer learning[6, 12]. Unlike many other methods of artificial intelligence, computer immunology is about purely mechanistic regulation of behaviour, rather devoid of 'intelligence' in the normal sense of the word. It concerns prescriptions for recognition of change with computer systems. In summary, this medium term stability is achieved with the following strategy:

- The system should be tolerant of anything determined by policy.
- Policy is partly specified and partly learned from experience.
- The system should react to abnormal or non-policy-conformant events in order to restore normality.

Two approaches have emerged for addressing these issues at different scales.

- At the University of New Mexico, the Computer Immunology group has examined strategies for detecting signatures of abnormal computer behaviour at kernel level. Their pH system[10, 14] learns new signatures over time, but is resistant to doing so. The primary motivation here has been in deflecting network intrusions, though the method is equally effective in detecting abnormal local usage, such as attempts to exploit buffer overflows. The response provoked by anomalies has been in the form of scheduling delays in processes with unknown call sequences, in order to urge attackers to lose interest.
- At Oslo University College we have focused on the configuration management aspect of policy, using a system of agents (cfengine) that detect and use their environmental conditions and current configuration to detect anomalous changes[1]. Again, the policy is partly specified and partly learned from patterns of usage, and the response to different events is specified itself as a matter of policy, and the agents ensure that the system tends to maintain the same state over time.

This paper describes the process of combining cfengine, a high level configuration engine with pH, a kernel patch which enables anomaly detection and reaction on a per process basis. The project has two independent goals: to provide a better anomaly detection capability for cfengine, and a better response engine for pH; to create a versatile framework for the collection of system related data for further research into anomaly detection. There is thus a security motivation and a research motivation. The 'science' of anomaly detection is still in its infancy, thus the latter should not be neglected for the sake of building a quick fix.

The plan for this paper is as follows. We begin by discussing the requirements for compatibility between pH and cfengine, as well as what we hope to achieve by combining them. In sections 3 and 4 we provide some details about these two systems in order to contextualize their marriage. Finally, we provide a cooperative model for these systems and discuss further extensions for future work.

2. Compatibility

On the surface, it would seem that pH and cfengine are two very different systems, with somewhat different goals. How then are they to be meaningfully combined?

The common thread between the systems is their long term goal: that of system regulation, or *homeostasis* (state regulation)[2, 3, 15, 6]. When a change occurs in a system, there are two general ways that it can respond. With negative feedback, the system responds in such a way as to reverse the direction of change; this tends to keep things constant and allows us to maintain a regular state; positive feedback would tend to amplify the change. This has a de-stabilizing effect, so it does not result in homeostasis. While it can be useful for rapid responses, it is a dangerous strategy, since the change will tend to dominate and eventually consume a system. Regulation, or homeostasis, is thus a self-regulating mechanism that allows a system to avoid paying detailed attention to its most basic functions thereby helping keep it in a steady state

The University of New Mexico's pH kernel modification stands for process homeostasis. Its goal is to seek a 'steady state' list of tasks that are undertaken by a computer system (i.e. a normalized list). It detects previously unknown tasks and offers resistance to their execution. If the new tasks persist, they are eventually tolerated by the system.

Cfengine, on the other hand, seeks to maintain a 'steady state' *configuration* of a system, where configuration means the state of the file system, process table and service ports. It detects and opposes changes by two strategies: i) by referring to a descriptive policy about what is considered acceptable, and ii) by monitoring key system resource usage over days and weeks, and responding to statistical irregularities. Thus, both pH and cfengine have a policy of maintaining a 'normal' or 'regular' state, and both are able to learn about long term changes by adapting their reference state. Their key difference is the scale at which they operate: pH works at the microscopic, short-term level of system calls, while cfengine works at medium term user time-scales.

How can these be combined? As we have already stated, an adaptive, learning system is necessarily a statistical system; we should therefore ask: i) Do they have compatible data models? ii) What are the tolerances of the systems? i.e. with what accuracy can they make claims about system normality; hence, when is it appropriate to activate countermeasures?

iii) Resource utilization is known to be a strongly social phenomenon, with a marked variation over the working week. Cfengine uses the working week as a model for measuring its medium-term state. Is the same time reference appropriate for pH, which deals with short term events?

