UNIVERSITY OF OSLO
Department of Informatics

A Distributed Computer Immune System

Kyrre Matthias Begnum

Cand Scient Thesis

January 2003

Contents

Chapter 1

Introduction

1.1 Abstract

This text is about building a distributed computer immune system. It is a master thesis project at the Department of Informatics, University of Oslo, in collaboration with Oslo University College.

Computer Immunology is about the detection and reaction to changes in the state of the computer system. The goal is to maintain system integrity by detecting and protecting against attacks and failures. Its methods and models are inspired from the biological immune system of living organisms. In this project, the aim is to approach such a system by combining two existing immunological approaches: pH a kernel patch for the GNU / Linux kernel and cfengine a high-level policy-based con guration engine. These two systems are independent of each other. By combining them we mean to enable a way for them to bene t from each other by sharing information or communicating events. We hope in that way to learn about the feasibility of distributed computer immune systems and what requirements they have.

The project is divided into three phases. First we try to determine the effect of running pH on a computer system. Second we will explore how these two systems can be combined and if this would lead to a better computer immune system than the two systems working in isolation. Third we will try to de ne a more general model for the components of a distributed computer immune system based on our experience.

1.2 Plan for the remaining chapters

In Chapter 2 we will describe the background and motivation for this project and de ne the problems and how we wish to approach them. Chapter 3 contains the theoretical background for our project. In Chapter 4 we describe the methods we wish to use to deduce characteristics of the system components empirically. In Chapter 5 we present the results of the experiments and give a preliminary discussion of our observations.

Chapter 6 and 7 constitute the second part of the project. Here we investigate how to combine the two systems and what the bene ts of their collaboration would be. In Chapter 7 we implement some of our suggestions.

In Chapter 8 we apply our experience so far to de ne what requirements have to be met by two security tools to be compatible to our idea of a distributed immune system.

Chapter 9 holds a nal discussion on how the project has satis ed our initial problems.

Chapter 2

Thesis Plan

2.1 Introduction

The focus in this master thesis is the concept of a computer immune system [1, 2, 3]. In such a system the de nition of system normality is important [4]. This concept is often referred to as de ning a sense of “self” and of an autonomous system. How can a normal state for a system be de ned, and how can that system keep that state by itself? Normality is interesting because we want systems to be predictable [5]. These questions require much research to be answered fully. A beginning is to build a system which can gather as much information from the OS as possible. Further, this system must then be able to analyse the information, identify a state and nally react in a way that will either keep the state or change it.

We will try to approach Computer Immunology from the perspective of a distributed security and integrity system. There have been several projects which try to build systems that specialise in detecting intrusions and anomalies. All have different algorithms and de nitions of normality and many succeed in their domain. This resembles a biological immune system, where we nd tiny components who are all experts in their domain. System administrators often install several security tools on their system.

What remains in computer immunology, is the collaboration of these tools. Is it possible to have an immune system for computers that works like a distributed set of small software components that interact, even if they where not designed to do so initially, to secure the integrity of the system?

This question cannot be answered without many more fundamental questions being answered rst. Can this distributed model actually ensure the integrity any better then these components by themselves? What are the bene ts of this collaboration and how does it take place? How can we at all evaluate whether two or more tools are compatible and useful in such an immune system? We will approach these questions with two security tools, pH and cfengine, both developed in the spirit of computer immunology.

PH (Process Homeostasis) is a patch for the 2.2.x GNU/Linux kernel [6], that enables the kernel to track system calls generated by all processes and analyse these traces using a pattern matching algorithm. It also generates a pro le of every binary that is being executed. A binary, often called program or application, is the set of instructions that the process can execute. When we choose to run a program, we execute the binary and get a running process. Many of these instructions are so-called system calls - methods made available by the operating system, like opening and reading les. The possible sequences of these calls for every binary can change from execution to execution, but by logging them, we can establish a pro le for each binary based on system-call pairs. [7]

If a process should generate system call patterns that do not correspond to the pro le, the process will be delayed using a time-delay algorithm. This is meant to sabotage intrusion attacks which are based on executing alien code (buffer over ow). Using a special system call, pH's reaction sensitivity can be adjusted in runtime.

PH is a process based anomaly detection system (ADS) with the ability to react. It uses machine learning techniques to build it's pro le. The same method, only on a different scale, is used by Cfengine - An autonomous agent and a middle to high level policy language for building expert systems which administrate and con gure large computer networks. [8] Cfengine has a module for detecting anomalies based on system variables and the ability to de ne a reaction pattern to a given anomaly event.

First we want to analyse the pH-system. Not in the sense of it's ability of stopping attacks, but a rather different angle: How will using a system like this affect the operating system? Why is this important? This brings us back to the network of smaller "immune systems". We want to incorporate pH into the anomaly-detection framework of cfengine. Also, being in kernel-space, pH offers a opportunity to monitor processes closely, and we want to make cfengine able to use this functionality. Before that, we need to nd out if it is at all feasible. Cfengines anomaly-detection model relies on analysis of time-series data. Should pH introduce too much noise or unpredictability into this analysis, the two systems would not help each other at all. The rst phase of this project can be viewed as an control experiment.

After sorting out which variables are usable for our analysis, we can focus on how to best combine these two systems from a design perspective. This is the other half of the project. PH offers information about the immediate state of a process, both in respect to used system call sequences and system call count. This information can be accessed and logged from another system to enable precise tracking of process activity within the computer system. Many of today's intrusion-detection algorithms that use data analysis focus on other variables, like frequencies of network traf c or network protocols [9, 10, 11, 12] and are often called network intrusion detection systems. Different statistical analysis methods are used in intrusion detection [13]. Our aim is not to test these, but to investigate if there are new variables available which can be incorporated into cfengines statistical component.

The project requires programming and modi cations to both pH and cfengine. It does not aim towards a optimal development solution as in a bug-free and stable system. This would not be practical at the current time since pH is only available for Linux 2.2 and further testing is required to evaluate the usefulness of this idea. That may be developed in future work. Another important goal for this project is to suggest what is required of two ADSs in order to to make them part of an computer immune system. These requirements can then be used as an analytical base for evaluating the compatibility of other ADS in the future. The model will be strongly in uenced by the types of systems we evaluate, but it may serve as a template which can be expanded as we investigate other ADS systems in future projects.

The resulting questions are therefore as follows:

Central Questions

1. How does a system with pH running differ from a pH-free system on an overall system perspective? (control experiment)
   - (a) How can we nd a basis on which to compare two apparently similar computer systems with regard to system variables and low uncertainty in measurements?
   - (b) What can we say from this about how much any system differs under different conditions and what is tolerable? (Anomaly detection)
   - (c) Would the difference between two computer systems increase if one of the systems ran a pH-patched kernel?
   - (d) Can any of the information from the previous problem be used to detect anything useful in the behaviour of computers? If so, what can we use the information for?

2. How could a process based anomaly detection system be incorporated into a more generic con guration engine like cfengine?

   (a) In what way does cfengine offer an interface for communicating with pH and vice versa?

   (b) What bene ts would a higher level system con guration-tool have on pH and vice versa?

   (c) How could a pH-based event-handling approach t into cfengines policy-based anomaly detection framework?

3. What are the requirements for the development of a computer immune system?

   (a) What makes two or more ADS compatible to this model?

   (b) Can pH and cfengine be used as two components in a computer immune system?

2.2 Problem Specification

Here consider how to approach the problems and what kind of results we regard as suf cient for the project.

2.2.1 Phase One: System Similarity/Difference

The uncertainty between two similar machines is important from a computer immunology perspective. How does adding pH to one machine alter its baseline “normal” behaviour? Is pH intrusive and does it affect performance? Our rst phase is a control experiment where we will try to answer these questions.

The difference between two systems can only be measured by considering a set of speci c variables. We de ne difference and similarity as a statistical correlation between a selection of system-wide variables of two computer systems given a time-series sample of those. Further, we don't want to see how big differences (low correlation) we can achieve. What's interesting in this problem is how small differences are between twin systems.

The variables of choice will be standard system variables describing resource utilisation (See [14] for instance). These variables are on the OS-level, like memory usage, CPU load and disc activity. There are other variables we could take into account, like the number of processes and number of users. Cfengine uses some of these already. Since the test-environment is controlled in that aspect we will not focus on them. We could also monitor network traf c, but that has no importance in our setting, since pH is not engaged in any network communication. The data-gathering is separated from the analysis. This way we can expand our data-model more easily with new variables, should we not be satis ed with our initial selection.

Our main analysis method of choice will be Spearman's rho, a correlation coef cient for two independent variables which are not normally distributed. It shows the linear dependency between two variables.

The data will come from two appointed machines with an identical setup. By “different conditions” we mean small additions to the two machines, like running pH on one machine and not on the other, which will result in less similar con guration. This will affect both le-system and performance, but not the number of processes or users being active. A time-series measurement will be done after each of these changes. We want to measure the noise-level so it can be taken into account. We don't want the machines to do very much initially. In the end we want both machines to run a service that we can stress from the network and one of them to run a pH-patched kernel.

We then analyse the data from these tests, and look at how the correlation coef cient changes as we introduce more differences. A satisfying result would be to see if we have a statistical way of detecting if the systems are in fact using system resources differently. A limitation is that we only investigate the relationship between each variable in isolation. The variables describing the computer system are often inter-dependent.

There are different ways of de ning a multivariate statistical pro le for a dataset [13], but these methods are used for anomaly detection and not for comparing two datasets. One application of such methods in this context would be to build a multivariate pro le of one of the sets, and use it on the other dataset to see if it would be accepted. A simpler approach could be to do a multivariate test for each set on the relationship of the variables in it. The result for the two sets could then be compared. We choose the novel approach of analysing the dependency of each variable in isolation because a multivariate test cannot show what variables in the set differs the most. We are interested in following the evolution of each variable during the different tests.

The tests can be continued for every system-variable, aspect of usage and each service. But given the time constraints we will keep to only this set of variables. The design should nonetheless be good enough for others to repeat them in different contexts.

2.2.2 Part Two: Combining pH and cfengine

Once we have measured the uncertainties in the experimental procedure, the focus shifts towards the variables that come from pH. We need to nd out how to best preserve and analyse these variables. An important goal, is to see if pH introduces new variables that are at all useful for statistical anomaly detection and if they are compliant with cfengines framework. Hopefully it will give a better perspective on the use of system resources. We will concentrate on the machine running pH. If we introduce a load to a service running on that computer, will the new variables give us a accurate account of what happened? We hope so.

As said earlier, the interest lies not in making one system out of both, but merely opening a couple of doors so that the two systems can take advantage of this through more interrelated teamwork. One of the advantages will lie in the sharing of data.

Several questions need to be answered for this to happen. First of all, do they have a compatible data-model? What kind of data is important? And how should the two systems communicate or affect each other? The main goal for this part is to establish a usable speci cation for the changes that need to be implemented in both systems. We will use code inspection and more experiments to learn more about the new data which can be gathered and analysed in a way that is already implemented in cfengine. A system has to be designed for collecting and analysing these data, before we enable cfengine to do it. If time allows, then some prototype programming on cfengine and pH will nish this part.

This part is the actual step towards a Computer Immunology System. The rst part was the groundwork and the last part will be to discuss how our results can be used for other projects.

2.2.3 Part Three: Building a Computer Immunology System

If we succeed in combining the two systems, how could this be useful for other similar projects in the future? There are many useful tools available today for detecting anomalies and intrusions. We will try to formulate a requirement speci cation which can be used to evaluate the compatibility of other systems involved in building a computer immune system.

Now that we have described the goals of the project, it is important to ask why we choose to design the project like this. The answer lies in the philosophy of system normality and computer immunology. Our aim is to explore idea of combining different anomaly detection systems and making them work like an interrelated immune system. The rst part of the project can seem a bit unrelated to the later problems, but it constitutes an important groundwork for the project as a whole. Cfengine detects changes in the state of the system. PH operates in kernel-space and can therefore in uence the performance of the OS considerably. If pH introduces to much noise in cfengines data, then the two systems are incompatible.

The larger aim of this project is to apply an empirical method to the evaluation of collaborating systems. What results can we nd to determine whether such collaboration is bene cial or not? Our experience from this project will hopefully give us more insight into the complexity of anomaly detection systems which try to monitor and react on all aspects of the computer system, and how to build them.

This project does not cover the software engineering aspects of building a computer immune system. There are many interesting questions regarding maintainability, development process, version control and quality assurance. We will make a few notes about them in the nal discussion at the end of this text.

Chapter 3

Theoretical Background

3.1 Intrusion Detection

We often divide Intrusion detection into two categories: misuse (or rule-based) intrusion detection and anomaly intrusion detection [15]. In misuse intrusion detection we have an already de ned set of so-called signatures that depict an attack. This approach is sometimes referred to as signature based intrusion detection. An example of this type of system is `portsentry`, a package for detecting ports-cans [16]. Another famous open source intrusion detection system is `SNORT`, which tries to detect network attacks by monitoring network traf c in real-time [17].

In anomaly intrusion detection, we rst have to de ne or learn what behaviour is to be regarded normal or benign for the individual system. Any future deviations from the norm will then be regarded as an anomaly. Behaviour in this context can mean different things: usage patterns of services, user behaviour or a combination of system variables are all valuable sources. An anomaly intrusion detection system is often viewed on as a learning system. Data are gathered and analysed constantly in order to de ne what is normal using a statistical approach. The two anomaly detection systems in this project, `pH` [18] and `cfengine` [19] fall under the category of learning systems. They both build pro les and compare new data to a dynamic norm.

Statistical techniques can be used to build a pro le of a subject's long-term normal behaviour, and detect signi cant deviations of the subject's short-term behaviour from the long-term statistical pro le [13]. A subject can be a process, program, user, OS or anything else that can be measured. The pro le can have been built in advance from audit trails or updated constantly as more data is available.

In ref. [9], the authors describe an approach to anomaly detection based on transactional properties, where network traf c is considered normal if it coheres to the known ACID[1] properties of transactions. Every network protocol is described using a deterministic nite state machine, DFSM. A monitoring system could check the connections consistency using the appropriate DFSM. Some network attacks, like Ping of Death violate the ACID properties and could be detected if monitored this way.

When speaking of anomaly detection, it is not always certain if this term only covers anomaly intrusion detection or has a more broader view, where anomalies can come from miscon gurations, faults or other deviations which do not have anything to to with any intrusion attempt [5].

Both approaches have their advances and limitations [20, 21]. Misuse intrusion detection systems have to be updated when new signatures become known. On the other hand, they are more ef cient and accurate. Anomaly intrusion systems can generate false alarms or ignore actual attacks. Many systems try to use a combination of both approaches. `NIDES` is an intrusion detection system which uses both signature-based methods and statistical pro les [22, 21]. It monitors network traf c as well as individual user behaviour.

3.1.1 Related Research

Ref. [23] describes an immune system to protect against new versions of different types of viruses. The motivation is the increasing frequency of new computer viruses and that one needs a faster and automatic way of creating xes against them. Their system is part of IBM's AntiVirus (as of 1997). This system could create xes for viruses and spread them to other machines, thus creating an immune system for cyberspace.

In [24], the authors approached intrusion detection by de ning unacceptable events for processes in advance. These events where sequences of system calls and their parameters. When executed, they where monitored in real-time. A speci cation language was developed to both describe detection criteria and reactions to the violations.

Ref. [15] describes a similar approach to pH in that it is a learning system. Their approach is to divide the softwares functionality into modules and to generate a behavioural pro le from the softwares transitions between these pro les when executing. This system does not have any reaction capabilities, though.

[1]*ACID* stands for atomicity, consistency, isolation and durability

3.2 Computer Immunology

Todays computer systems are fragile and unreliable. Operating systems may be called stable or secure, but they all depend on the human to con- gure, monitor and repair them. Computer systems pervade almost every aspect of our daily life. The more computers, the higher the probability of one of them breaking down and demanding a repair. The task of system administration grows not just in the technical requirements needed to keep a large system healthy, but also in the risk involved should something go wrong. Biological systems have a greater complexity and are also fragile and vulnerable, but they possess the ability to detect errors and heal themselves.

Every computer system, regardless of how small, needs time-consuming administration. Systems that fail can have an impact on our lives, like sensitive information available for all, open bank accounts and computer hang-ups. Despite this we just get more and more systems to do important tasks on our behalf, like banking. How could we apply the the idea of an autonomous system to our computer systems, minimising the need for human interaction? Then, perhaps, we could use the computers only for the tasks and services they where intended for instead.

Computer Immunology is an approach to integrity management. In the hart of the Computer Immunology philosophy is the autonomous system. One of the smallest autonomous systems we know today is the procaryotic cell. Bacteria without a cell nuclei. Seemingly primitive, these cells have complex mechanisms for transcription of DNA into RNA dictating the construction of complex proteins from amino acids. Other proteins detect errors in the DNA and order the cell to commit suicide. These systems can detect, react and adapt to conditions which could be lethal otherwise.

Another example is the managing of our defence-cells, lymphocytes, or generally white blood-cells. They attack intrusions of non-self substances and foreign cells. The lymphatic system can be “trained” for detecting speci c bodies not previously known to the immune system before. The production of speci c lymphocytes varies with the demand for them.

A key concept here is that of homeostasis - or regulative feedback [1, 3]. 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. On the other hand, positive feedback would tend to amplify the change. This has a de-stabilising 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 built-in, automated property of a system that executes and monitors events essential to the existence of the system. It is 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.

This state of “self” is the foundation of anomaly detection. It is an acquired sense, meaning the system has to de ne this state based on its own behaviour and adjust it accordingly [1]. We are speaking of a machine learning system or learning-system. The problem with such systems is that they can be inaccurate and oversee dangers (false negatives) or react to normal situations (false positives). The Computer Immune System needs a reliable framework and data sources so that the de nition of normality can be as accurate as possible.

Our immune system consists of small and primitive participants. We have white blood cells tagging and destroying illegal cells. Our perspiration adapts body temperature and we even have a system for the collection and removal of toxic chemicals. The point is that this immune system is a collaboration of subsystems, most of them transparent and simple. It is because we have such a subsystem for almost every anomaly (or illness) that we survive. These subsystems work together unknowingly. They interact in the sense that one's reaction pattern sets off others alarm bells. Could the same be possible for a computer system?

Computer Immunology is thus about normality, a de nition of a state for the system. Without the sense of what is allowed or expected values for i.e system variables, we could not detect an anomaly. These values are of course subject to variations within acceptable limits and the mean will have to be adjusted after time of day and weekday. If the system has the tools for measuring and adjusting these values by itself, we have a learning system. In such a system a statistical framework suitable for the different types of data is needed. Doing statistical analysis in runtime can be too time-consuming and would probably be done periodically instead. See also the real-time approach by Burgess in ref. [25]

Another aspect of normality is the de nition of a policy. Every event that violates this policy is an anomaly. An example of such a policy could be what permissions should be set on system les. Policy is also combined with statistical measurements. One could de ne what relative deviation from the established pro le should be perceived as an anomaly event or the frequency of a certain error to occur. Policies also appoint when and how often statistical measurements are to take place, affecting the granularity of the learning system. Normality is thus about both policies and learning.

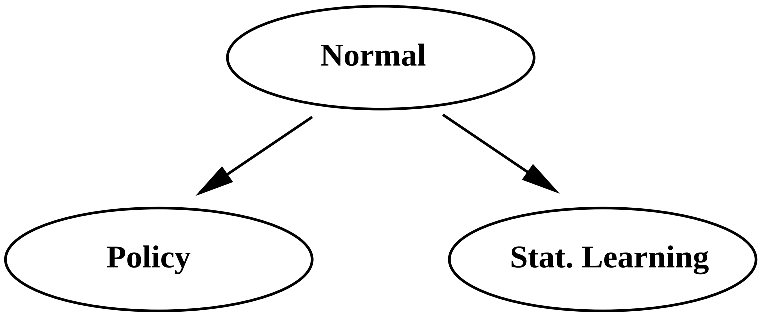


Figure 3.1: The definition of system normality consists both of a set of rules (policy) and learning capabilities (statistical measurements).