Combining these seemly disparate mechanisms is thus a scaled approach to system regulation. PH detects events on short time scale, responds simply and propagates the data forward as medium term statistics which it uses privately for future reference. Cfengine measures medium term events and activates medium to long term response strategies. Our aim here is to see whether medium term data from pH can be read and utilized by cfengine in order to bring the knowledge of short term behaviour to bear on longer term strategy.

3. Short introduction to pH

Ph is a patch for the Linux 2.2 kernel. It addresses the anomaly countermeasure problem at the level of system call sequences. A lot of security software today is designed to detect attacks (e.g. Intrusion Detection Systems (IDS)) or to find vulnerabilities in the system; only a few tries to stop attacks as they occur (e.g. some can delete viruses and even repair damaged files). Although a response capability exists in some programs, most software only issues a warning, and waits for a system administrator to react.

The key is the ability to tell the difference between "benign" and "hostile". For pH, like most other IDS, *normal* is benign, and *abnormal* is hostile. By analysing the pattern of system-calls made by each process using pattern-matching algorithms, pH gains knowledge about what it perceives as normal behaviour. It also maintains a profile of each binary as to see if each process produces the expected patterns of system-calls. As long as a process keeps it's number of strange patterns under a certain level, it is considered normal. If the number rises above a threshold, pH starts to sabotage the process by delaying all the system-calls made by it.

Ph keeps one profile for each binary, but it reacts to individual processes. Every process has a sequence of system calls, known as a *trace*. The profile of each binary is updated and adjusted to the behaviour of the processes. This means that instances of a specific behaviour will in time be considered normal. Not all anomalies are real threats to the system, but earlier research by UNM suggested that "To date, all of the intrusions we have studied produce anomalous sequences in temporally local clusters[10]; pH is therefore designed to react regarding the density of anomalous system call patterns.

Strategy

The algorithm used by pH is called "time-delay embedding"; it looks at the trace of each process' system calls. For each system call, pH notes the calls preceding this one within a window. This gives a number of system call pairs for each position in the window. For instance, suppose we have the following trace of an imagined process:

```
getpriority, open, write, write,
close, open, pipe, close, exit
```

We read the trace from the left. While reading, we note which calls come behind the current one. Just like sliding a window over the trace. The default window-size for pH is 6 calls. pH does not consider each sequence it encounters as part of its process profile right away. There is a distinction between the current profile (called *test*) for a given binary and the temporary profile of the running process (called *train*). The *train*-array is continuously updated with new pairs. Should a pair occur, that is *not* part of the *test*-array, then it is considered an anomaly. The test-array can only be updated by replacing it with the current train-array. This replacement occurs under one of three conditions (from the documentation): i) The user explicitly signals via special system call (`sys_pH`) that a profile's training data is valid. ii) The profile anomaly count exceeds the parameter `anomaly_limit`. iii) A specialized training formula is satisfied.

Should an anomaly occur, then a number of the following system-calls will be delayed. After a certain number of anomalies, the train-profile will switch to the test-profile. This is called *acquired tolerance*, meaning the profile adapts to the behaviour of the process. But should the anomalies occur too close to each other, then pH will react in the opposite manner and reset the train-profile.

Implementation

It is possible to control pH at runtime with a system call interface: `sys_pH()`, and `pH-admin` which is basically a front-end to the system call. This tool can, amongst other things:

- Turn the monitoring on/off
- Write profiles to disk
- Adjust pH-variables (i.e `delay_factor`)
- Force the train profile to be copied to the test profile.

pH saves information about each running process from the `/proc` directory. Each folder belonging to a process has a file called `pH`, which contains information about delay, system call count, if the profile is considered normal and if the process is currently frozen.

All the messages created by pH are logged in the log file `/var/log/syslog`. The profiles for all the binaries are located in the folder `/var/lib/pH/profiles` where they are sorted in a hierarchy which mirrors the actual file-system. Each binary is therefore identifiable by it's path. For example, the program `less`, which has the path `/usr/bin/less`, will have it's profile at `/var/lib/pH/profiles/usr/bin/less`.