The idea of the reasoning computer or the emotional robot is known to all readers of science fiction literature. It is an old concept not only forbehold computers. Frankenstein's monster and clay golems have had their share too. Today we call it artificial intelligence. Immune system also has intelligence. It exhibits learning, selection, state and responsiveness. It solves a complex search-problem. However intriguing and romantic the thought may be, this is not Computer Immunology. We try to keep the focus on security systems, not fiction. Learning techniques, although prone to false alarms are needed to make computer systems adaptive to new threats. Signature based detection techniques can be used to give computers *a priori* knowledge of existing vulnerabilities.

3.2.1 Computer Immunology and System Administration

Today there are few software components with the task of detecting, reacting and perhaps even adapting to conditions in the computer system. One example of such a mechanism is a kernel thread which checks for deadlocks between other threads and processes. A more sophisticated example would be a program that records the load on the OS continuously and adjusts the time of periodic tasks to times when the system usually experiences low workload. It could also run optimisation algorithms, like prefetching important files in advance to reduce expected peaks of user interactions.

Introducing extra functionality into a operating system has always been a subject of debate. Arguments usually concern simplicity, predictability and performance. A too complex and unpredictable system can be a security risk in it self. [26]

Using the idea of an autonomous system consisting of several independent but interacting components, we could imagine a computer system having many processes or threads each doing a highly specified and simple task most likely related to detecting some sort of anomaly. These components could trigger each others alertness by sending messages or leaving hints.

We have many types of anomaly detection systems [27]. Most of them are aimed towards security, like detecting network intrusion [22, 17, 16] or ltering out viruses from incoming mail [28, 29]. None of them are especially aware of each other. These applications are developed by different programmers and companies. This could look like an ideal starting point for building an immune system.

3.2.2 Thoughts on Computer Immunology

Using terms like anomaly, immune and virus it's easy to get a too personalised view of the machine. We could de ne sickness or health for a computer, but would that bring us further towards better system administration? Computer Immunology is not about making the machine resemble an organism. It's about using techniques derived from one of nature's oldest instincts: To stay alive. One of these techniques is load-balancing. Another one is running away from danger. We could make a server stop unimportant services when the load gets too high. We could also make the system shut down it's network-services if someone should try to exploit it.

Shutting down services can be perceived as hiding from danger just like organisms do, but resembling a life-form is not the goal in itself. It's about programming certain patterns of reactions and responses. But right now we have no other place to look for reactions that we can understand and apply than in nature. Where else in the universe do we nd decision-making and policy if not in living organisms?

Although subtle, there is a difference in making a huge and expensive Tamagotchi, a household pet if you like, and programming the computer to react to values of prede ned variables. The difference lies in our perception and the system and its purpose. Both the server and the Tamagotchi react in a pattern that we recognise from nature, but while the later tries to mimic an organism through well crafted rules, the former is just applying a set of reactions that we know will make the system more stable and self-maintainable, an “emergent” property.

Ref. [1] has an interesting thorough description of the fundamental properties of a computer immune system.

3.2.3 Related Research

Different projects have often different views on what event constitutes as a anomaly, but they are similar on on the notion that computer systems are healthy when their behaviour is free of anomalous occurrences[1, 3]. It falls on researchers to de ne 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[5]. 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 clearly a statistical concept, which accrues over time, and computer immunology is a form of computer learning[4, 30]. Unlike many other methods of arti cial intelligence, computer immunology is about purely mechanical regulation of behaviour, rather devoid of `intelligence' in the normal sense of the word.

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[31, 6] learns new signatures over time, but is resistant to doing so. The primary motivation here has been in de ecting network intrusions, though the method is equally effective in detecting abnormal local usage, such as attempts to exploit buffer over ows. 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 con guration management aspect of policy, using a system of agents (cfengine) that detect and use their environmental conditions and current con guration to detect anomalous changes[32]. Again, the policy is partly speci ed and partly learned from patterns of usage, and the response to different events is speci ed itself as a matter of policy, and the agents ensure that the system tends to maintain the same state over time.

3.3 Introduction to pH

PH (Process Homeostasis) is a patch for the Linux 2.2.X kernel developed by Anil Somayaji. It addresses the automated response problem. A lot of security software today is designed to detect attacks (i.e IDSs) or to nd vulnerabilities in the system. Many of them can even try to stop attacks as they occur. Other can delete viruses and even repair damaged les.

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 pro le of each binary as to see if each process produces the expected patterns of system-calls. While the the process keeps it's number of strange patterns under a certain level, it is considered normal. If the number rises too high ("high" is an adjustable value), pH starts to sabotage the process by delaying all the system-calls made by it.

An important point, is that the pro le is for each binary, but the reactions are for individual processes. Every process has its sequence of system calls, which we call a trace. The pro les of each binary is updated and adjusted to the behaviour of the processes. This means that instances of a speci c behaviour will in time be considered as normal. Not all anomalies are real threats to the system. Earlier research by Somayaji suggests that "To date, all of the intrusions we have studied produce anomalous sequences in temporally local clusters." pH is therefore designed to react regarding the density of anomalous system call patterns.

3.3.1 Algorithm

The algorithm used by pH is called "time-delay embedding", which 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.

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.

Starting with the rst call, `getpriority`, we see that it is followed by no other calls (the rest of the window is empty). The system call `open` however, is followed by `getpriority` and ,later on, by `close`. We now have two pairs: (`open`,`getpriority`) and (`open`,`close`). Should pH ever encounter a new `open` in the trace of this process that is not proceeded by neither `getpriority` or `close`, then pH would call it an anomaly.

This is a highly simpli ed model of pH. First of all, we only considered pairs of system calls coming directly after each other. This gives a window of size 2. The default window-size for pH is 6. This gives even more pairs generated by each trace. Table 3.3.1 on page 22 shows the pairs of this trace given a window-size of 4. Second, pH does not consider each pair it encounters as part of the processes pro le right away. There is a distinction between the current pro le (called test) for a given binary and the temporary pro le 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):

1. The user explicitly signals via special system call (`sys_pH`) that a pro le's training data is valid.
2. The pro le anomaly count exceeds the parameter `anomaly_limit`.
3. The training formula is satis ed.

The training formula is actually a set for constraints that the training array has to comply with. These constraints are the length of the training array, how long (in system calls) the training array has been left unchanged and the difference of these two. Should the training array ful l the minimum requirements set by pH, then the training array will be copied to the test array automatically.

The formula consists of three variables [6]:

```
train_count:    # calls since array initialization
last_mod_count: # calls since array was last modified
normal_count = train_count - last_mod_count
```

The conditions for the automatic training to test copy is as follows:

```
last_mod_count > mod_minimum
normal_count > normal_minimum
(train_count / normal_count) > normal_ratio
```

Should a anomaly occur, then a number of the following system-calls will be delayed. The amount of time they will be delayed is dependent on the number of anomalies in the last 128 system-calls. The word time here means jif es - processor ticks. The delay is calculated using this simple exponential formula:

$$delay = \texttt{delay\_factor} \times 2^{LFC}$$

The default `delay_factor` is 2. LFC (Locality Frame Count) is the number of anomalies in the last 128 system calls. A anomaly, will therefore effect the next 127 system calls, because that is how long it is going to effect the LFC. Let us consider the following example:

A process generates two anomalies coming directly after each other. The rst anomaly sets the delay for the next 127 system calls to 4. The second anomaly increases the delay to 8 for the next 126 system calls before it goes down to 4 again for a single call.

Current	Position 1	Position 2	Position 3
getpriority			
open	getpriority, close	write	write
write	open, write	getpriority, open	getpriority
close	write, pipe	open, write	close, open
pipe	open	close	write
exit	close	pipe	open

Table 3.1: System call pairs given the following trace `getpriority, open, write, write, close, open, pipe, close, exit` and a window-size of 4

After a certain amount of anomalies, the train-pro le will switch to the test-pro le. This is called tolerization, meaning the pro le adapts to the behaviour of the process. Default value is 30 anomalies. But should the anomalies occur too close to each other, then pH will react in the opposite manner and reset the train-pro le. Default limit is 12 anomalies within the LFC.

Also, should a process reach a higher LFC then 10, then every `execve` call will be delayed[2] for two days (by default).

3.3.2 Implementation

When a new process starts, the pro le for the binary is loaded from disk. This pro le contains both the test and training array. If a pro le does not exist, it will be created. When the process terminates, the pro le gets written to disk. Several simultaneously running processes from the same binary share the same pro le, and it will be written do disk when the last process terminates.

Before each system call, a function (`pH_process_syscall`) in the kernel is called. This is where monitoring, response and training logic takes place.

It is also possible to control pH at runtime with its own system call: `sys_pH`. Together with the patch comes a tool which is called `pH-admin`. This is basically a front-end to the system call. This tool can, among other things:

- Turn the monitoring on / off
- Write pro les to disk
- Adjust pH-variables (i.e `delay_factor`)

[2]This is a new feature in pH-0.18. Earlier versions aborted the `execve` call instead.

- Force the train profile to be copied to the test profile.

pH gathers it's 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. As an example:

The program `less`, which has the path `/usr/bin/less`, will have it's profile at `/var/lib/pH/profiles/usr/bin/less`.

3.4 Short introduction to cfengine

Cfengine is a configuration management system written at Oslo University College[32], which is comprised of a number of components (see fig 3.2). An agent component is responsible for enforcing specified policy by comparing a description of the permissible 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[25], 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 in relation to what has been learned previously, in units of the tolerance defined by 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 classified data as a `class 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 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

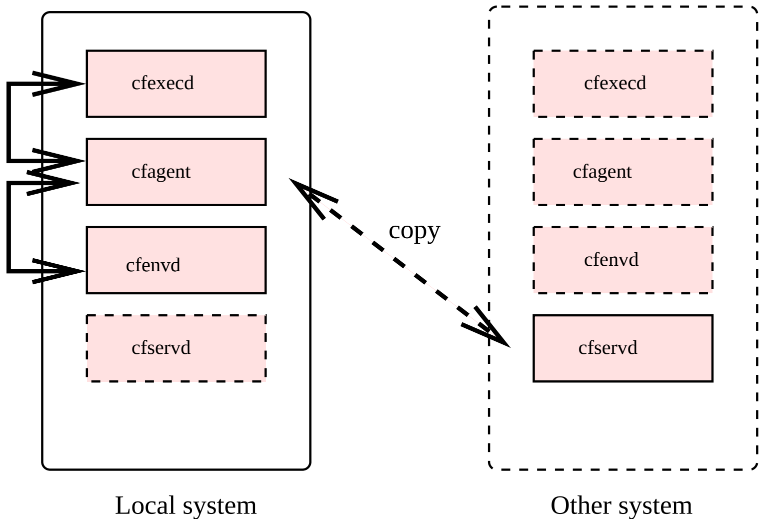


Figure 3.2: **A schematic r**

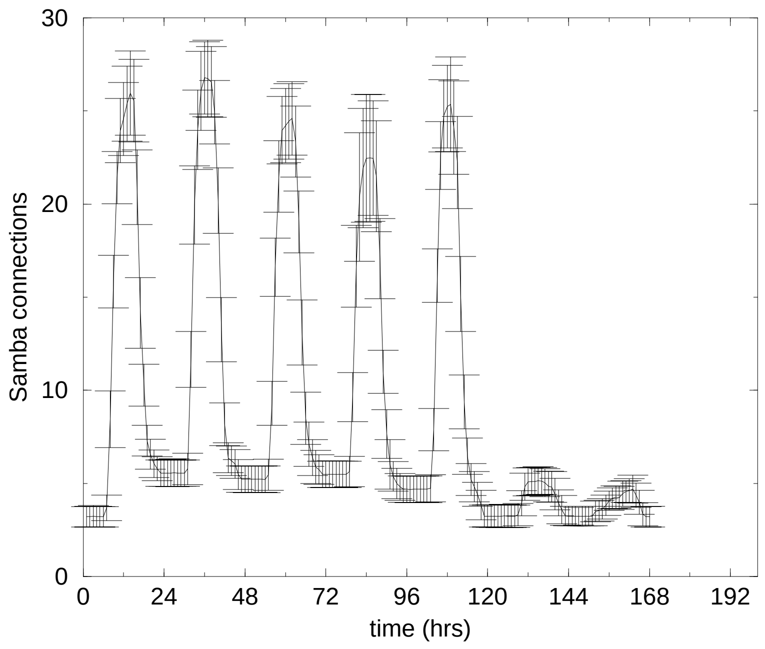


Figure 3.3: **Cfengine measures patterns of resource usage over the working week. This example shows how measurements of Samba (Windows le sharing) service requests lead to an average picture of behaviour at different times of the week. The solid line is the average value over many weeks and error bars indicate the standard deviation.**

Chapter 4

Experimental methodology

This experiment is the rst phase of the project and addresses the rst part of the problem. We want to establish a measurement for how similar two computer systems are. Here it will actually be the comparison of two datasets, one from each system. We will gather the data from two machines during four trials. In each trial, the systems will differ a little more. The results gathered from this experiment will allow us to determine the error-margins for two twin systems. If the two systems should differ even if they are almost identical con guration-wise, then this would in uence our expectancy of what would be a signi cant difference later on during the project.

Both hosts in the sample are running the Debian GNU / Linux operating system with identical con guration.

Each host was equipped with a single process in addition to the normal process list which used vmstat to collect data about system variables every 30 seconds. These data were written to a at text le which was timestamped both in the beginning and the end of each trial. During the rst trial, we used the standard kernel (`2.2.19pre17`) that was default in the Linux-distribution and without pH installed. The goal was to get some data on how different the systems are on identical installations. In the later trials we used a different kernel, 2.2.19, on both machines. These were also identical except that one of them had a pH-patched version. The reason we switched the kernel was because this version was being used by the developers of pH. Now we wanted to collect some new data on their behaviour. First with pH deactivated and then activated. During all of these tests, the machines where doing "nothing". In this context “Nothing” means that no users were logged in during the test period save for starting and stopping the monitoring. They both ran a `ssh` service for remote access.

This is the result of the command `ps aux` on one of the machines. It shows the processes running at the moment.

```
USER        PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
root          1  0.0  1.5  1020  464 ?        S     2002   0:06 init [2]
root          2  0.0  0.0     0    0 ?        SW    2002   0:00 [kflushd]
root          3  0.0  0.0     0    0 ?        SW    2002   0:01 [kupdate]
root          4  0.0  0.0     0    0 ?        SW    2002   0:00 [kswapd]
root          5  0.0  0.0     0    0 ?        SW    2002   0:00 [keventd]
daemon       79  0.0  1.6  1140  500 ?        S     2002   0:00 /sbin/portmap
root        139  0.0  2.0  1352  620 ?        S     2002   0:02 /sbin/syslogd
root        142  0.0  2.9  1484  892 ?        S     2002   0:00 /sbin/klogd
root        153  0.0  1.9  1312  572 ?        S     2002   0:00 /usr/sbin/inetd
root        164  0.0  3.1  2240  944 ?        S     2002   2:15 /usr/sbin/sshd
daemon      168  0.0  1.8  1140  548 ?        S     2002   0:00 /usr/sbin/atd
root        171  0.0  2.0  1168  620 ?        S     2002   0:00 /usr/sbin/cron
root        176  0.0  4.3  2520 1284 ?        S     2002   0:01 /usr/sbin/apache
root        179  0.0  1.4  1004  444 tty1     S     2002   0:00 /sbin/getty 38400
root        180  0.0  1.4  1004  444 tty2     S     2002   0:00 /sbin/getty 38400
root        181  0.0  1.4  1004  444 tty3     S     2002   0:00 /sbin/getty 38400
root        182  0.0  1.4  1004  444 tty4     S     2002   0:00 /sbin/getty 38400
root        183  0.0  1.4  1004  444 tty5     S     2002   0:00 /sbin/getty 38400
root        184  0.0  1.4  1004  444 tty6     S     2002   0:00 /sbin/getty 38400
root      21029  5.6  4.9  2856 1480 ?        R    14:13   0:00 /usr/sbin/sshd
kyrre     21030  2.5  3.8  1968 1160 pts/0    S    14:13   0:00 -bash
kyrre     21031  0.0  4.0  2916 1200 pts/0    R    14:13   0:00 ps aux
```

Each test was run simultaneously on both machines. We place special care in having the two systems as much in the same state as possible just before a experiment starts. One way to achieve this, is to reboot the machines before each trial. This gives us two similar systems with regard to running processes and memory-usage. This is just as important when the goal is to test the machines under stress. The load has to come from the same processes. “Load” itself can be divided into different variables: Memory, CPU, IO. Running one process that only uses a lot of memory and waits for user-input would not stress the CPU in the same way as running a lot of small processes who all run in the background.

During the rst three trials, we have hopefully already got an impression on how the system variables vary as long as the machines have no particular tasks they have to attend. Next we want to do a similar test, but this time we introduce a load to a process during the testing period. We use the same measuring and analysis and hope to see a clearer deviation then before.

In this experiment we monitored the evolution of a pH-pro le belonging to a speci c application over time (one week). This application will run on both machines and they have the same random user input. The overall system variables on both machines will be monitored on both machines in the same way as before.

The application to be monitored was the apache web-server. Apache is a well-known web-server for many operating systems. It it widely used together with Linux. Before the experiment starts, the pro le of this program will be deleted so that pH will have to start on a blank one. Every ve minutes a snapshot will be taken from this pro le and put in a log- le. The system itself will be monitored with vmstat every 30 seconds.

A remote process will download webpages from both machines and sleep a random amount of time. Several such processes and short sleeping intervals will strain the system somewhat. The script is shown below:

```
#!/usr/bin/perl
# A simple script for downloading a webpage from
# romulus and remus and prints the event
# The script sleeps a random amount of time.
# $max holds maximum sleep time

$max = $ARGV[0];
srand;

while(true)
{
$return1 =  system ("lynx -dump http://romulus.iu.hio.no >> /dev/null");
$return2 =  system ("lynx -dump http://remus.iu.hio.no >> /dev/null");
$time = localtime;

system ("echo $time >> surf.log");

$tall = int (rand($max)) +1 ;

print "Surfed at $time, sleeping for $tall seconds...\n";
sleep $tall;
}
```

We will in the end have three log- les:

```
vmstat.log.romulus
vmstat.log.remus
apache.log.pH.romulus
```

These will dowloaded from the systems and analysed off-line. The two vmstat- les will be compared in the same way as before, and should there be any differences in them we hope to nd an indication in the pH-log- le that pH's treatment of apache has some connection to it.

The log- le `apache.log.pH.romulus` is updated every ve minutes, and we use `cron` for the execution of a `perl` script that parses the pro le and put it in a data le.

It's important to note, that there will be one new difference between the two systems. The script for gathering pH-related data on romulus does not exist on remus. The script forces pH to write pro les to disk, reads from a pro le and writes to the data le. This repeats every 5 minutes. This can very much in uence the data. One possible compensation could be to run a script on remus every 5 minutes as well. This bogus script could then do some writing an reading on it's own, just to compensate. We chose not to implement this.

Another approach is to measure the systems during user interaction. This can be tricky to implement, since the systems need the same type of

Con guration / Trial	Description
Default	Both machines run the same installation-default kernel
pH-Passive	2.2.19 on remus and 2.2.19-ph on romulus. PH is disabled.
pH-Active	Same as pH-passive, but pH is enabled on romulus
pH-Load	Same as pH-Active, but the machines are running apache with instances of high load.

Table 4.1: Every trial had a different con guration. This table shows the four trials and each corresponding con guration.