4. Short introduction to cfengine

Cfengine is a well-known policy based configuration management system written at Oslo University College[1], which is comprised of a number of components (see fig 1). An agent component is responsible for enforcing specified policy by comparing a description of the permissable states of a host's configuration with the host's actual state. There are also file-server and scheduler components for deploying cooperative management schemes. The `cfenvd` environment daemon is a component that measures system resource usage, independently of the other parts and records it in a database[5], which becomes the definition of 'normal'. This tool is intended both for regulative feedback and for gathering research data. It classifies the current state of resource usage

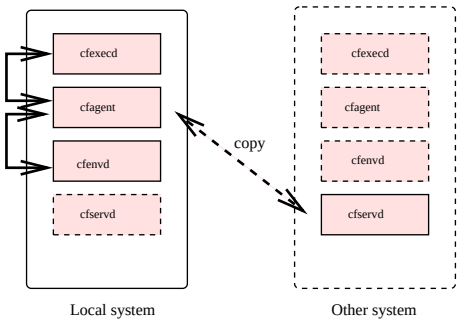

Figure 1 A schematic representation of cfengine components. The environment daemon communicates with the agent on each host, by providing it with classified state information.

in relation to what has been learned previously, using units of the statistical uncertainty (standard deviation) for each time of week. It then publishes its results for other programs to use, notably `cfagent`. `Cfagent` receives the data as a 'classified event' which can be used to predicate countermeasures or follow-up responses for the state concerned. Some examples of classes which can become active in the `cfagent`:

```
RootProcs_low_dev2
netbiosssn_in_low_dev2
smtp_out_high_anomalous
www_in_high_dev3
```

The first of these classes tells us that the number of root processes is two standard deviations below the average for past behaviour. This might be fortuitous, or might signify a problem, such as a crashed server; we do not know

the reason, only that an anomaly has occurred. The WWW item tells us that the number of incoming connections is three standard deviations above average. The SMTP item tells us that the number of outgoing SMTP connections is more than three standard deviations (this is the defined meaning of anomalous) above average, perhaps signifying a mail flood. The setting of these classes is transparent to the user, but the additional information is only visible to the privileged owner of the cfengine work-directory, where the data are cached.

Countermeasures or follow-up actions can be attached to events in order to automate a policy decision to the occurrence. For example, one might decide to shut down an offending service temporarily, and then follow up with a file audit:

```
processes:

 smtp_out_high_anomalous::

   ‘‘sendmail’’ signal=kill

files:

 smtp_out_high_anomalous::

   /usr recurse=inf checksum=md5
```

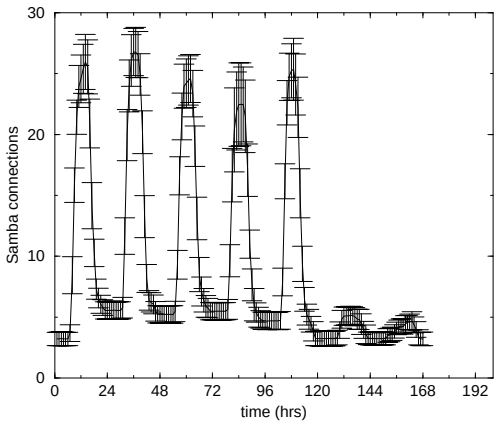

Figure 2 Cfengine measures patterns of resource usage over the working week. This example shows how measurements of Samba file sharing lead to an average picture of behaviour at different times of the week. The solid line is the average value over many weeks and error bars indicate the standard deviation.

5. A cooperative model

We wish to combine these two systems in order to create a better and comprehensible high level system that can react to the systems state. This is a topic which was not clear from the available research; therefore wish to find a framework for collecting, storing and analyzing data on their properties. A combined system has to be both reliable and secure if it is to be used on systems that do actual work. Creating a isolated system for testing makes sense for keeping the system clear from uncontrollable noise (users, network traffic and so on). But if noise is normal, and normal is what you're looking for, then the only way to test it, will be real-life.

Various models might be used for establishing a connection between cfengine and pH. The first alternative is a plug-in architecture, where pH is considered to be a cfengine plug-in module. This would facilitate a close working relationship, but it requires permanent structural modifications to both.

A second alternative, would be a model where a higher level system invokes and controls smaller components. The process monitoring would be done by a detached participant. The higher level system would act on the information delivered by the component. This information could be gathered via a spacial interface or by parsing log files.