Trial	Machine	Con guration	Samples collected
1	romulus	Default	179229
1	remus	Default	179282
2	romulus	pH-Passive	51173
2	remus	pH-Passive	51189
3	romulus	pH-Active	51091
3	remus	pH-active	51106
4	romulus	pH-Load	40269
4	remus	pH-Load	40293

Table 4.2: Different trials

interaction. Simulation of user activity seems to be the solution, but this isn't easy. First of all, its hard to simulate user activity (like mouse-clicks and doodling) in the X window-system. As a result we tend to run scripts on the machines that simulate a user typing commands. But what type of user? An experienced user with a UNIX background would use a different set of commands and possibly fewer misspellings than a rst-year student with no former knowledge of command-line interaction. What type of editor would they use? What type of commands? One solution to this would be to use transcripts from different students history- le (i.e .bash_history) and generate shell-scripts that encompass all the different ways of command-line usage. We do not follow this approach. The data collected in this project would be an interesting comparison to data from systems with user interaction. Future projects may address this issue.

An interesting point will be if the variables described earlier will be able to show the effects of pH. In other words: can we conclude that the system in fact is unaffected just because we don't nd any signi cant changes in

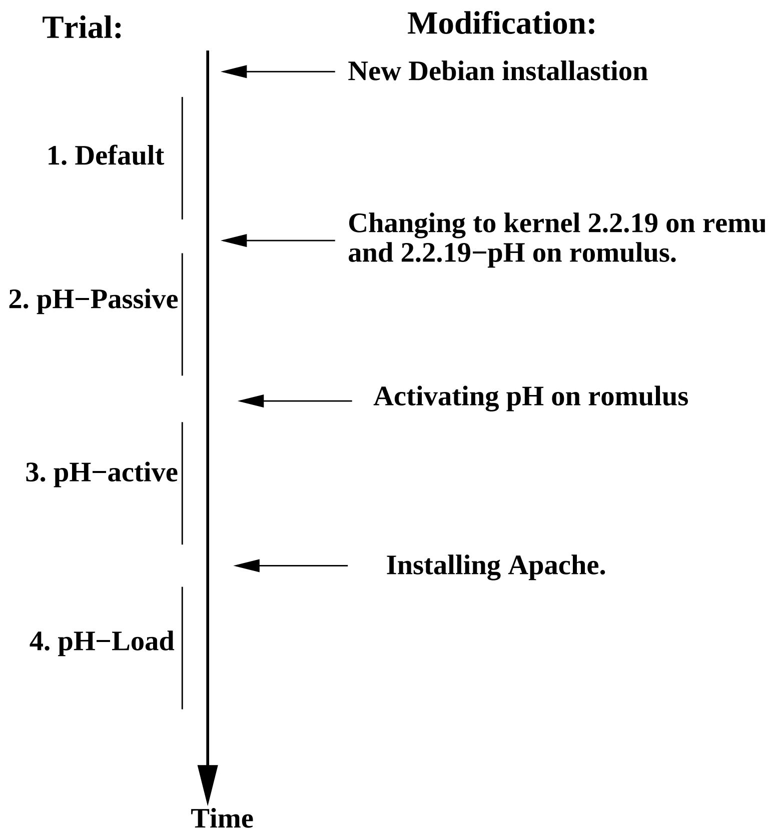


Figure 4.1: This diagram shows the four trials and the modi cations between each trial. For simplicity, each trial is depicted as having the same length.

system behaviour using these variables? Here we have to remember why we conduct these experiments. If we do not nd any signi cant changes using these variables, then other anomaly detection systems using the same variables won't be too affected of pH.

4.1 Data analysis

The data for each experiment were analysed the same way by an application we developed for this purpose. A program written in the programming language Perl takes the two data- les as it's arguments and reads them. The result is a LaTeX document containing all the plots and tables of all the calculations. The plots are generated by `xmgr`, which has an option for batch-mode processing, and our script serves as a front-end for it.

This way we have a standarized way of looking at the data and the calculations both i electronic form and in a printable format using the shell-command `dvips`.

4.1.1 Descriptive data analysis

The data from the two systems is analysed individually before they are being compared with each other.

For each individual measurement of the scalar variable $x = \{x_1, x_2, x_3, ..., x_n\}$, we calculate the following attributes:

- Mean:

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

- Standard deviation:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

4.1.2 Comparative data analysis

For us, the similarity of two systems is de ned by the similarity of their time-series data concerning system-performance and load. We use two types of analysis, inspection and the linear relationship between the two variables.

The correlation (r) measures the direction and strength of the linear relationship between two quantitative variables x and y for n samples. With their respective standard deviation S and mean the formula for r is given as follows (Pearson-Bravai):

$$\textbf{r} = \frac{1}{n-1}\sum_{i=1}^{n}(\frac{x_i - \overline{x}}{S_x})(\frac{y_i - \overline{y}}{S_y})$$

Like the mean and standard deviation, r is strongly affected by a few outlying observations. This poses two problems for our experiments: First comes the rather strong assumption that the data is under a normal distribution. We cannot predict that the variables will have that property and should therefore expect them not to be. Second, we cannot rule out outlying observations. Our experiment is not designed to keep all the noise out. Since the two machines are connected to the Internet all the time, we cannot guarantee that all interference will be identical on both hosts.

If our calculation of their relationship should prove to be to weak compared to the level of noise under these circumstances, then it would be unfeasible to try the same under the later tests. We need another way to describe their relationship.

Ranks may also be employed to determine the degree of association between two random variables.

The correlation coef cient of choice is therefore the Spearman's rho coef cient (r). Serving as a descriptive statistic r provides a numerical value for the amount of linear dependence between two random variables. However, using a standard correlation coef cient, like Pearson's product moment correlation coef cient, the sampling distribution for r holds only under the assumption that the joint distribution for X and Y is normal.

Since the data given to us is assumed to be non-normally distributed, Spearman's rho solves this problem. Rank correlation methods surmount the limitation of a normal distribution and demonstrate a stronger attribute of measuring certain relationships that are not linear. The word "rank" means the position of a sample relative to the value of the other samples. The lowest rank (1) is given to the lowest value and the second lowest rank (2) to the second lowest value and so on. The samples 2,10,7 would be ranked 1,3,2. The Rank Correlation test is a distribution free test that determines whether there is a monotonic relation between two variables (x , y). A monotonic relation exists when any increase in one variable is invariably associated with either an increase or a decrease in the other variable. The monotonic relation is expressed using rank-order numbers instead of the values. This also makes the Rank Correlation a test distribution free test.

The two independent variables are converted into rank-values among themselves, meaning that each value will be interchanged with it's corresponding rank value. Let's illustrate this with an example.

Given two variables:
X = 3, 4, 7, 5, 4
Y = 4, 5, 8, 6, 5

The rank-values alway start from 1. Sorting the values from X, we get the following chain: 3, 4, 4, 5, 7. Since we have two identical values at position 2 and 3, they have to get the same rank value. Their value will correspond to the mean of their occupied positions: 2 + 3 / 2 = 2.5 . The other values will get the value corresponding to their position in the sorted chain:

R_x = 1, 2.5, 5, 4, 2.5

Using the same method on Y, we actually get the same results:

R_y = 1, 2.5, 5, 4, 2.5

Using the two transformed variables, we can apply the formula:

$$r_{Sp} = \frac{\sum_{i=1}^{n}(R_x - \frac{n+1}{2})(R_y - \frac{n+1}{2})}{\frac{n(n^2-1)}{12}}$$

The rank correlation shares the properties of r that $-1 \leq r_{Sp} \leq 1$ and that values near +1 indicate a tendency for the larger values of X to be paired with the larger values of Y. However the rank correlation is more meaningful, because its interpretation does not require the relationship to be linear.

Two variables, describing time-series measurements, will now have the opportunity to get a higher correlation even if they have completely different values as long as the “trace” is similar i.e the peaks coming at the same time but with different value. For our experiments this is an important point.

Further, the transformation of the values to rank-values smoothes the data with regard to outlying values. Thus we have a way to measure the relationship between the two variables in spite of the problems mentioned before. [33, 34, 35, 36]

4.2 External conditions

4.2.1 Machines

Both machines had the same Hardware:

```
CPU: Pentium 133MHz
RAM: 32MB SDRAM
HDD: WDC AC2850F 850MB

PCI-bridge (from lspci):
00:00.0 Host bridge:
Intel Corporation 430HX - 82439HX TXC [Triton II] (rev 01)
00:07.0 ISA bridge:
Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II] (rev 01)
00:07.1 IDE interface:
Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]
00:0e.0 VGA compatible controller:
ATI Technologies Inc 215CT [Mach64 CT] (rev 09)
00:0f.0 Ethernet controller:
```

```
3Com Corporation 3c905 100BaseTX [Boomerang]

Debian version:
Debian GNU/Linux 2.2 r3 _Potato_
- Official i386 Binary-1 (20010427)
```

4.2.2 Network

The two machines are connected to The Oslo University College computer network. They share a hub, which connects them to the rest of the network. This network consists of between 50 and 100 hosts running different Windows avours (95/98/Me/2000/XP) as well as Linux and Sun Solaris. In addition to the occasional noise from the rest of the Internet, there is also an amount of packets coming from the local network. This is mostly broadcast messages from routers and other hosts. We choose not to hide the machines behind a rewall, since broadcast traf c will affect both hosts the same way.

4.3 The variables

A large number of variables could be used to characterise the state of a host. In this experiment, we are narrowing it down to the information given to us by the GNU/Linux program called `vmstat` because we are mainly interested in the comparison of operating system behaviour, with respect to resource handling. This program prints 16 variables, all describing overall system performance. Among these are: CPU load, memory usage, number of interrupts and disc activity. Table 4.3 on the next page shows a list of the available variables and their meaning.

Using this program to print periodically and wait a given interval on both machines, produces a dataset for each system. The data we are given here constitutes a snapshot of the system state. The data is also discrete and all samples have positive values.

We expect some uncertainty in the timing of the two machines. We suspect both clock-drift and `vmstat` not waiting exactly 30 seconds. The clocks were synchronised before each trial. We did no measurement of clock-drift beforehand.

4.4 Description of analysis methodology

We measure the systems in four different trials, where we alter the systems kernel con guration between each trial. During each trial, we measure both systems and analyse the data using the statistical methods described above.

Variable	Description
r	The number of processes waiting for run time.
b	The number of processes in uninterruptable sleep.
w	The number of processes swapped out but otherwise runnable. This field is calculated, but Linux never desperation swaps.
swpd	The amount of virtual memory used (kB).
free	The amount of idle memory (kB).
buff	The amount of memory used as buffers (kB).
cache	The amount of memory used as cache (kB).
si	Amount of memory swapped in from disk (kB/s).
so	Amount of memory swapped to disk (kB/s).
bi	Blocks sent to a block device (blocks/s).
bo	Blocks received from a block device (blocks/s).
in	The number of interrupts per second, including the clock.
cs	The number of context switches per second.
us	user time
sy	system time
id	idle time

Table 4.3: The available variables in `vmstat` (from the `man`-page).

4.4.1 Trial layout

Each trial on a machine is started remotely by running a local shell-script, that generates a data- le and starts the logging to that le. All remote access is done via ssh. It waits for the vmstat-command to nish (it has to be killed explicitly), and then copies the nished data- le to another machine for backup and analysis. A started trial could also be nished remotely, by running a script on both machines that ended the logging process.

Each output le consists of a header containing the output from the following commands: `uname -a`**,** `uptime` **and** `date`**. It also contains two lines, describing the interval which vmstat is going to use to log, and a string describing the current con guration.**

When the vmstat-process gets killed, the script writes the output of a new date-command to the bottom of the le. It then renames it, so that the le-name consists of the following: machine-name, starting date, ending date. The le is then copied with `scp` **to another machine for analysis.**

The script itself is started with `screen`**, a utility for running processes in the background. When started, screen can detach from the terminal, and waits for the process to nish. The use of this utility improved the stability of the trials.**

The main reason for starting the trials with a script, was the need for standardisation and additional information about the system state before each trial. We also wanted a possibility to start and stop the trials remotely from other machines. In addition, we saw the need for doing similar measurements on other machines in exactly the same way and ending up with les in the same format. This also eased the later analysis and the use of automated solutions that parsed the les and generated both plots and statistics.

The format of the log- les and the use of a start-up script developed during the rst trials. First we only had a time-stamp in the data- le, so it was impossible to other than guess the uptime of the machine. After a few trials the number of data- les grew, and so did the need for a accurate naming-convention that both covered the time span of the trial and on which machine this data le was generated. Thus, with a script, we can both automate the starting trials, and the later analysis of the data-sets.

Example data le header:

```
Linux romulus 2.2.19-ph #1 SMP Thu Nov 29 15:54:07 CET 2001 i586 unknown
configuration: apache
interval: 30
uptime:  12:19pm  up 91 days, 20:15,  2 users,  load average: 0.00, 0.00, 0.00
Fri_May_10_12-19-44_CEST_2002
```

```
   procs                      memory    swap          io     system         cpu
 r  b  w   swpd   free   buff  cache  si  so    bi    bo   in    cs  us  sy  id
 3  0  0     16    820  14040   5868   0   0     0     0    0     4   0   0   2
 1  0  0     16   2380  14448   5620   0   0     1    33  378   204   1   1  98
 1  0  0     16   2380  14448   5620   0   0     0     0  105   202   0   0 100
```

4.4.2 Data analysis

The results where analysed with regard on how well the two machines correlated with each other. For each variable we calculated a correlation coef- cient between the systems. For that we used Spearman's rho coef cient which does not make any assumptions on the distribution of the dataset.

See the appendixes for the detailed results from the four trials.

4.4.3 Developed applications

A few programs where written in this project. For reasons concerning space, they where omitted from this text. They can be downloaded from this address:

`www.iu.hio.no/~kyrre`

4.5 Calibration of data

When it comes to the expected data, we anticipate the amount of memory used to reach maximum pretty fast since it's only 32MB. On a more modern machine, the data could have wider ranges in terms of memory usage. Other variables could be attened because of a fast processor and memory throughput. There is certainly a trade-off between a new or old machine. On an old machine, every little task is more apparent in the data. We will only focus on our two old machines, but only because we don't have the time for setting up and repeating the tests on other machines.

Chapter 5

Phase one: System Similarity

5.1 Introduction

In this chapter we will present the results of our rst experiments. This part of the project is what we described as “phase one” in the problem description.

1. How does a system with pH running differ from a pH-free system on an overall system perspective? (control experiment)
   (a) How can we nd a basis on which to compare two apparently similar computer systems with regard to system variables and low uncertainty in measurements?
   (b) What can we say from this about how much any system differs under different conditions and what is tolerable? (Anomaly detection)
   (c) Would the difference between two computer systems increase if one of the systems ran a pH-patched kernel?
   (d) Can any of the information from the previous problem be used to detect anything useful in the behaviour of computers? If so, what can we use the information for?

In the previous chapter, we said that we wanted to carry out four trials with small modi cations between each of them. Let us now take a look at the results from those trials and their results.

First we present the four trials in more detail and the resulting observations. We then take a look at the statistical analysis of the data and discuss them in short. The question if our method answers our initial problem will be discussed at the end of this report.

5.1.1 The first trial - Default

Samples of around 170,000 points were collected, initially, over several weeks. The two systems where now running a standard Debian GNU / Linux Potato 2.2r3 installation (see appendix for complete list of installed packages) with a 2.2.19-kernel. When the trial started, the systems had already been up for a couple of weeks.

The data- les where then copied and analysed on a different machine. By looking at the plots, we found a periodic behaviour on both machines in the variables concerning memory usage (buff, free, cache). These cycles went over a week and where most likely caused by scheduled jobs. We have no audit to support this assumption, though. Other variables (in and bo) produced much more jagged plots. Some variables (w, swpd, si, so) had a the value 0 throughout the dataset and had therefore a correlation of 1 (max) during this time. The application used for the automatic generation of the plots did not produce any plots for these four variables.

5.1.2 The Second Trial - PH-passive

For the rst time there was a distinguished difference between the two machines. Remus used a 2.2.19 kernel while Romulus used a 2.2.19-ph kernel which contained the pH-0.18 patch. Even though pH was installed on the machine, it was deactivated. (it was deactivated in that sense that it never started during boot. pH is started by a special system-call and which is called by a script in /etc/init.d/. So all the lines from the rc?.d/ catalogues concerning PH where removed.). Another difference to the rst trial, was that both systems had just been rebooted before the trial. This was only noticeable in the plots describing memory usage.

5.1.3 The Third Trial - PH-active

In this trial, we wanted to know if the data deviated more with pH activated then before. Right before the monitoring started, the machines where booted. The links to the startup-scripts concerning pH where recreated prior to the reboot.

5.1.4 The Fourth Trial - pH-Load

We installed apache after the third trial. Two packages were installed, `apache` and `apache-common`, as shown by the listing of this output:

```
ii  apache         1.3.9-14       Versatile, high-performance HTTP server
ii  apache-common  1.3.9-14       Support files for all Apache webservers
```

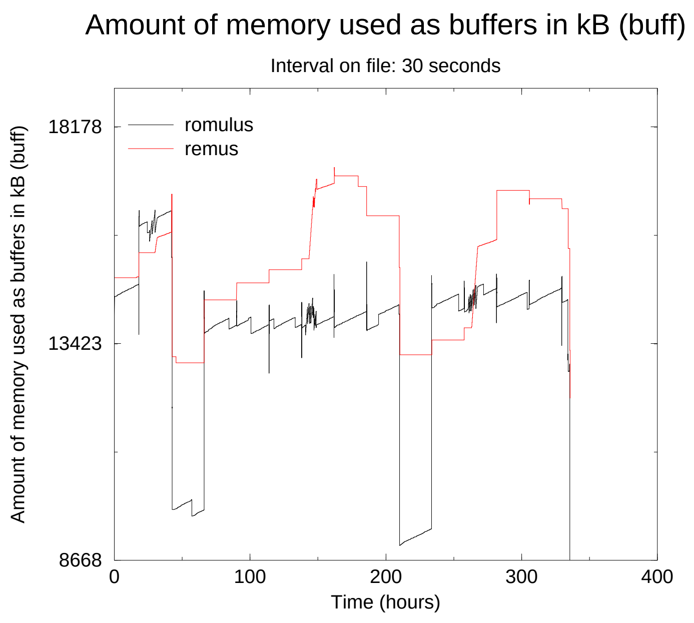


Figure 5.1: Amount of memory used as buffers (kB). This plot shows a greater spread between the two systems. The dark spots come from when the webserver is under load. `pH-load`**. Correlation:** `0.426`

Note that we stressed the system three times during this trial. The webserver only had the standard GNU / Apache welcome page and no underlying server-side scripts.

The data concerning memory usage was much more spread, and showed even a negative correlation on one variable. Only one of the memory-related variables (`cache`**) had a coef cient higher than** `.5`**. See 5.1, 5.2 on the next page and 5.8 on page 47 for more detail.**

The amount of CPU-time spent in system mode (`sy`**) varied also a bit, but if this is due to more network traf c or simply more kernel-code is unclear. The difference is largest at the number of context switches (**`cs`**) with a difference of** `-0.505`**.**

5.2 Observations

During all trials, the plots seemed to be quite similar. A frequency distribution diagram on the variables also showed, that there is no apparent normal distribution. The plots layout was similar with regard to maximum values, periodicy and mean. On most plots both machines showed similar use of resources and "rhythm".All results are found at the end of this text as an appendix.

Some of the variables (`si, so`**) where constant zero and hence corre-**

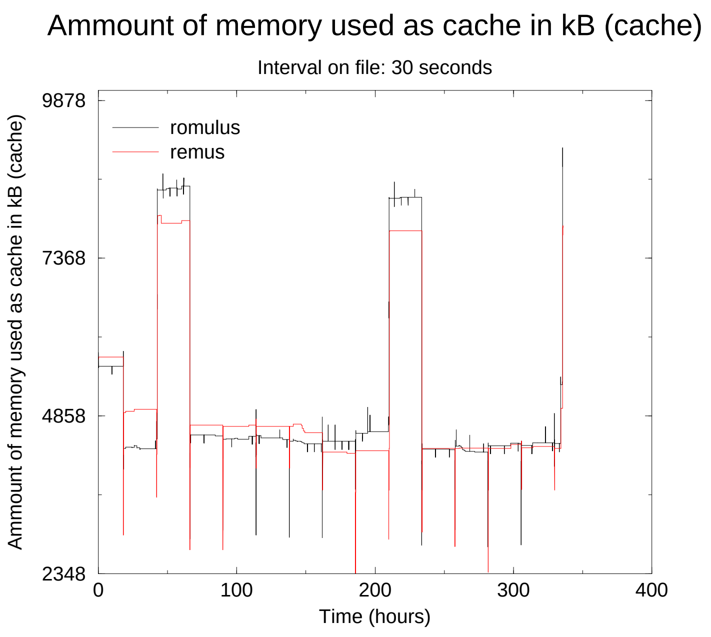


Figure 5.2: Amount of memory used as cache (kB). pH-load. Correlation: 0.527

lated perfectly but could not be plotted.

Both gure 5.3 on the next page and 5.4 on page 43 show the frequency distribution for two variables: cs and free. Note that these plots do not resemble a normal distribution at all.

One of the variables, in, showed an increasingly bad correlation during the four trials. This variable describes interrupts, and is dependent on factors like disk activity. During the third trial, when pH was activated, the pro le of the process had to be read from memory together with the binary itself. This would only happen on romulus, and that would explain the bad correlation in the third trial.

The third trial is the one with the lowest correlation average. For the experiment, the differences between the second and third trial are most important. As the plots and correlation show, the greatest difference is found regarding memory usage (free, buff) and interrupts (in). These variables can be related directly to pH: The memory gets lled up faster on romulus because of the pro le-handling.

Table 5.1 on page 43 shows the calculated linear dependency between the two systems for the rst three trials. Table 5.2 on page 44 shows the third and fourth trial and the difference between each variable.

The gures 5.5 on page 45, 5.6 on page 46, 5.7 on page 46 and 5.8 on page 47 show the memory usage over time during all four experiments. Note how the linear dependency of free drops from the rst experiment (0.931) to the last (-0.147). This shows us that memory usage is highly

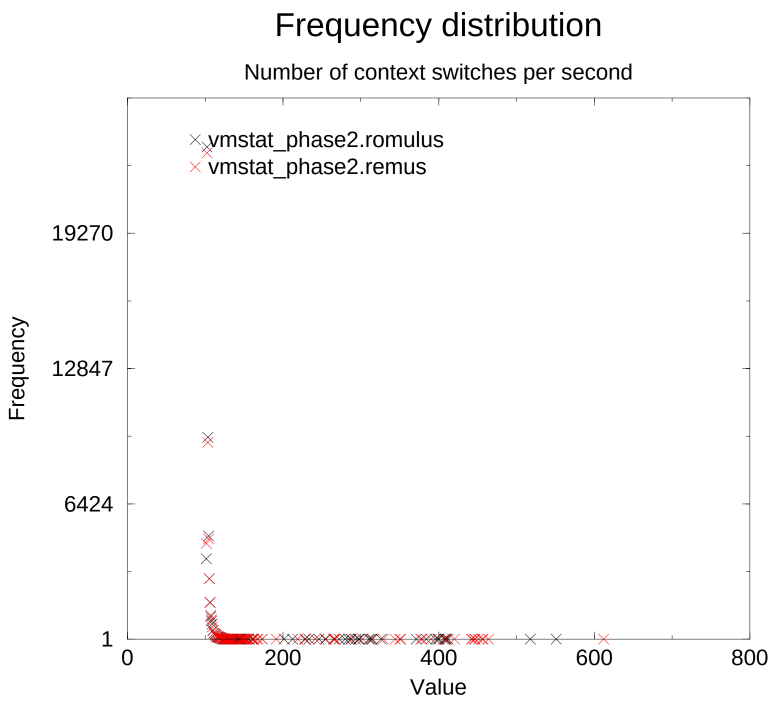


Figure 5.3: Frequency plot. The number of times a value appears in a variable. If this was a normal distribution, we would have seen a curve resembling a bell. We would not expect to see a normal distribution in a non-redundant process. Con guration: `pH-Passive`**. Variable:** `cs`

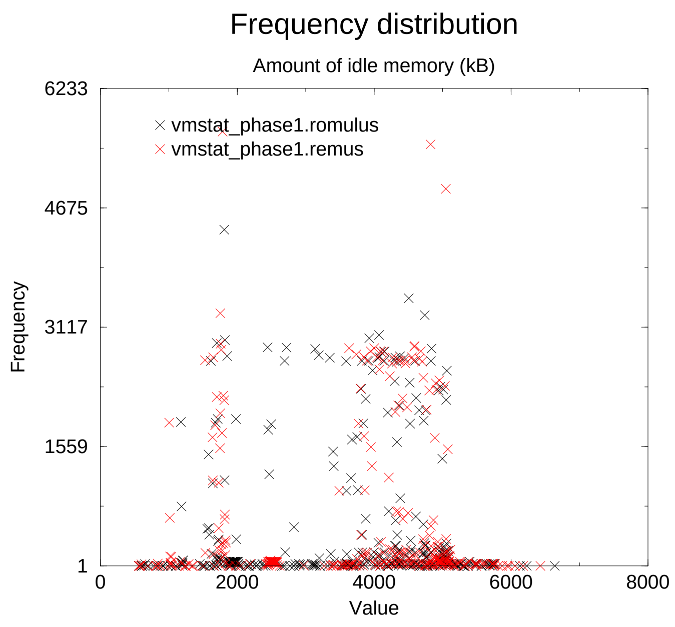


Figure 5.4: Frequency plot. The number of times a value appears in a variable. If this was a normal distribution, we would have seen a curve resembling a bell - Con guration: `Default`**. Variable:** `free`

Variable	Default	pH-Passive	pH-Active	pH-Load
r	0.999	0.998	0.998	0.550
b	1.000	1.000	1.000	0.816
w	1.000	1.000	1.000	1.000
swpd	1.000	1.000	1.000	1.000
free	0.931	0.808	0.323	-0.147
buff	0.462	0.844	0.410	0.426
cache	0.899	0.722	0.889	0.527
si	1.000	1.000	1.000	1.000
so	1.000	1.000	1.000	1.000
bi	0.997	0.997	0.996	0.995
bo	0.949	0.948	0.947	0.812
in	0.735	0.673	0.435	0.424
cs	0.996	0.995	0.971	0.466
us	0.997	0.996	0.996	0.921
sy	0.998	0.998	0.997	0.846
id	0.997	0.996	0.996	0.833
Average:	**0.935**	**0.936**	**0.872**	**0.719**

Table 5.1: Spearman's rho r_{Sp} for the four trials. This table shows the linear dependence between the two machines for all four trials.

variable	pH-active	pH-Load	Difference
r	0.998	0.550	-0.448
b	1.000	0.816	-0.184
w	1.000	1.000	0
swpd	1.000	1.000	0
free	0.323	-0.147	-0.470
buff	0.410	0.426	0.016
cache	0.889	0.527	-0.362
si	1.000	1.000	0
so	1.000	1.000	0
bi	0.996	0.995	-0.001
bo	0.947	0.812	-0.135
in	0.435	0.424	-0.011
cs	0.971	0.466	-0.505
us	0.996	0.921	-0.075
sy	0.997	0.846	-0.151
id	0.996	0.833	-0.163

Table 5.2: Spearman's rho r_{Sp} for the third and fourth trial

affected by the changes we introduced.

Although many plots seemed to be chaotic and noisy, they proved to have a high correlation nonetheless. As we can see from gure 5.9, the plot is full of sharp peaks from both machines, but the peaks appear at the same interval. The difference in the peak value would normally throw off our rst correlation analysis, returning a low correlation coef cient (no linear dependence), while our second calculation, the Spearman's rho, attened those peaks and the result was a measurement of `0.997` (high correlation). Figure 5.10 on page 49, 5.11 on page 49, and 5.12 on page 50 show how this dependency between those two variables continues to exist through all experiments. The correlation decreases a bit in the last experiment. As a comparison, the standard correlation coef cient for this variable was `0.000` for all four experiments.

In gure 5.13 on page 51, and 5.12 on page 50, we clearly see the effect of the stress-tests. Each test lasted for about one hour and caused the plots to have peaks or scrambles in during the tests. What is worth noting, is that the these marks seem similar on both systems.

During the fourth trial we also monitored the pro le for apache on the machine running pH. Taking a look at this dataset we found that the variable describing system call count for the apache-process clearly showed when and how much this process was under stress. Figure 5.14, and 5.15 show this in detail.

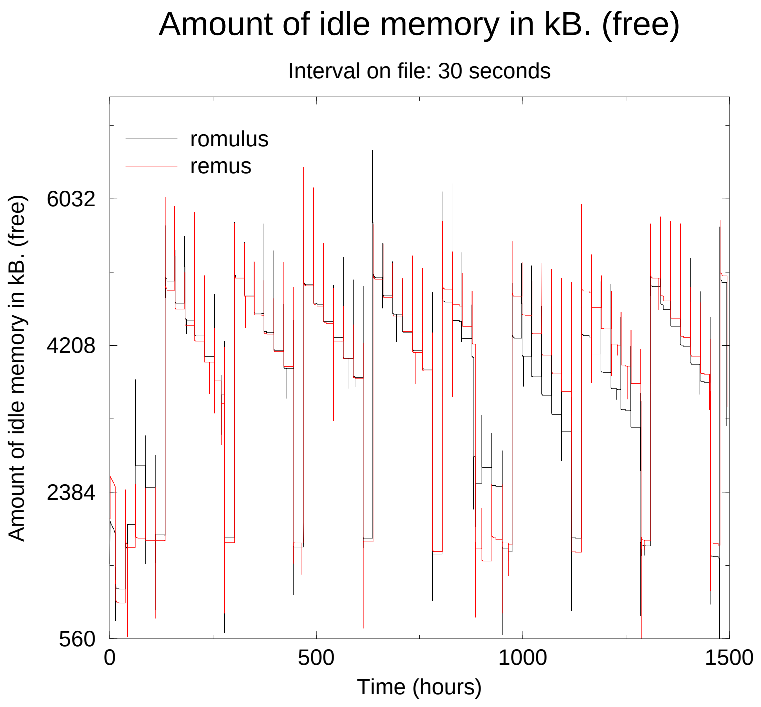
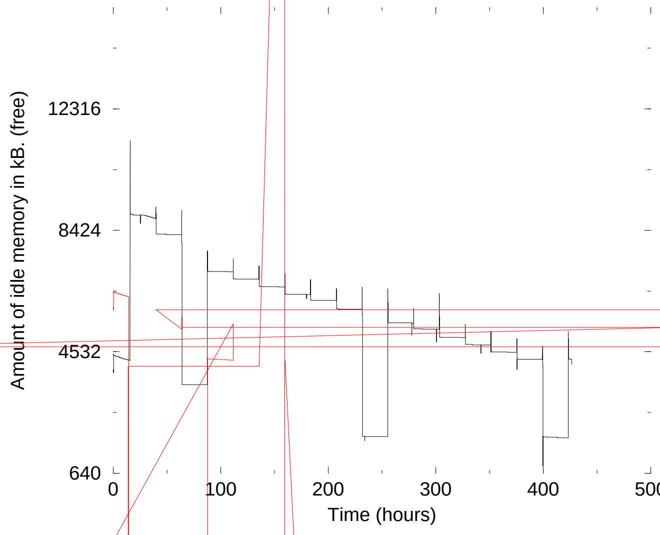


Figure 5.5: Free Memory during the rst trial. The pattern shows the cyclic behaviour of the systems through each week. This pattern comes from scheduled jobs in the system. Con guration: `Default`**. Correlation:** `0.931`

Amount of idle memory in kB. (free)

Interval on file: 30 seconds



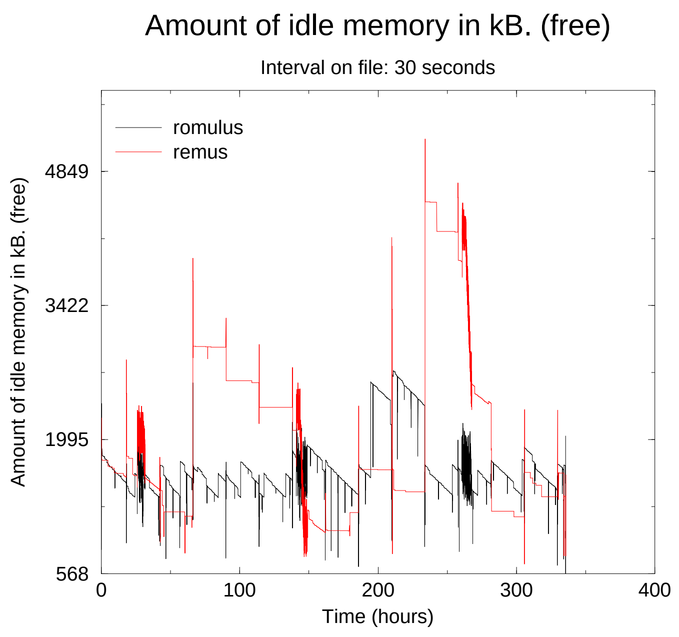


Figure 5.8: Free Memory during the fourth trial. The difference is now even greater then in the previous trials. The dark spots are from when the apache webserver was under high load. Con guration: `pH-Load`. Correlation: `-0.147`

It is the aim of this projects next phase to examine how this information can be bene cial for other tools, like cfengine.

5.3 Discussion

One has to keep in mind, that the only tasks done by the system are OS-speci c processes and scheduled tasks. We still have to compare these plots to a situation where the systems run more services.

Another point is how much deviation between the two systems is unavoidable? We can see from our four trials that the difference increases for each trial. But many variables are affected by the weekly cycle of scheduled tasks. These variables show a different type of similarity other than just e.g same amount of memory used. What shows, is the coincidence of peaks and trends. If they weren't synchronised, then the correlation would be harder to detect since the peaks and curves would not coincide. Two systems can react similar but it would only make sense to compare it in a plot if they would do it at exactly the same time. This argues for a better data-representation. One approach is to generate a weekly pro le, built from all proceeding weeks by adding a new week to the average. A pro le like this from each system can then be compared off-line. Cfengine uses a similar method for building a pro le on resource usage, but uses it for local

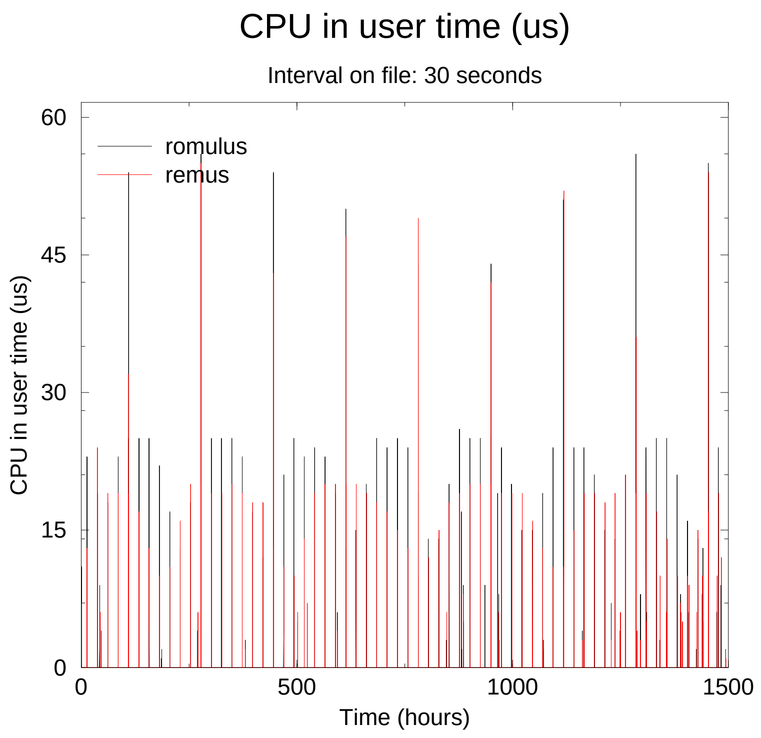


Figure 5.9: **CPU user-time during the rst trial. The plot indicates a vague pattern of peaks, which seem to t with the weekly cycle found in other variables. This comes from the execution of scheduled jobs. Con guration:** `Default`**. Correlation:** `0.997`

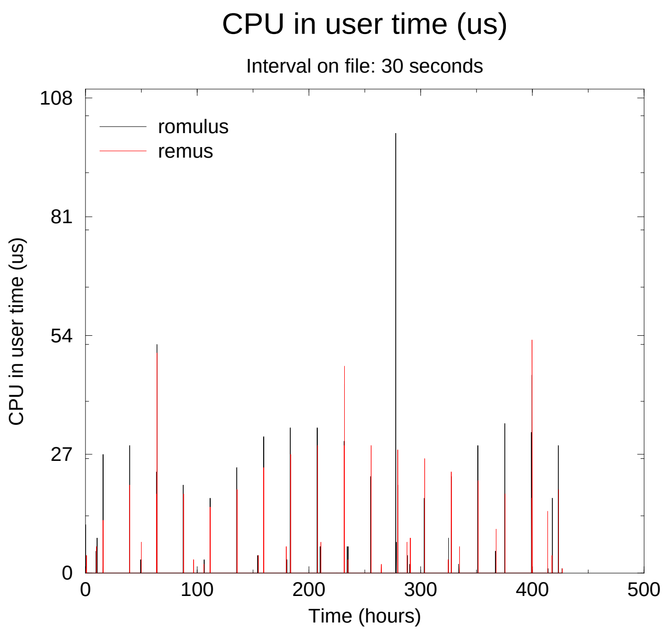


Figure 5.10: CPU user-time during the second trial. Con guration: `pH-passive`**. Also here we can recognise the same waves as in the previous plots. Correlation:** `0.996`

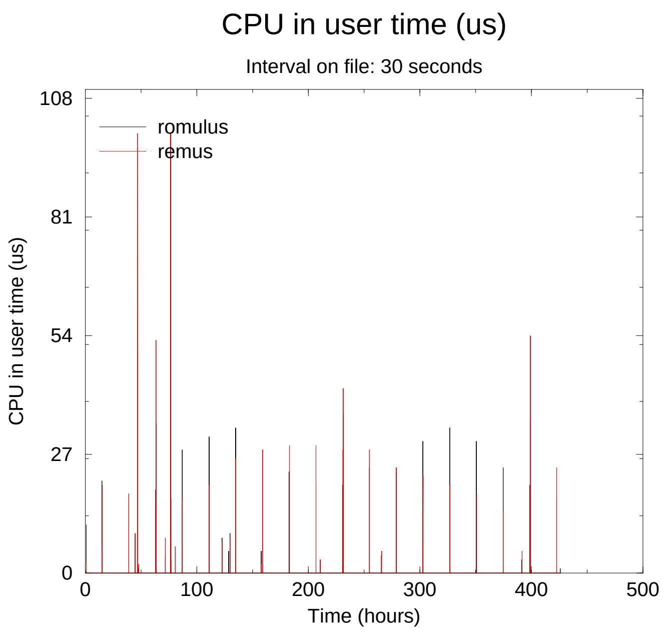
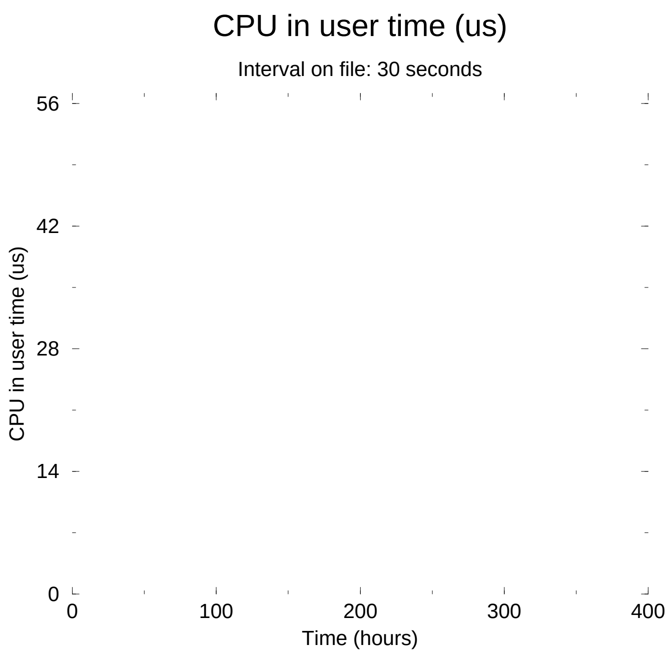