The model we have chosen is a feedback model that preserves the domain of each software system, but allows a passive communications channel between them (see fig. 4), using files and databases. Thus pH will be able to adjust it's monitoring level depending on instruction from cfengine, and cfengine can adjust its behaviour based on results from pH. pH has it's own engine for data-analysis and cfengine analyzes the data further.

pH already has an interface that cfengine can use to control it in the form of shell commands. We could also go directly to the system call `sys_pH` instead of going via the `pH-admin` command. pH stores its information in several places: `/proc`, `/var/log/syslog` and `/var/lib/pH/profiles`. The profiles are in a self-defined binary format and will be printed to the terminal by the command `pH-print-profile`. The same holds for the sequence files, with the corresponding command `pH-print-sequences`.

In order to collect the data from pH, we use cfengine's `cfenvd` daemon and database, which in turn provides information to the agent when it activates.

In fig. 3, a number of identical trials was performed in order to simulate a long term variation of the form measured by `cfenvd` (see fig. 2). An apache web server was used as the pH monitored process. It was loaded by a number of clients in an identical pattern of variation. Repeating the same changing load five times, a pH process counted the total number of system calls. The average of the five identical trials with standard deviation shown as error-bars is plotted in the figure. Each trial measured 120 values, recorded each 30 seconds over the space of an hour. This figure is sufficient to make two points:

- The statistical model of average behaviour with certain tolerance is compatible with that currently used by `cfenvd`.
- The error bars are not zero, thus there is a natural uncertainty in the results, even with close to identical trials.

The latter point is interesting, since these additional system calls cannot be explained by other processes. Ph measures only system calls related to a specific binary. No other binaries could be responsible for this error.

The fact that there is a statistical uncertainty is very important. It means that the purely digital approach to anomaly detection is not sufficient to yield exact repeatability. Thus if one is looking for repeat-ably identifiable signatures, one must allow a margin for error. This is clearly significant for intrusion detection systems, which normally recognize only exactly learned patterns. The source of the uncertainty could lie both on the side of the server, or on the side of the clients loading the server. It could be a result of scheduling differences, since measurements are cumulative values over a 30 second period. Differences due to network traffic load can be ruled out, since the trials were performed in isolation.

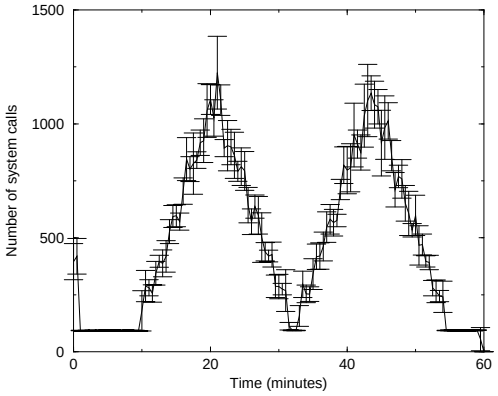

Figure 3 Repeated trials on a simulated load, showing how the average number of system calls varies in proportion to applied load, within measured tolerances. This shows that the basic cfengine statistical model applies to pH also.

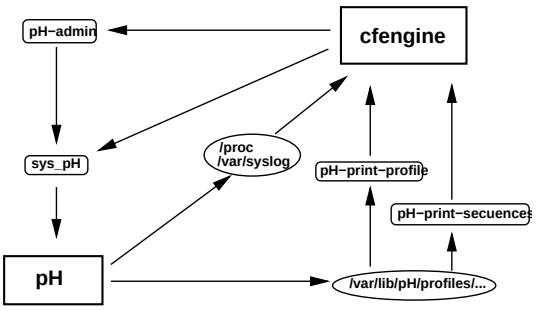

Figure 4 Information Flow Diagram: how cfengine and pH exchange information and management intructions. Communication makes use of existing operating system abstractions, like the /proc filesystem for kernel tables. Similarly, cfengine uses the standard pH API, maintaining the independence of the two systems.

Using pH to measure process load shows us one other thing, that is interesting for future work: the simple measurements show a clear pattern, i.e. the input pattern is reflected linearly, up to a standard error, in the output graph. Monitoring the number of system calls for a process over time, we can determine when it has been used, and how much. We could also build up a statistic here to gather a trend of how much a program is used, and how much we can expect it to be used. By measuring individual sequences separately, it would be possible to perform a code analysis of software, indicating how much users used different parts of the software. This is very interesting for future research.

Modifications to pH

The most important modification to pH, is having the ability to specify what processes to delay. The monitoring will still be done on all profiles, but a variable describing if this process is subject to delaying must exist for every binary. In addition, we must be able to choose if this variable should be set to *delay* or *ignore* by default on the creation of a new profile. If the default value is *delay*, then pH will work as before. The administration of these variables can be handled from cfengine. This enables us to achieve the following: i)Delay all but these binaries (Default on). ii) Ignore all but these binaries (Default off). Note, that the default value could be changed in runtime too.