Figure 5.11: CPU user-time during the third trial. The waves are still recognisable. - Con guration: `pH-active`**. Correlation:** `0.996`


CPU in user time (us)
Interval on file: 30 seconds
56
42
28
14
0
0
100
200
300
400
Time (hours)
CPU in user time (us)

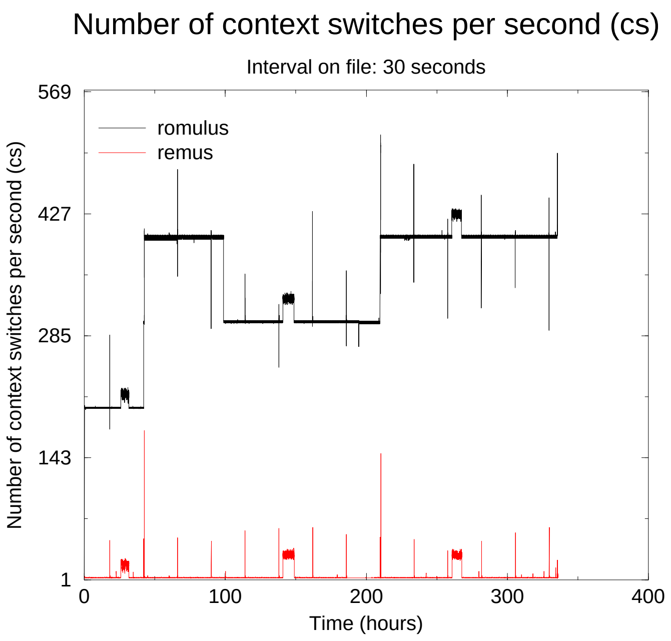


Figure 5.13: Number of context switches per second during the fourth trial. Here we see a clear difference between the two systems. The three small bursts indicate when the scripts downloaded pages from the two systems. The peaks coincide also here. Con guration: `phase-apache`**. Correlation:** `0.921`

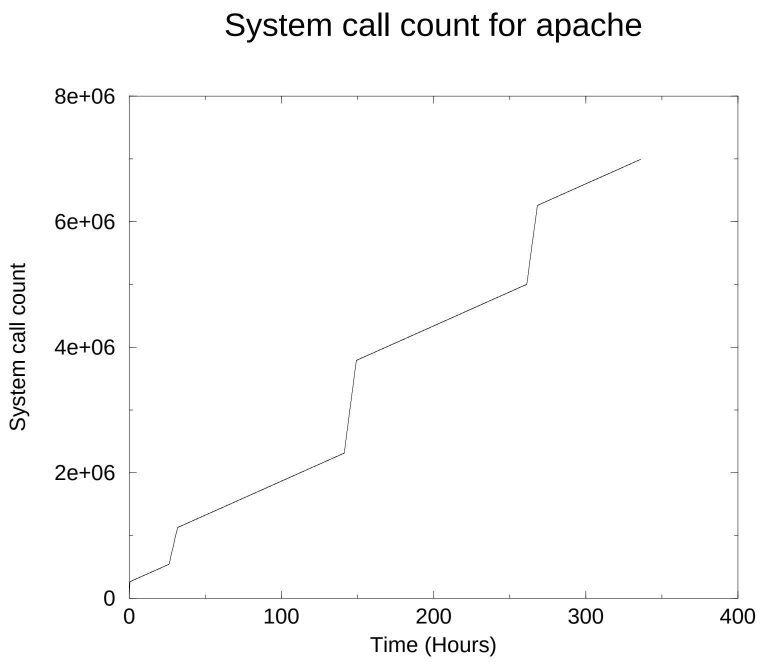


Figure 5.14: The system call count for the apache processes. We see that the count rises more rapidly three times and falls back to a steady linear rise. These rises coincide with the times we stress apache, meaning that the apache processes execute more system calls during the stress-periods.

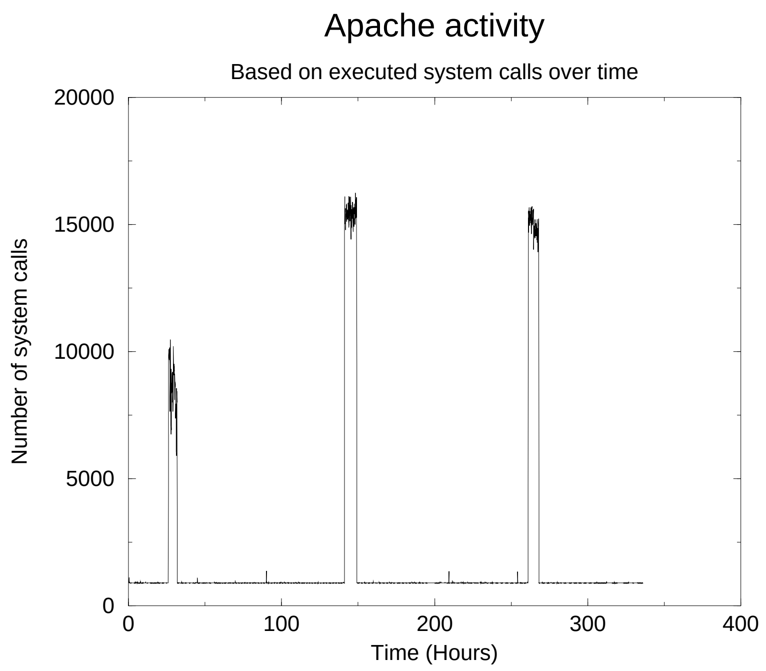


Figure 5.15: **This plot shows how the amount of executed system calls for the apache processes rises signi cantly every time we stress the webserver. Can this information be useful in a time-series context?**

already give a jagged plot (i.e context switches and variables concerning CPU usage).

By smoothing the data-sets to longer time-intervals consisting of the average of the measurements in that interval, we could make up for some of the time-deviation problems. Should we use a classical correlation method (i.e Pearson-Bravai), then the correlation coef cient would be in uenced by this smoothing (see 5.3 on the following page), but Spearman's Rho does, interestingly enough, not seem to be much in uenced by it. The reason lies in the transformation of the data to rank values. There we have already taken care of most spikes and bursts in the plot. Table 5.4 on page 55, show the correlation coef cients for the smoothed data compared to the original. We averaged for 5, 10, 30 and 60 minutes.

The most jagged plots look unusable at this time-interval. Local averaging make a few variables appear more attractive for further study, but that will probably show better when the system is more active.

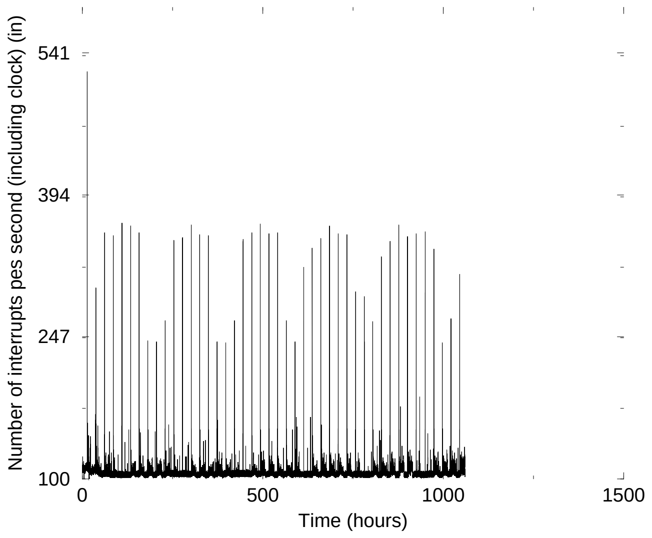
Figure 5.16 on page 55 shows an example of what the data looks like if recorded every 30 seconds. In gure 5.16 on page 55 the data are smoothed using a local average over every 30 minutes instead of the original data. Note how the ow in system usage becomes more visible and also how the different correlation coef cients react to the transformation of the data.

variable	Original	Smoothed
r	-0.0002899	0.2975995
b	-0.0000122	-0.0007340
w	0.0000000	0.0000000
swpd	0.0000000	0.0000000
free	0.9488736	0.9558609
buff	0.7191931	0.7199619
cache	0.8523288	0.8594925
si	0.0000000	0.0000000
so	0.0000000	0.0000000
bi	0.0072123	0.6123918
bo	0.0016221	0.5233057
in	0.1640182	0.9164983
cs	0.0056725	0.4691590
us	0.0024800	0.4940068
sy	0.0018850	0.5262725
id	0.0000000	0.0000000

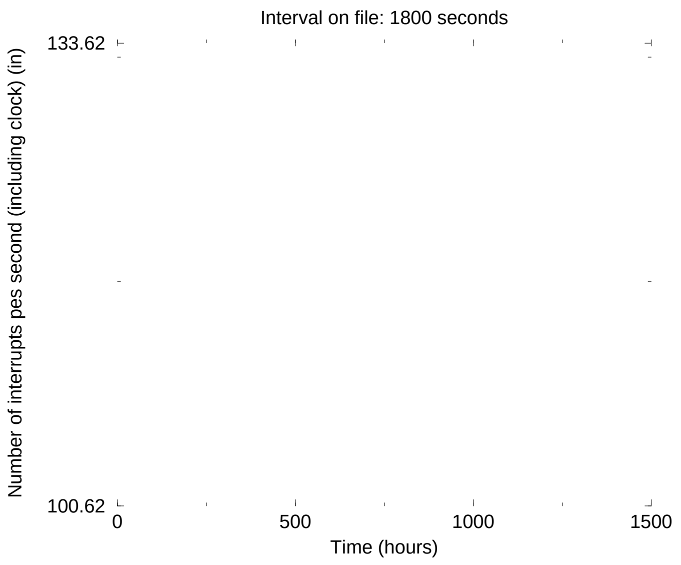
Table 5.3: Comparison of Pearson-Bravai correlation coef cient for the rst trial (Default) using original (30 seconds) and smoothed (30 minutes local average) data. Note how the coef cient changes drastically in some variables. This argues that this type of calculated linear dependency is much more sensitive to time-intervals than Spearman's rho r_{Sp} in 5.4 on the next page.

Number of interrupts pes second (including clock) (in)

Interval on file: 30 seconds



Number of interrupts pes second (including clock) (in)



5.3.2 How can these results help to define system normality and state?

How could a system define normality based on these results?

A system could generate a dataset for a selected subset of variables for each week. It could then merge this set into an existing average of all sets generated so far. It could generate a correlation coefficient for this week against the average week. By storing both the average dataset and each correlation coefficient, it could determine how normal the past week was in terms of how well it correlated with the average. A time-series of the correlation coefficients could be stored to determine the long-time effect of certain cases which the system experiences.

By calculating the average correlation we could also look at how similar the weeks actually are, and how much difference from the average one could expect for this particular system. This last value would be very interesting to compare with many different computer systems just to test whether it is possible to even define both a metric and a value for system normality throughout different systems.

The method just described is a so-called best-of-fit method. It uses a statistical method to test whether the new data fits the established profile. There are many different statistical tests, both multi and single-variable, that could be used here. Some are discussed in [13]. Cfengine uses this type of anomaly detection in its framework. We have not done any such tests in these experiments. If the project was primarily on the statistical analysis of this type of data, then a further inspection of our collected results would be the next step.

Another metric would be time and resources used for a specified task. This task could either be one big job, or a series of small jobs, all with different contents. One could then compare the results with an average from all subsequent results. The problem with this, is that this test, if designed to stress the system, would have an impact on the dataset described above.

Important variables in this context are how much CPU and memory the task consumes, and how long it takes to finish. Especially the amount of time the CPU is in kernel-mode. These numbers can vary between each trial on a single system, so several tests regarding each task are necessary. These tests need to be done at every different machine configuration, and the results may vary because of other factors then just pH itself. Kernel-configuration and effectiveness is expected to influence CPU-usage and the time used to execute a given binary.

We have seen, that pH affects system variables, but how visible this effect is under higher load is difficult to say. Our data indicate (see 5.8 on page 47), that higher load will have a greater impact on the data then pH, but this needs to be examined further. Now that we have analysed the effect of pH on system variables, it is time to turn towards our main goal in this project. How can cfengine pro t from pH? Cfengine has a component for analysing time-series data. Can pH offer new insight into the “life” of a computer system? How can we answer this question? This is the goal of our next phase. We will now take a closer look at pH's data and inner workings. One approach is to create a similar logging system for pH's variables over time to see if they are usable for cfengines anomaly-detection component. We also recognise the need for further data-gathering. The next phase will therefore also concentrate on building a better tool for both data-harvesting and analysis.

Chapter 6

Phase Two: A scaled, immunological approach to anomaly countermeasures (Combining pH with cfengine)

6.1 Introduction

This chapter describes the process of combining cfengine, a high level con- guration engine with pH, a kernel patch which enables anomaly detection and reaction on a per process basis. This part of our project has two independent goals:

1. To provide a better anomaly detection capability for cfengine, and a better response engine for pH, thus going a rst step towards a distributed computer immune system.

2. 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 x. We want to give our computer immune system data-gathering capabilities for future research because we believe there is still much testing to do in the eld of system normality.

We will begin this chapter by discussing the requirements for compatibility between pH and cfengine, as well as what we hope to achieve by combining them. In sections 3.3 and 3.4 we provided some details about these two systems. Let us now take a look at their compatibility and how we want them to function together. Second we will look at what modi -cations are necessary in order to implement the wanted functionalities. In the next chapter we will try to actually implement some of the proposed modi cations.

6.2 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 (steady state)[37, 3, 6, 4]. As a short re-cap:

The University of New Mexico's pH kernel modi cation stands for process homeostasis. Its goal is to seek a steady state in the tasks that are undertaken by a computer system. It detects new tasks, as unknown events, 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 con guration of a system, where con guration means the state of the le 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:

- Do they have compatible data models?
- 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 seek reparation?
- Resource utilisation 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 utilised by cfengine in order to bring the knowledge of short term behaviour to bear on longer term strategy.

The idea is sharing of information, or data. Cfengine can pro t from the information made available by pH. If we succeed, we have gained more functionality by combining the two systems then by having them work independently. Let us now detail a model where the systems have the possibility to interact.

6.3 A closer look at cfengines anomaly detection component: cfenvd

In cfengine 2.x, additional classes are automatically evaluated based on the state of the host, in relation to earlier times. This is accomplished by the additional cfenvd daemon, which continually updates a database of system averages, which characterise "normal" behaviour. The state of the system is examined and compared to the database, and the state is classi ed in terms of the current level of activity, as compared to an average of equivalent earlier times. e.g.

```
RootProcs_low_dev2
netbiosssn_in_low_dev2
smtp_out_high_anomalous
www_in_high_dev3
```

The rst of these tells us that the number of root processes is two standard deviations below the average of past behaviour, which might be fortuitous, or might signify a problem, such as a crashed server. The WWW item tells us that the number of incoming connections is three standard deviations above average. The smtp item tells us that outgoing smtp connections are more than three standard deviations above average, perhaps signifying a mail ood. 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.

The current long-term data recorded by the daemon are: number of users, number of root processes, number of non-root processes, percentage disk full for root disk, number of incoming and outgoing sockets for netbiosns, netbiosdgm, netbiosssn, irc, cfengine, nfsd, smtp, www, ftp, ssh and telnet. These data have been studied previously, and their behaviour is relatively well understood.

Cfenvd sets a number of classes in cfengine which describe the current state of the host in relation to its recent history. The classes describe whether a parameter is above or below its average value, and how far from the average the current value is, in units of the standard-deviation (see above). This information could be utilised to arrange for particularly resource-intensive maintenance to be delayed until the expected activity was low.

For more information, see [38].

6.4 A cooperative model

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 traf c 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 rst 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 modi cations to both. This model would be preferred if we wanted a centralised computer immune system where everything goes out from one controlling unit. This approach, however, does not comply with our initial de nition of a computer immune system. By making pH a plug-in for cfengine, we loose the independence between them, making pH only work optimal if cfengine should function. The most important argument against this model, is that pH is in kernel-space. Operating system design focuses on layering of the system to achieve transparency, security and ease of development [26]. Using pH as a plug-in is therefore a weak design.

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 special interface or by parsing log les. This can be viewed as opening communication channels between them. The systems could also interact passively by either leaving hints as in log- les or sending messages directly, or actively by using commands and already de ned user-interfaces. Both passive and active channels can be used between the systems. This approach is more in line with our idea of a computer immune system. Passive communication channels are an advantage, since the receiving part can choose when to read new information and is thereby more independent.

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 g. 6.2), using les 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 analyses 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 pro les are in a self-de ned binary format and will be printed to the terminal by the command `pH-print-profile`. The same holds for the sequence les, with the corresponding command `pH-print-sequences`.

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

In g. 6.1, a number of identical trials was performed in order to simulate a long term variation of the form measured by cfenvd (see g. ??). 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 ve times, a pH process counted the total number of system calls. The average of the ve identical trials with standard deviation shown as error-bars is plotted in the gure. Each trial measured 120 values, recorded each 30 seconds over the space of an hour. This gure is suf cient 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 speci c 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 suf cient to yield exact repeatability. Thus if one is looking for repeat-ably identi able signatures, one must allow a margin for error. This is clearly signi cant for intrusion detection systems, which normally recognise 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 traf c load can be ruled out, since the trials were performed in isolation.

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 re ected linearly, up to a standard error, in the

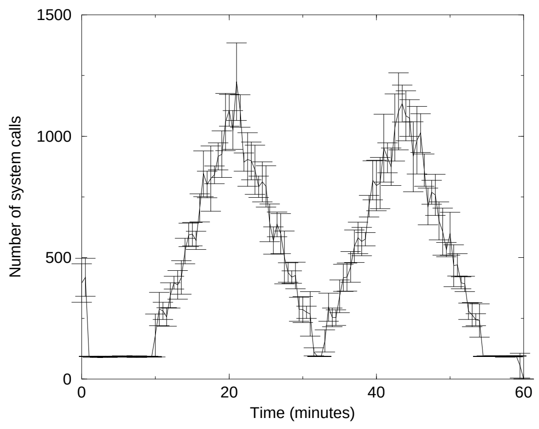


Figure 6.1: Repeated trials on a simulated load, showing how the average number of system calls varies in proportion to applied load, within measured tolerances. The separation of signal and noise shows that the basic cfengine statistical model applies to pH also.

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.

We can conclude, that pH and cfengine have compatible data-models. Data extracted from pH can be used in cfengines statistical approach to system normality de nition. The relationship between system-call frequency and process-load was unintended, but as our experiment shows, it gives a useful and new addition to our data-gathering capabilities and increases the functionality of our computer immune system.

6.4.1 Modifications to pH

Let us now take a closer look at what parts of pH need to be modi ed in order to communicate with cfenvd and why.

The most important modi cation to pH, is having the ability to specify what processes to delay. The monitoring will still be done on all pro les, 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 pro le. If the default value is delay, then pH will work as before. The administration of

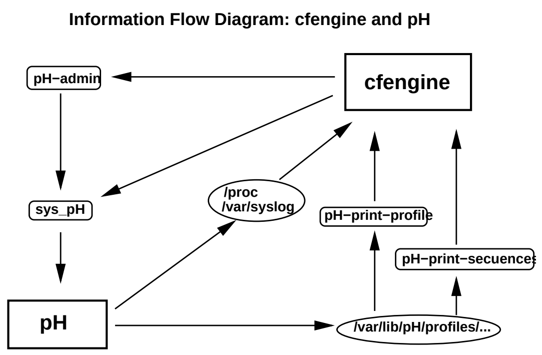


Figure 6.2: Information Flow Diagram: cfengine and pH.

these variables can be handled from cfengine. This enables us to achieve the following:

- Delay all but these binaries (Default on)
- Ignore all but these binaries (Default off)

Note, that the default value could be changed in runtime too.

6.4.2 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 pro les 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 in uenced 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 rst encounter with all applications will in uence the average and deviation so much that small and potentially interesting deviations later on will be unrecognisable.

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 analyse them statistically (see g 6.3).

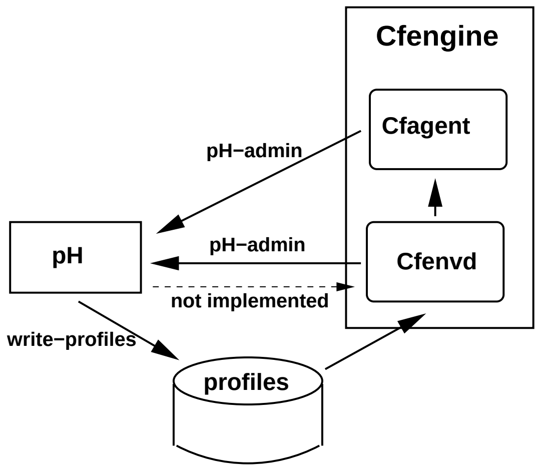


Figure 6.3: Information Flow Diagram: cfengine and pH.

6.5 Example regulation strategy

We envisage automated responses to anomalous behaviour. Such responses have been considered before in other contexts (see refs [39, 40]). A simple example of a cfagent response helps in visualising 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 identi ed, 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.

It not necessary that a `ph_sequence_anomaly`-event has to correspond to a single sequence-pair. We could use an anomalies-count within a frame instead. To differentiate even more, we could de ne different classes for binaries, giving them stronger or weaker limits for accepted amount of anomalies. We have to remember, that pH will have reacted to that process in particular already.

```
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 cfengines lter 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.

6.6 Computer Immune System Functionality

What would a higher level system environment daemon like cfengine want to know from pH? What information is useful? We can see several bene- ts from creating a computer immune system from these two components. Some of them give us better reaction capabilities as in the ability to change system state. Others help gaining a better and more accurate view of the system state, enhancing the learning and adaption capabilities.

One important point, is that the proposed functions of this computer immune system exist only because of the teamwork of these components.

6.6.1 Reaction Capabilities

- Discontinue a service with a high anomaly count

  pH has no functionality for killing a process. Second, the variable for determining how much anomaly is tolerable is set globally. We could have the need to specify this more accurately per pro le, say a high tolerance on unimportant pro les, and a tighter watch on some core services.

- Notify other machines of the behaviour of a process. Interchange or compare pro les with other systems.

  Should a host experience a high anomaly on one process it could warn other machines on the network about it. In addition, different hosts could interchange pro les. This, off course, implies a secure channel and a protocol for the communication. Today, cfengine offers a framework for the communication, and there is research going on to de ne a standard format for intrusion detection (Intrusion Detection Message Exchange Format - IDMEF).

  Additionally, should a host experience a high anomaly on a service, it could “ask” another machine to take over the service. This would be an algorithm for replication management based on anomaly detection instead of common variables such as load.

- Dynamic Invocation of Other Components.

  Cfengine and pH alone are perhaps not able to detect intrusions on all fronts of the system. They should therefore be able to spawn other intrusion detection systems on demand if they are present. With a higher level engine on top, we could gather information from several other monitoring tools to create a substantial data on the system state. This could be time and resource consuming, so we might have to have the opportunity to adjust the monitoring to our needs. Say we suddenly get a high anomaly count from a process running a network service. We could then spawn other network related tools to monitor the network more closely. We could also start the monitoring on other hosts in case this one has to take counter measures, like shutting down the network.

6.6.2 Learning and Adaption Capabilities

- How many processes have anomalies? When a system is using pH for the rst time, most processes will have a high anomaly count. Over time, however, the processes will be used mostly the same way and reliable pro les will be generated. The number of processes with a anomaly will decrease. Should we experience a high number again, while expecting routine behaviour, this could mean two things: (1) The host is using a lot of new software that misses suf cient pro les. (2) The host is running processes that behave in a different manner as usual.

  This number would also be a indicator to the stability and predictability of a host. Logging this variable over time could give an indicator on how the host adapts to it's work. A machine acting as a stand alone web-server, would have a relatively low count with a low deviation. A desktop computer standing in a student computer room will have higher sum with higher deviation, since it probably would be running many different applications and even programs designed by the students. Given a time perspective, we could also build an expectancy regarding the time of day. During the night, we don't expect any new applications to be run. During of ce hours, this would be more likely.

- Analysing the behaviour of a binary over time. A comparison of pro les across different hosts could also indicate how similar the different applications are being used on the different hosts. This has Human-Computer-Interaction rami cations, and is especially interesting for complex programs, such as computer games or of ce 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 i.e use more complex programs at the end of their work-day? The bene t 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.

  From a learning perspective we could use these data to determine how user skill affects the way software is used. We could ask if a pro le generated by a novice user would be different from that of a thought user.

- Does this special process experience any anomalies?

  We could also monitor only one process, say a web-server or DNS service. This would imply that we trust all other applications on the system. Still, it would mean that pH would not affect anything else then this program. As said earlier. On a system where students write programs, we would not want pH to generate a pro le for every new binary and delay the students new program. Therefore a selective approach could be more useful by simply choosing the binaries you would like pH to monitor.

6.7 Conclusions

As we can see, there are a lot of enticing features and enhances to our security model by creating this computer immune system. We have of yet not discussed how feasible and comprehensive the changes have to be in order to make this work properly. We have not showed how we wish to test these functionalities and to show that they contribute to higher system integrity.

Note, that all this comes from only two independent anomaly detection systems. Imagine what a third participant could add to our computer immune system. Would it at all be possible? This discussion will be continued at the end of this document.

We will now turn towards realizing some of these functionalities. Alas, time does not allow us to try to implement them all. We have therefore chosen a some speci c capabilities and discuss how they may be best implemented with regard to existing code in the following chapter.

Chapter 7

Implementing functionalities

This chapter describes the functionality we are searching for in the result of a cooperative, interactive and computer immunology based approach of combining cfengine with pH. We have earlier described the possibilities and bene ts of combining these systems. Let's take a closer look at how this interaction can be implemented. We have chosen three speci c cases where we foresee a possible bene t of information-sharing. At the end of this chapter, we will take a look at an actual implementation of one of these functionalities.

7.1 Introduction

There are two types of anomalies that have to be detected. The rst is dubbed real-time-anomalies which means anomalies that emerge because a process has executed an unknown system-call pair and is being reacted to. The detection of these anomalies is done by pH, and we want cfengine to be able to react to them too.

The second type of anomaly is our time-series based anomaly detection algorithm currently implemented by cfengine. An anomaly here means time-series data that does not comply to an expected value based on already gathered information. In this case, pH delivers new additional types of data, which must be communicated to cfengine in a performance-friendly way.

The three functionalities we want to implement are as follows:

1. Sharing triggers. PH noti es cfengine when reacting to anomalies, giving cfengine a chance to participate.
2. Sharing Pro les. cfengine can analyse process behaviour on it's own based on the data in pH's pro les. How can they be made accessible?

3. Time-series analysis. As shown earlier, there is a clear connection between number of system calls per unit of time and the load of the process. pH has this information, but does not incorporate it into it's own anomaly-model. Cfengine could easily add this data to it's model. How can it access the data?

7.2 Real-time Anomalies

7.2.1 Case 1: Sharing Anomalies

We want cfengine to process the same information that we usually see in syslog regarding detected anomalies. One obstacle is to solve the time-dependency problem between the two components. Cfengine does not run all the time, and must therefore process a batch of messages every time it runs. This batch should not be on disc, because this would force pH to write to disk every time it would write to syslog additionally.

The way to solve this, is to make pH write to a pipe every time it writes to syslog. Every time cfengine runs, it can process all the messages in the pipe according to it's own policies. This is probably the highest real-time we can achieve between the two systems. We cannot predict when cfagent will run or how often, but we can predict when cfenvd will run. But this is not entirely bad. It gives us a comfortable independence between the two systems, and besides, pH has already functionality for reacting to these types of anomalies.

All these messages in the pipe should have a speci c format. The simples way of doing this, would be to write the same information to the pipe as to syslog. Cfengine could then parse the information.

Implementation

pH's method to get messages to syslog is to call the method `KERN_DEBUG` which is a pointer to a method that sends a message to the syslog-daemon (is this true?). Every time there is some message to be delivered, the method `action(''message'')` is used. For the sake of transparency in the code, we have several options:

1. We make a new method, i.e `pH_push_message_2pipe(char *string)` which writes the message to a pipe. We could then call this method prior to or after each `action()` call where we nd it appropriate.

2. Another approach would be to rede ne the `action()` method to write both to syslog and to pipe. This would give nicer code, but instead we could ll up the pipe with uninteresting information.

Suggestion 1 is a bit more elegant, because we get the possibility to chose the type of information we wish to communicate more selectively.

```
#define action(format, arg...) \
{ \
        if (pH_loglevel >= PH_LOG_ACTION) { \
                printk(KERN_DEBUG "pH: " format "\n" , ## arg); \
        } \
}
```

7.3 Time-series Anomalies

7.3.1 Case 2: Sharing Profiles

The information we want cfengine to process can be found in the pro les of some previously selected applications. The pro les are read from disc when the application starts, but are not written to disc until the last instance of the process exits. We want to read updated data from a selected application at any time, regardless if the current pro le is in memory or stored on disc.

Currently we can only ask pH to write all pro les to disk, not print one selected pro le. We need functionality for cfengine to get a speci c pro le directly from pH or at least not through disc rst. Secondly it would be quite expensive if pH should write down all pro les if we are only interested in, say, 10% of them.

The data given from pH is in a raw format and has to be parsed before it is stored in the database. This parsing can be done by cfengine. All pH needs to know is to write this speci c pro le to this le (which could be a pipe).

If cfengine has a list of processes it wants to monitor, it can then get the pro le for each of these using a loop that iterates the list.

Implementation

Currently, pH has a function called `void pH_write_all_profiles(void)` :

```
void pH_write_all_profiles(void)
{
        pH_profile *profile;
        state("Writing all profiles to disk.");

        down(&pH_profile_list_sem);

        profile = pH_profile_list;
        while (profile != NULL) {
                pH_write_profile(profile);
                profile = profile->next;
        }
        up(&pH_profile_list_sem);
}
```

It iterates a list of pro les currently in memory and calls the method `pH_write_profile(profile)` for each of them. This method is called when the system-call `sys_ph` gets the parameter `PH_WRITE_PROFILES`, which stands for the number 16. The system call itself is done through three lines of x86 assembly code.

Since the pro les are held in a list, a request to write a single one would force us to iterate the list anyway and test for the right pro le. The identi- er could be the absolute path of the binary we want to monitor. We then have to write a new method, i.e `pH_write_profile_2pipe(profile)` which would resemble `pH_write_profile(profile)` in most ways except that the information is written to a pipe instead of to a le. This does not have to be a pipe. The method could just update the single pro les le on disc instead.

A drawback to the pipe-approach, is that the two systems get a bit too closely woven. What if we have to restart cfengine? What if other programs would be interested in this information as well? Should pH not write to the pipe, if cfengine is not installed on the system? Besides, a pipe could ll up if someone constantly asked pH to write to it.

7.3.2 Case 3: Time-series analysis

PH has already a good deal information in the `/proc` directory. It has a folder for global information and a status- le in each process-folder. This status le contains the information about each process, but we are mainly interested in information about the pro le of the binary. A typical status- le looks like this:

```
romulus:~# cat /proc/161/pH
normal        : 0
frozen        : 0
delay         : 0
count         : 8023
LFC           : 0
maxLFC        : 3
profile : /var/lib/pH/profiles/usr/sbin/cron
```

There are different ways to solve this. From cfengine's point of view, this means that to get access to pro le-related information, it rst has to nd a process of that binary, and then read the le in the `/proc` directory. This is possible, because all processes of a given binary share the same pro- le.

The bene ts of this approach would be true independence between the two systems in this regard. To make the information available in this way, means that other systems could read the data too. In addition, no communication channel has to be initialised.

Another advantage for this alternative, is that it is performance-friendly. Files in th `/proc` directory are not stored on disc. When reading such a le, one is actually causing the kernel to run a method for that le, which returns the contents. Thus we will never have to read from disc, since this information won't be swapped either.

Implementation

The modi cation of the code is rather simple. This is the method that is responsible for returning what any user would read from a pH le located in a process-folder:

```
/* called by get_process_array() in fs/proc/array.c */
int get_pH_taskinfo(int pid, char *buffer)
{
        struct task_struct *tsk;
        int len, fn_len;

read_lock(&tasklist_lock);
tsk = find_task_by_pid(pid);
        /* FIXME!! This should be done after the last use */
read_unlock(&tasklist_lock);

        if (!tsk)
                return 0;

        if (tsk->pH_state.profile) {
                len = sprintf(buffer,
                              "normal  : %d\n"
                              "frozen  : %d\n"
                              "delay   : %d\n"
                              "count   : %lu\n"
                              "LFC     : %d\n"
                              "maxLFC  : %d\n"
                              "profile : ",
                              (tsk->pH_state.profile)->normal,
                              (tsk->pH_state.profile)->frozen,
                              tsk->pH_state.delay,
                              tsk->pH_state.count,
                              tsk->pH_state.alf.total,
                              tsk->pH_state.alf.max);
                fn_len = strlen((tsk->pH_state.profile)->filename);
                if (fn_len > (PAGE_SIZE - len - 2))
                        fn_len = PAGE_SIZE - len - 2;
                strncpy(buffer + len, (tsk->pH_state.profile)->filename,
                        fn_len);
                len += fn_len;
                buffer[len] = '\n';
                buffer[len + 1] = '\0';
                len += 2;
                return len;
        } else {
                return sprintf(buffer, "No profile.\n");
        }
}
```

As we can see from the code, the data comes from a `task_struct` and this struct has a pointer to the pro le. Currently only one variable from the pro le is written, the name of the pro le. This could be extended to include more variables, like the global system-call count of that binary.

In the next section, we will try this alternative and implement the described changes into the kernel.

7.4 Implementing Case 3

One of the data we want cfenvd to collect, is the global system-call count for that variable. Currently, we have to ask pH to write to disk, and then read the information from a le. This approach is cumbersome and is not suited for long term data-collection. The preferred solution is to make this data available without going via the disc. To implement the changes described in alternative 2, is very simple. We just expand the `sprintf` call to include more variables. The solution would then look like this:

```
len = sprintf(buffer,
              "normal  : %d\n"
              "frozen  : %d\n"
              "delay   : %d\n"
              "count   : %lu\n"
              "LFC     : %d\n"
              "maxLFC  : %d\n"
              "profile-count   : %lu\n"
              "profile : ",
              (tsk->pH_state.profile)->normal,
              (tsk->pH_state.profile)->frozen,
              tsk->pH_state.delay,
              tsk->pH_state.count,
              tsk->pH_state.alf.total,
              tsk->pH_state.alf.max,
              (tsk->pH_state.profile)->count);
```

Note that the printed string now includes the `count`-variable belonging to the pro le (`(tsk->pH_state.profile)->count`).

We did this modi cation and made a debian-package of the new kernel, which we installed on one of the test machines. The result was this:

```
romulus:~# cat /proc/161/pH
normal        : 0
frozen        : 0
delay         : 0
count         : 10483
LFC           : 0
maxLFC        : 3
profile-count : 3669385
profile : /var/lib/pH/profiles/usr/sbin/cron
```

As we can see, the pro le-count is added to the output. Now, the variable is easily accessible for all, and can be stored in a log le with i.e a little command like this:

```
cat /proc/161/pH | sed -n ’s/profile-count : //p’ >> logfile
```

7.5 Discrete anomaly countermeasures

The need to tell pH which processes to react to arises when we want to use pH i a production environment. To illustrate, imagine a user who compiles and runs his rst program ever. As soon as he starts executing, pH blocks the process. The user won't realize why his program doesn't work.

The functionality we want, is the ability to set a variable for each pro le which says if processes from this binary are to be blocked or not. In addition, we want to be able to de ne a global variable describing the default value for all new pro les. The list of processes and the global variable could be managed by i.e cfengine.

This is a functionality which requires a considerable modi cation to the source code.

7.5.1 Alternative I: Utilising the file-system

As noted earlier, all binaries are stored on disc. The easiest way to implement this requirement is to add a le in the same directory as the pro le with the same name save for a suf x. As an example: The program `less`, which is located in `/usr/bin/less`, has its pro le in `/var/lib/pH/profiles/usr/bin/less`. By creating the empty le `/var/lib/pH/profiles/usr/bin/less.no_delay`, pH could stop delaying that program.

The upside to this alternative, is the ease of implementation. A suitable place for pH to look for the le, would be in the method `pH_delay_task(pH_task_state *s, int` If the le should exist, then a different message would appear in the log and the process would be saved.

The downside is performance and spreading information about one pro le to more than one place.