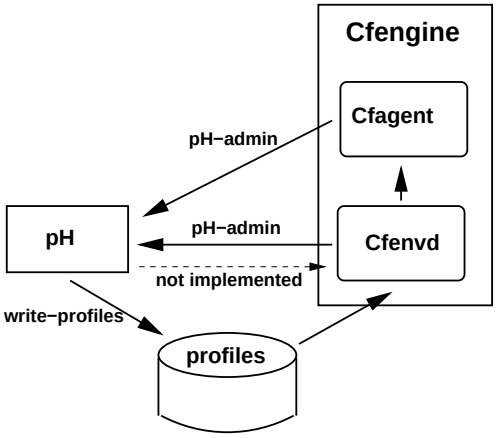

Figure 5 Architectural view of cfengine and pH's collaborative scheme. Both systems are independent of one another and communicate only minimally through profiles held in the kernel.

Data storage

The new pH-related data need to be stored, e.g. the number of abnormal processes, number of system calls for selected processes. The size of the database will vary depending on the number of profiles we wish to monitor and how long we wish to keep the data. `Cfenvd` stores only one week's worth of data, and merges the data together with a average from all other preceding weeks, by a geometric series. This approach would also be useful for data like the number of system calls for a process. It would give us enough to generate the expected usage throughout the week for a given application.

For other data, like the sum of anomalies at any given moment, this variable is a bit more tricky. This variable will be influenced by the use of new applications and has to be monitored over a longer period. Clearly, not all anomalies are genuine and the system must learn to tolerate those that are not dangerous. A one-week local average can be useful as soon as the variable is stable or else the first encounter with all applications will influence the average and deviation so much that small and potentially interesting deviations later on will be unrecognizable.

Cfengine is designed to work independently. An anomaly in the data will trigger an event in cfengine, but we are not always interested in anomalies. We need an option for getting the datasets so that we can view them in plots or analyze them statistically (see fig 5).

6. Example regulation strategy

We envisage automated responses to anomalous behaviour. Such responses have been considered before in other contexts (see refs [11, 9]). A simple example of a `cfagent` response helps in visualizing the interplay between the two anomaly systems. A special `cfagent` class is made to activate on the presence of a recent anomaly. This class persists until it has been expedited by an agent.

Note that pH does not try to start `cfagent` immediately. For one thing, pH is in kernel space, and the agent must run in user space. However, it leaves a

semaphore to the cfengine scheduler to activate the agent with a special class, on its next scheduled run.

If the agent were started immediately as a direct result of the anomaly, it would be trivial to use this in a denial of service attack. Our strategy here is a scaled approach: using cfengine with its normal 'policy' level of statistical uncertainty, and leave pH itself to deter potential attacks with its delaying tactics.

Two classes can become active: a sequence anomaly semaphore, indicating that a potentially dangerous sequence of system calls was identified, and a load anomaly, indicating that `cfenvd` has found the total load being processed by pH is anomalous. We therefore cover qualitative and quantitative anomalies.

```
control:

 actionsequence = ( files processes )

files: // ph_sequence_anomaly::

 bin_bash_sequence_anomaly::

   # Do MD5 integrity check on system files, in case of intrusion
   /usr owner=root,bin checksum=md5 recurse=inf action=warnall

processes:  // ph_load_anomaly::

  bin_bash_high_dev1::

    # Kill the processes causing anomalous load, if it still exists
    ''*'' signal=kill filter=ph_load_filter
```

pH communicates its variables (the list of offending processes) to `cfagent` using one of cfengine's filter interfaces for selecting processes. pH has no functionality for killing a process itself, so this is a natural task for `cfagent` to perform, assuming the offending process is persistent over the `cfagent` scheduling interval.

7. Conclusions

We have measured the behaviour of pH and cfengine and found that they have compatible goals and data models. Cfengine's statistical tools for state analysis complement the powerful pH data-microscope. We have implemented a pilot scheme for combining them into a multi-scaled approach to anomaly detection. Our interest has been two-fold: we were keen to devise a fully autonomic response to anomalous behaviour in computer systems, and were driven to learn more about the meaning of 'normal' and 'anomalous' in the context of the human-computer interaction. We feel that we have made headway towards both of these goals in part, by showing how two such disparate mechanisms can cooperate in a scaled information model. In future work, we hope to study the behaviour of the combined system in a production setting.