```
inline void pH_delay_task(pH_task_state *s, int delay_exp)
{
        if ((pH_delay_factor > 0) && (delay_exp > 0)) {
                unsigned long delay, eff_delay;
                const int max_delay_exp = sizeof(delay) * 8 - 2;

                if (delay_exp > max_delay_exp)
                        delay_exp = max_delay_exp;
                delay = 1 << delay_exp;
                eff_delay = delay * pH_delay_factor;
                action("Delaying %d at %lu for %lu jiffies",
                       current->pid, s->count, eff_delay);
                pH_do_delay(delay);
        }
}
```

7.5.2 Alternative II: Modifying the profiles

A second alternative is to modify the pro le-struct to contain a new variable called `trusted`. A program will not be delayed if its `trusted`-variable should be set to 1. The test on the variable will happen in the same method mentioned above.

7.5.3 Implementation

This functionality requires substantial modi cations to the code compared to the rst alternative. We have to change the struct, the methods for initialising pro les, the methods for reading and writing them to an from disc and the method for displaying them in the `/proc` directory.

The bene ts would be a better design. Distributing attributes over several les is never a good solution. Also, bringing this variable into the struct has the additional bene t as to simplifying future changes in the code.

Although this is the most elegant solution, the rst alternative has the advantage of ease and will therefore be a suf cient choice for prototyping. In terms of design, the latter alternative wins.

Chapter 8

Phase Three: Requirements for a Distributed Immune System

Let us now turn towards a more higher level view on building a computer immune system. What have we learned from this project and how will our experiences help us in our future projects?

One point we noticed, is that there is often no clear interface between two ADS, especially if they are both complex to some degree. In essence this is what we try to de ne, though. The process can be viewed from a distributed systems angle: how to make single and independent components work together like a distributed system? Distributed systems are information-sharing systems. What information is important?

Second, the thing about ADS, is that they monitor the the same computer system they use as an arena for communication. The accuracy of the computer immune system would decrease should interaction between components introduce noise into the monitoring processes.

We now present the requirement model we found comprehensive enough to encompass all the obstacles that we encountered. If we had used two different systems, then these requirements would perhaps have a different focus, but essentially the same.

8.1 Defining requirements

There currently no standarized method to test whether two systems affect each other. There is also no standarized way of describing the requirements needed for two anomaly detection systems to be able to share data or at all to pro t from it. We will now de ne our requirements.

These requirements should be viewed as base for two or more systems. By adding more systems, the complexity increases exponential, since the new system has to be compatible with every other participant.

8.1.1 Definitions

Channel A communication channel between two or more ADS, where one or more ADS writes information and one or more ADS read and interprets the information. This channel can exist without any ADS actually setting it up or knowing who's reading and writing. `syslog` can be viewed as a channel if one ADS writes that it found an anomaly, and another ADS reads this and takes counter-measures of it's own.

Trigger The indicator of an anomaly. An ADS keeps track of variables describing some aspect of the system. Should one or more of these variables violate any de ned constraints, then this would trigger the ADS to an reaction.

Reaction The behaviour of an ADS in response to a detected anomaly. This could be anything from just logging the event to system integrity checks to actively killing processes.

8.1.2 Compatible abstraction level

Different ADSs have different views of the system. PH has a per-process view. It considers a process of only a sequence of system calls. Other ADSs, like cfengine, view the system as consisting of among other things, checksums, permissions on le-systems and other remote machines which it has to interact with. If the two systems would have no mutual perspective, they would not be able to help each other.

We solved this by adding a new perspective in cfengine, the ability to view processes as a sequence of system calls with possible deviations in the pattern. This is the same as in pH.

We also modi ed pH to offer a data-variable which matches an already existing system perspective in cfengine: a process is a time-series variable which can be monitored and analysed. The modi cation in pH enabled cfengine to monitor even more processes.

8.1.3 Increased overall functionality

Another requirement is that there should be an actual (or at least possible) gain in functionality. In psychology, the gestalt-theory states that an entity is perceived as more than the sum of its components. This is also applicable here. How we de ne functionality depends on the initial tasks the different systems have. It can be immediate gains, like better performance, higher accuracy (fewer false positives), new detection methods or better and more accurate reaction capacity.

These mentioned improvements will then lead to more control over the system, higher security and a more stable system.

But let's not forget the possible bene t of redundancy. Two systems to monitor the same trigger can be an advantage if one of them would go down. This gives higher fault tolerance, which can be interpreted as better functionality. It could also lead to higher accuracy if the two systems could compare the detected anomaly.

8.1.4 Compatible and predictable data representation

This is only a point for the systems which actually share data (knowing or unknowing) between each other. As an example: say cfengine polls a variable from pH residing in the `/proc` directory regularly. It expects it to be an integer which should denote the global count of system calls for a certain pro le. What if this variable suddenly contains -1? Or perhaps the string “two thousand”. Although this example seems a bit far fetched, its still an important point: Can the systems rely on the data? Is the data always describing what they think it is? If not, then the risk is high for so-called false-positives or errors in general.

This brings us to another important point: could evil intentions render data- les or con guration les and make the immune system react against itself? What if pH would start to delay cfengine? This is an classic problem in system administration: When placing more trust on a system, you have to place the same amount of trust on all its dependencies. In this case le- and user permissions.

This is a requirement also to the environment in which the computer immune system is meant to function. A predictable data representation means that we can trust the the systems ability to offer le-protection.

8.1.5 Zero interference / No trigger loop

This requirement concerns the reaction-patterns of a ADS. What if one system detects a anomaly, reacts to it and thereby causes the alarm-bells to go off in another ADS, witch itself sets to work with considerable system-checks causing even more anomalies in the rst ADS? These trigger loops could bring the whole system down in no-time and saving it would mean shutting down security-services and making the system vulnerable attacks and dependent on a system administrator to help, the direct opposite of our intention.

This could to some extend happen in the case of pH an cfengine. If pH should delay a process really hard, then this would trigger an anomaly in the part of cfengine that logs the load of the process, nding it to have a much lower load than expected.

A solution is to make enough information as possible about reactions available to the other systems. If cfengine knew that pH was delaying the process it could ignore its own trigger.

Another problem, which is more subtle, is the interference that ADS could cause on each other. This is important for systems who rely on data inspection with a low noise tolerance. Should an ADS prove to introduce more noise than tolerable we would loose accuracy and functionality. This is harder to detect by code inspection and has to be tested. There is no standard way for measuring the effect of one system on the others dataset, which imposes an even greater dif culty for detection of future noise sources.

We approached this with our model for comparing two initially similar systems and test the divergence when the new ADS was set to work on only one of them.

8.1.6 Independence

An important requirement regards to keep the independence of the different ADSs. If one should go down, the others should be able to continue their work with as much loss in security or functionality as possible. This, of course, depends on how tight they are woven together. Also, should a ADS have to be stopped it should be able to resume its interaction with the other systems without having to restart them as well.

This requirement holds regardless if you have a system to monitor and restart eventual failing services. This would lead to an weak point, where an attacker or failure only had to target the "mother" process or the monitor and by that rendering the immune system vulnerable or even useless.

Independence complicates the system since it can introduce unpredictability. In addition, how will turning one system on or off affect the data of others? How could this affect learning systems? As we learned in our experiments, the degree of in uence varies with the type of data.

8.1.7 Openness / Availability

The process of making a ADS work interact with others requires close inspection of the systems design and inner workings and reprogramming. This is mostly to be able to ful l all the requirements mentioned above. Surely, if every developer would keep its designs and code for himself, one could make a solution based on only meetings and testing. But our experience is that the best ideas that lead to the discovery of a new functionality came through inspection of the source-code and internal data-representation. Available source code and a transparent view of the operating system is crucial for the design and development of an computer Immune system.

This topic is probably worth a longer discussion, but unfortunately that is not in the scope of this research. As a last note, it is worth pointing out, that many different projects try to establish open standards in the eld of intrusion detection. One of these project is The Intrusion Detection Working Group, who develop a XML-standard for detected intrusion events [41].

Chapter 9

Discussion and Conclusions

9.1 Phase one

How does a system using pH differ from a normal one on a overall system perspective?

To answer this question, we conducted four experiments on two initially identical computer systems. The two systems where modi ed between each experiment, resulting, in the end, in one system running pH and the other not. In the last experiment the systems ran a webserver which was stressed periodically.

A system was created to generate a statistical summary of each experiment together with plots.

Our experiment showed us, that the different variables react differently to the changes. Memory usage showed a clear drop in similarity between the two systems. This is most likely caused by the extra le-handling by

Con guration	Description
`Default`	`Both machines run the same installation-default kernel`
`pH-Passive`	`2.2.19 on remus and 2.2.19-ph on romulus. PH is disabled.`
`pH-Active`	`Same as pH-passive, but pH is enabled on romulus`
`pH-Load`	`Same as pH-Active, but the machines are running apache with instances of high load.`

Table 9.1: Explaining the different con gurations

the system running pH. Also, pH generates more messages to log- les like `syslog`.

Other variables, like `bi` and `us` had a much more stable value through all tests.

The correlation coef cient we used handled peaks and outliers better than expected. This is due to the transformation of the data before the actual computation. Another point, is that the results seemed to be similar when using smoothed data instead. This opens for longer intervals between measurements in similar experiments in the future.

Our analysis only took into account the linear dependency between each variable in isolation. No multivariate tests where used and we can have missed important hints because of that.

A limitation of our tests was that the systems where not tested under more different conditions. Most of the data came from inactive systems. In retrospect, we recognise that we would have a better foundation if we used less time on each trial (especially the rst one) and instead more trials where we stressed different services and perhaps also simulated user-interaction.

There is a clear difference between the two systems when one of them uses pH. But this is only when the systems are doing almost nothing. Will the difference still be noticeable when the systems are used under more real-life conditions? Our data cannot answer that question.

9.2 Phase Two

How could a process based anomaly detection system be incorporated into a more generic con guration engine like cfengine?

Here we settled on two types of anomalies that cfenvd can detect based on available data from pH:

1. A “symbolic” approach where a process generates an anomaly event based on one or more alien system call pairs.
2. A quantitative approach where a process experiences a different load compared to an established statistical pro le.

The rst type of anomaly has already a reaction pattern in pH - delays. The second type was discovered a bit unexpected and offers a new perspective to cfengines statistical anomaly detection model. There are more scenarios where these two systems could gain even more functionality, described both in the paper and in this report, but we consider them in need of more research before we can conclude anything. Regrettably we did not nd enough time to design possible future experiments for these.

We found that simple modi cations to both systems actually expanded their capabilities. This holds especially for cfengine, since it could use the same analysis on some variables from pH. Cfengine can now monitor processes closely without modifying the binary or running it in a special environment. Some modi cation was done to the systems, to accommodate the new functionality. They proved the concept but have still to show how well it will work under real attacks.