How many processes have anomalies? This number would be an indicator of the stability and predictability of a host.

Analysing the behaviour of a binary over time. A comparison of profiles across different hosts could also indicate how similar the different applications

are being used on the different hosts. This has Human-Computer-Interaction ramifications, and is especially interesting for complex programs, such as computer games or office applications, where perhaps only a small part of the program is actually ever used. The relevance here is thus not only system administration, but also software engineering. We could also use these data in a work-routine experiment. When are certain applications being used? Do people use more complex programs at the end of their work-day? The benefit of having the monitoring system separated from the application, is that we can gather these data for every program on the host. Eventually such data can also lead to better management policies.

On the issue of intrusion detection, there are numerous possibilities to explore. Should a host experience a high anomaly on one process it could try to warn other machines on the network about it, by sending inter-host semaphores. In addition, different hosts could interchange profiles. This, off course, implies a secure channel and a protocol for the communication. Today, cfengine offers a framework for the communication, and there is also research going on to define a standard format for intrusion detection (Intrusion Detection Message Exchange Format - IDMEF)[7].

Cfengine and pH alone might not be able to document intrusions on all fronts of the system. They should therefore be able to spawn other intrusion detection and forensic systems on demand if they are present, e.g. packet based detectors like SNORT[13], or Network Flight Recorder[8], that are perhaps too demanding to run all the time. In that way, the combination of pH and cfengine could act as a front-line defense against network intrusion, and vcooperate to switch on forensic capture software and perform backup checks on system integrity. Alternatively they could simply collaborate to identify the appropriate forensic data for human examination. We hope to return to some of these problems in future work.

8. Availability

GNU Cfengine may be obtained from http://www.cfengine.org. pH may be obtained from http://www.cs.unm.edu/s̃oma/pH

References

[1] M. Burgess. A site configuration engine. *Computing systems (MIT Press: Cambridge MA)*, 8:309, 1995.

[2] M. Burgess. Automated system administration with feedback regulation. *Software practice and experience*, 28:1519, 1998.

[3] M. Burgess. Computer immunology. *Proceedings of the Twelth Systems Administration Conference (LISA XII) (USENIX Association: Berkeley, CA)*, page 283, 1998.

[4] M. Burgess. On the theory of system administration. *Submitted to J. ACM.*, 2000.

[5] M. Burgess. Two dimensional time-series for anomaly detection and regulation in adaptive systems. *13th International Workshop on Distributed Systems: Operations and Management (DSOM 2002)*, page 293, 2001.

[6] M. Burgess, H. Haugerud, T. Reitan, and S. Straumsnes. Measuring host normality. *ACM Transactions on Computing Systems*, 20:125–160, 2001.

[7] J. Arvidsson et al. Terena's incident object description and exchange format requirements. *RFC3067*, 2001.

[8] M.J. Ranum et al. Implementing a generalized tool for network monitoring. *Proceedings of the Eleventh Systems Administration Conference (LISA XI) (USENIX Association: Berkeley, CA)*, page 1, 1997.

[9] J.L. Hellerstein, F. Zhang, and P. Shahabuddin. An approach to predictive detection for service management. *Proceedings of IFIP/IEEE INM VI*, page 309, 1999.

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

[11] P. Hoogenboom and J. Lepreau. Computer system performance problem detection using time series models. *Proceedings of the USENIX Technical Conference, (USENIX Association: Berkeley, CA)*, page 15, 1993.

[12] P.D'haeseleer, Forrest, and P. Helman. An immunological approach to change detection: algorithms, analysis, and implications. *In Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy (1996).*

[13] Snort. Intrusion detection system. *http://www.snort.org*.

[14] A. Somayaji and S. Forrest. Automated reponse using system-call delays. *Proceedings of the 9th USENIX Security Symposium*, page 185, 2000.

[15] A. Somayaji and S. Forrest. Automated response using system-call delays. *Proceedings of the 9th USENIX Security Symposium (USENIX Association; Berkeley, CA)*, page 185, 2000.

[16] A. Somayaji, S. Hofmeyr, and S. Forrest. Principles of a computer immune system. *New Security Paradigms Workshop, ACM*, September 1997:75–82.