Analysing two systems to this extent is time-consuming. There is no framework to be used for building a distributed anomaly detection system and we stumbled over new questions and problems along the way. This is also what motivated the next part of this project. Hopefully will the experience gained in this project speed up a similar analysis in future projects.

One other type of functionality we did not implement was the ability for pH to set if a process belonging to a given binary shall be subject to delays or not. Cfengine could manage the list of trusted / untrusted binaries and perhaps adjust it in runtime. Time constraints only allowed for the description of this functionality and not further. This will, however be an important point in an already planned future project. This functionality enables us to use cfengine an pH together in production environments and that makes further data-collection from real-life systems possible.

The result of this phase of the project culminated to a paper which got accepted at The Eighth IFIP / IEEE International Symposium on Integrated Network Management (IM 2003), which is gratifying.

9.3 Phase Three

What requirements exist for the development of a computer immune system?

Our requirement speci cation is strongly in uenced by the types of system we analysed. A new analysis using two different systems would probably bring a different and healthy light to distributed anomaly detection.

In a distributed computer immune system every component needs available interfaces and clear de ned communication channels. We focused on combining two existing systems, not designing a distributed system from the ground up. Our project showed that two anomaly detection systems can indeed bene t from each other if we modify them a little. We formulated the following requirements that we mean must be met by all systems in a distributed computer immune system:

- Compatible abstraction level
- Increased overall functionality
- Compatible and predictable data representation
- Zero interference / No trigger loop
- Independability
- Openness / Availability

An interesting question, is whether this approach to a distributed immune system is preferable from a software engineering point of view. One bene t is that one can incorporate systems that are developed by others. There are many excellent intrusion detection systems out there, why build a completely new system that does the same?

Another point is how well this approach can be handled from the middleware? Why not build a distributed immune system using, say CORBA or perhaps even JVM, as foundation for communication? This would at least simplify design by standardising the interfaces, but it violates the requirement of independability. The strength of our approach is that both systems still function perfectly well without each other. Further, pH would be dif cult to implement using middleware, since it is in kernel-space.

Building a Computer Immune System and Software Engineering

Now that we have discussed the bene ts and workings of a distributed immune system, let us take a look at the development process. As stated before, the approach chosen in this project was to take existing anomaly detection tools and make them interact to gain higher control and integrity. This approach raises some questions. First of all is the question of maintainability.

Let's say security tool A 1.0 is being modi ed to act as a component in a immune system. The resulting component is called Ac 1.0. Now the developers of A release a new version it, 1.1, which xes some important software bugs in the program. Who will modify this new version to become Ac 1.1? What if they release a much better version, called A 2.0. This version does not at all offer the same functionality than A 1.1, and so other immune system components lose the ability to cooperate. This type of version control problems arises when one group develops the tools and another develops the distributed immune system.

One is tempted to think in terms of evolution when discussing an computer immune system. The system can be developed in a way similar to an evolutionary process: If components help keeping the organism alive, then the organism is more likely to reproduce. As in computers: If a security con guration proves to be optimal for a speci c type of system, then it will "reproduce" to other systems with the help of other consumers installing the product or reading the documentation of how it is done. The question is if this approach is any good for the system administrator. Who would like to install and rely on a not-yet-optimal security system?

A more likely approach is to let the immune system mature in a research environment, but then the danger of using outdated and legacy tools when matured is bigger since such a research project often would not get maximum of attention. In addition, how can one guarantee the usefulness of a system that is "bred in captivity"? This is especially important for a learning systems ability to cope with real-life noise-levels. The best development process for this type of system is not part of this project, but it is central to the applicability of an immune system. As is the best way of maintaining, xing and updating of all its components.

9.4 Future Projects

In the beginning of this text, we pointed out that this project is part of achieving a higher goal: A computer immune system. At The University College of Oslo we plan to follow up this project in several ways. One project is to fully implement all the functionality we have described and to test them under real attacks and simulations. Second, we hope to integrate other intrusion-detection systems and to apply our requirements models on them to test its applicability. Third, we want to expand our system for collecting and analysing system variables to encompass more variables and perhaps offer different statistical analysis methods. This system could then be used when we test pH and cfengine in different contexts.

Bibliography

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

[2] J.O. Kephart. A biologically inspired immune system for computers. Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems. MIT Press. Cambridge MA., page 130, 1994.

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

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

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

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

[7] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. In Proceedings of 1996 IEEE Symposium on Computer Security and Privacy (1996).

[8] M. Burgess. Cfengine as a component of computer immune-systems. Proceedings of the Norwegian conference on Informatics, 1998.

[9] D. Kesdogan R. Büschkes, M. Borning. Transaction-based anomaly detection. Proceedings of the Workshop on Intrusion Detection and Network Monitoring (USENIX Assosiation, 1999).

[10] Peter G. Neumann and Phillip A. Porras. Experience with emerald to date. Proceedings of the Workshop on Intrusion Detection and Network Monitoring (USENIX Assosiation, 1999).

[11] V. Paxson. Bro: A system for detecting network intruders in real time. Proceedings of the 7th security symposium. (USENIX Association: Berkeley, CA), 1998.

[12] Joseph M. McAlerney Stuard Staniford, James A. Hoagland. Practical automated detection of stealthy portscans.

[13] Syed Masum Emram Nong Ye, Qiang Chen and Kyutae Noh. Chi-square statistical pro ling for anomaly detection. Proceedings of the 2000 IEEE Workshop on Information Assurance and Security, 2000.

[14] M. Burgess. The kinematics of distributed computer transactions. International Journal of Modern Physics, C12:759–789, 2000.

[15] S. Elbaum and J.C. Munson. In Proceedings of the Workshop on Intrusion detection and network monitoring, USENIX, 1999.

[16] The portsentry homepage. http: / / www.psionic.com / products / portsentry.html.

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

[18] pH: process Homeostasis. http: / / www.cs.unm.edu / soma / ph.

[19] cfengine's homepage. http: / / www.iu.hio.no / cfengine.

[20] E. H. Spafford S. Kumar. A pattern maching algorithm model for misuse intrusion detection. The COAST Project, Department of Computer Science, Purdue University.

[21] Aurobindo Sundaram. An introduction to intrusion detection. ACM (electronic publication) http://www.acm.org/crossroads/xrds2-4/intrus.html.

[22] Next-Generation Intrusion Detection System (NIDES) Home Page. http: / / www.sdl.sri.com / projects / nides / index5.html.

[23] Jeffrey O. Kephart, Gregory B. Sorkin, Morton Swimmer and Steve R. White. Blueprint for a computer immune system. Proceedings of the Seventh International Virus Bulletin, 1997.

[24] R. Sekar, T. Bowen, and M. Segal. On preventing intrusions by process behaviour monitoring. Proceedings of the workshop on intrusion detection and network monitoring, USENIX, 1999.

[25] Mark Burgess. Two dimensional timeseries for anomaly detection and regulation in adaptive systems. 13th International Workshop on Distributed Systems: Operations and Management (DSOM 2002).

[26] A. S. Tanenbaum. Modern Operating Systems. Prentice Hall, 2nd edition (2001).

[27] Lawrence R. Halme and R. Kenneth Bauer. Aint misbehaving - a taxonomy of anti-intrusion techniques. Proceedings of the 18th National Information Systems Security Conference, pages 163–172.

[28] AMaViS A Mail Virus Scanner. http://amavis.org/.

[29] DialogueScience Antivirus Kit. `http://www.dials.ru/english/dsav_toolkit/drwebunix.htm`

[30] 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).

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

[32] M. Burgess. A site con guration engine. Computing systems (MIT Press: Cambridge MA), 8:309, 1995.

[33] G. A. Leinert. Verteilungsfreie Methoden in der Biostatistik,. Verlag Anton Hain, 1973.

[34] David S. Moore and George P. McCabe. Introduction to the Practice of Statistics, Third Edition. Freeman, 1999.

[35] William T. Vetterling William H. Press, Saul A. Teukolsky and Brian P. Flannery. Numerical Recipes in C: The Art of Scienti c Computing. Cambridge University Press, (C) 1992.

[36] Bhattacharyya and Johnson. Statistical Concepts and Methods. Wiley, 1977.

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

[38] M. Burgess. GNU cfengine. Free Software Foundation, Boston, Massachusetts, 1994-.

[39] 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.

[40] 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.

[41] Intrusion Detection Working Group Intrusion Detection Message Exchange Format. www.silicondefense.com/idwg/draft-ietf-idwg-idmef-xml-03.txt.

Appendix A

Statistical summary for `Default`

This appendix contains the results from the rst trial in the rst phase of this project. Some of the variables where zero throughout the trial, and produced therefore no plot. These empty plots have been removed from this appendix.

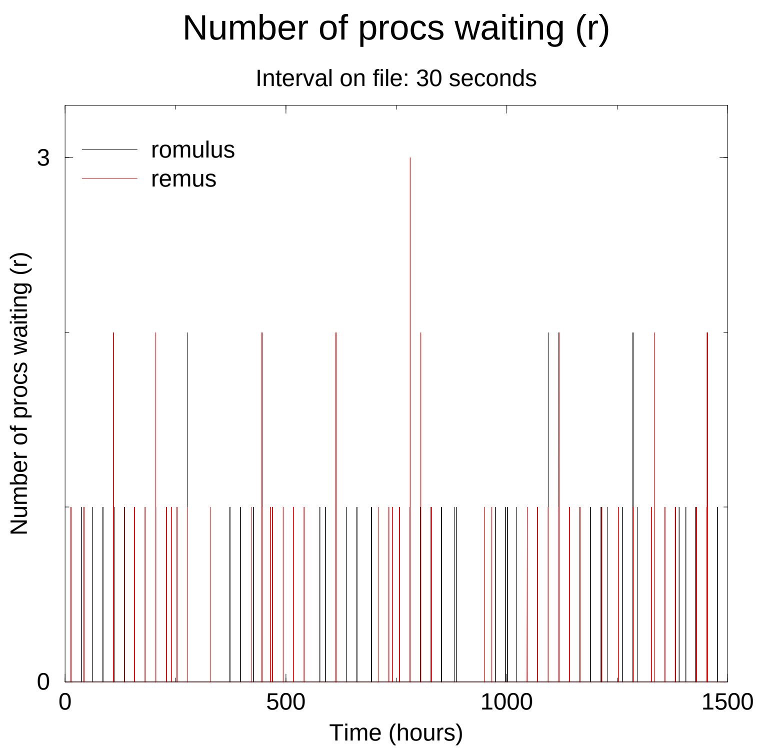


Figure A.1: This plot shows the number of precesses waiting.

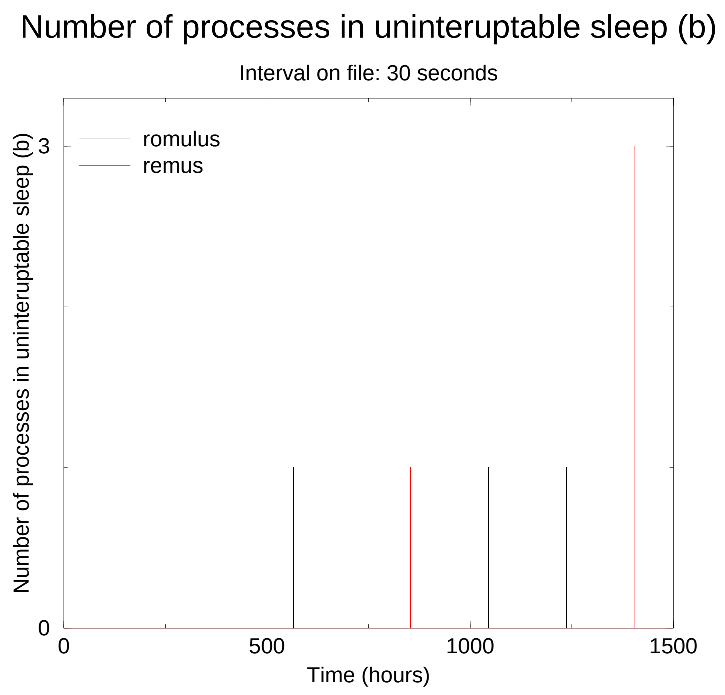
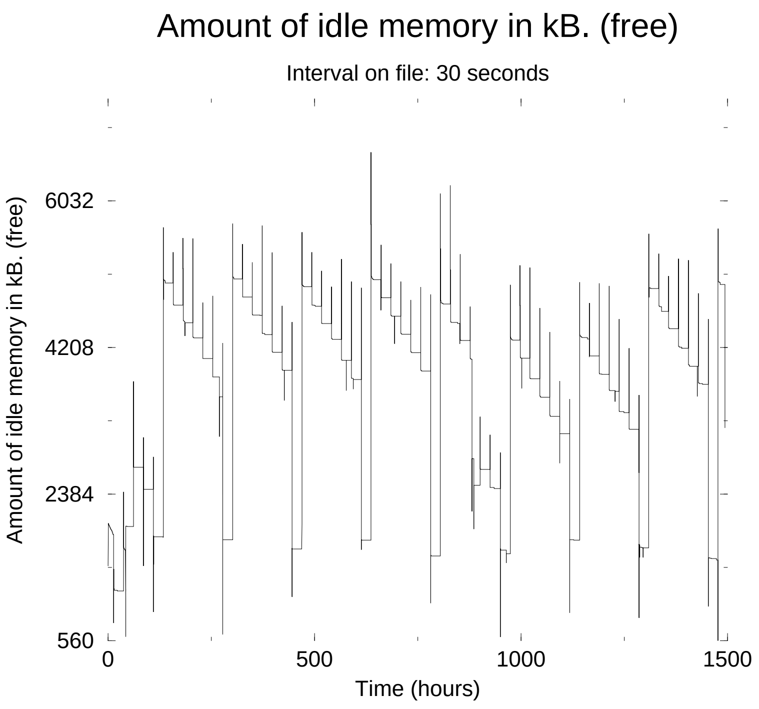


Figure A.2: This plot shows the number of processes in uninteruptable sleep. As we can see, there is not much activity in this variable.


Amount of idle memory in kB. (free)
Interval on file: 30 seconds
Amount of idle memory in kB. (free)
6032
4208
2384
560
0
500
1000
1500
Time (hours)

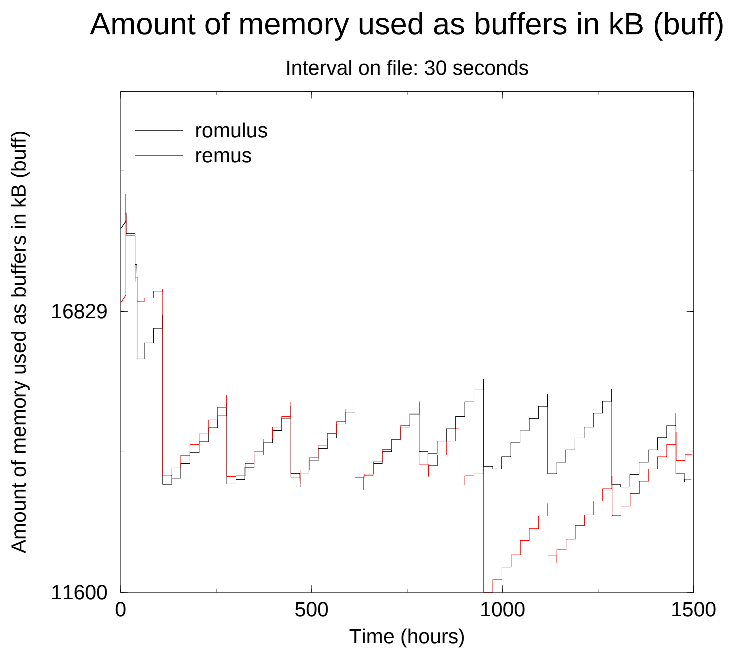


Figure A.4: This plot shows the amount of memory used as buffers. The same waves as in the previous plot are recognisable here.

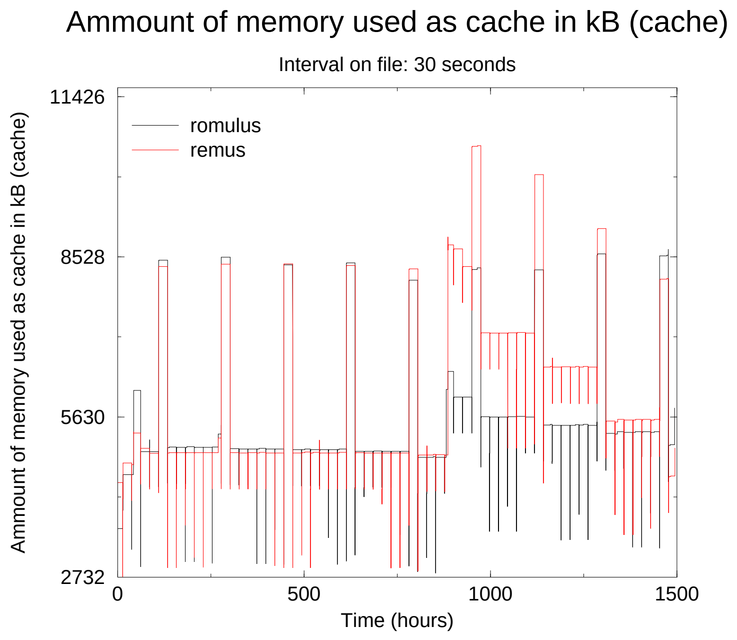


Figure A.5: In this plot we see the same pattern as in the other plots regarding memory usage.

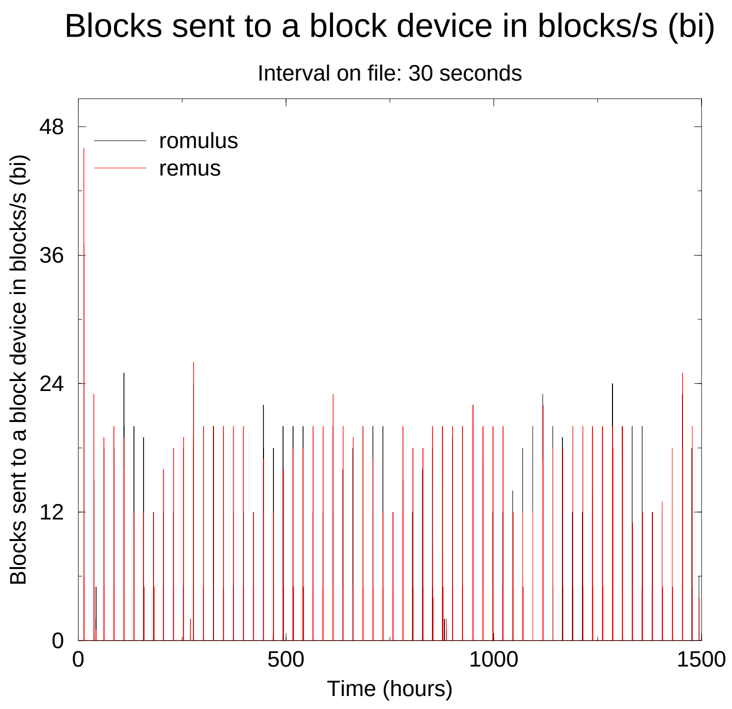


Figure A.6: This plot shows the amount of blocks sent to a block device. The plot may seem unorganised, but it shows a pattern similar to the plots regarding memory usage.

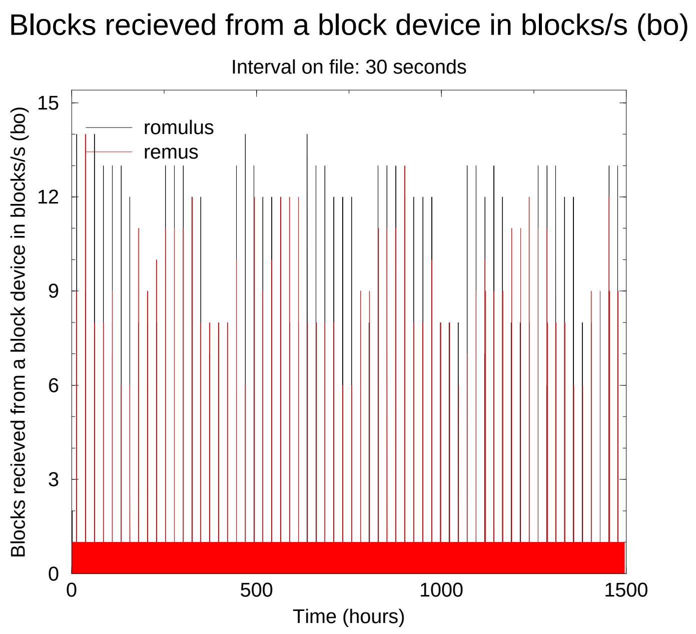
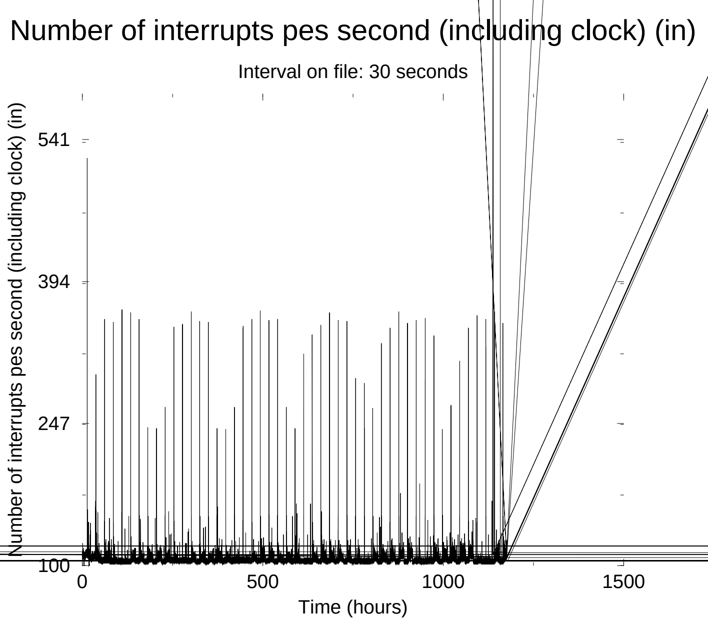


Figure A.7: In this plot we recognise a weekly pattern, although distorted.


Number of interrupts pes second (including clock) (in)
Interval on file: 30 seconds
Number of interrupts pes second (including clock) (in)
541
394
247
100
0
500
1000
1500
Time (hours)

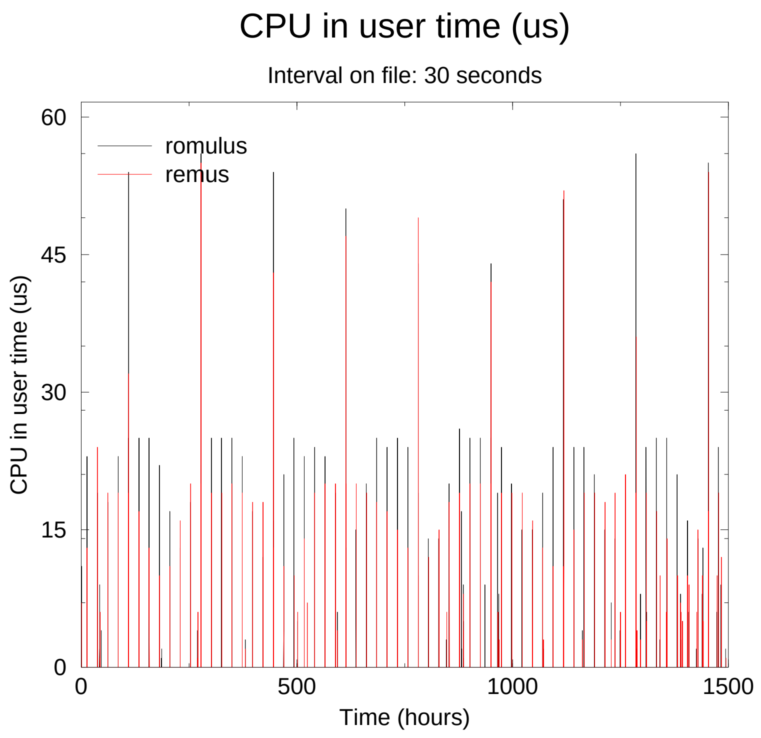


Figure A.10: CPU in user time. Note how both machines show similar behaviour. The peaks coincide here also.

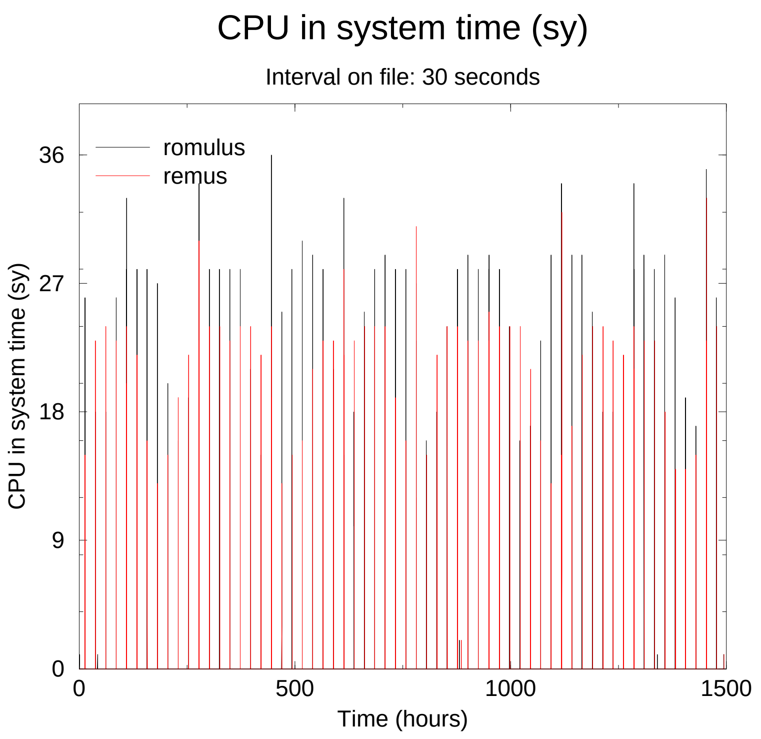


Figure A.11: CPU in system time. This plot is in many ways similar to the previous one in that it shows a different perspective on the same resource - CPU usage. A wavy pattern is recognisable on both machines.

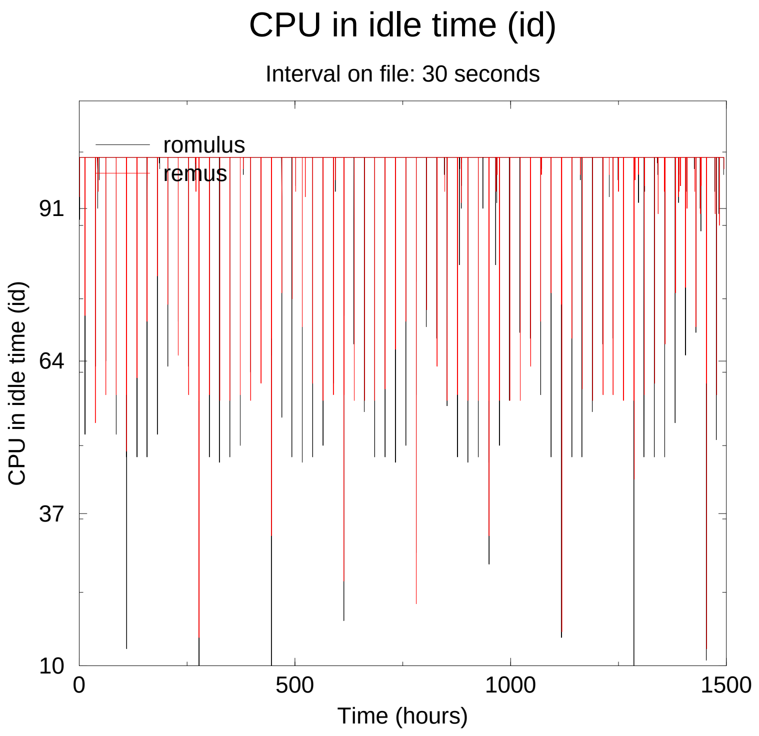


Figure A.12: This plot, as with the two previous ones, shows a periodic an constant behaviour on both machines.

variable	n	sum	max	average	stdDev
r	179229	76	2	0.000	0.023
b	179229	3	1	0.000	0.004
w	179229	0	0	0.000	0.000
swpd	179229	0	0	0.000	0.000
free	179229	656077548	6636	3660.555	1121.153
buff	179229	2600673264	18664	14510.337	882.285
cache	179229	1018055512	8668	5680.194	1165.853
si	179229	0	0	0.000	0.000
so	179229	0	0	0.000	0.000
bi	179229	1857	37	0.010	0.389
bo	179229	3947	14	0.022	0.262
in	179229	18710965	522	104.397	5.738
cs	179229	184294	171	1.028	1.404
us	179229	2441	56	0.014	0.578
sy	179229	2189	36	0.012	0.535
id	179229	17918272	100	99.974	1.095

Table A.1: vmstat_phase1.romulus

variable	n	sum	max	average	stdDev
r	179282	61	3	0.000	0.022
b	179282	4	3	0.000	0.007
w	179282	0	0	0.000	0.000
swpd	179282	0	0	0.000	0.000
free	179282	665360252	6636	3711.250	1240.858
buff	179282	2516836280	19016	14038.421	1235.697
cache	179282	1091583352	10540	6088.639	1506.088
si	179282	0	0	0.000	0.000
so	179282	0	0	0.000	0.000
bi	179282	1848	46	0.010	0.399
bo	179282	3785	14	0.021	0.232
in	179282	18708735	535	104.354	5.576
cs	179282	184115	186	1.027	1.391
us	179282	1946	56	0.011	0.489
sy	179282	1810	36	0.010	0.453
id	179282	17924448	100	99.979	0.923

Table A.2: vmstat_phase1.remus

variable	average of both
r	0.000
b	0.000
w	0.000
swpd	0.000
free	3685.902
buff	14274.379
cache	5884.416
si	0.000
so	0.000
bi	0.010
bo	0.022
in	104.375
cs	1.028
us	0.012
sy	0.011
id	99.977

Table A.3: average

variable	r
r	-0.0002899
b	-0.0000122
w	0.0000000
swpd	0.0000000
free	0.9488736
buff	0.7191931
cache	0.8523288
si	0.0000000
so	0.0000000
bi	0.0072123
bo	0.0016221
in	0.1640182
cs	0.0056725
us	0.0024800
sy	0.0018850
id	0.0000000

Table A.4: Pearson-Bravai Correlation

variable	Spearman's rho
r	0.999
b	1.000
w	1.000
swpd	1.000
free	0.931
buff	0.462
cache	0.899
si	1.000
so	1.000
bi	0.997
bo	0.949
in	0.735
cs	0.996
us	0.997
sy	0.998
id	0.997

Table A.5: Spearman's rho

Appendix B

Statistical summary for `pH-Passive`

This appendix contains the results from the second trial in the rst phase of this project. Some of the variables where zero throughout the trial, and produced therefore no plot. These empty plots have been removed from this appendix.

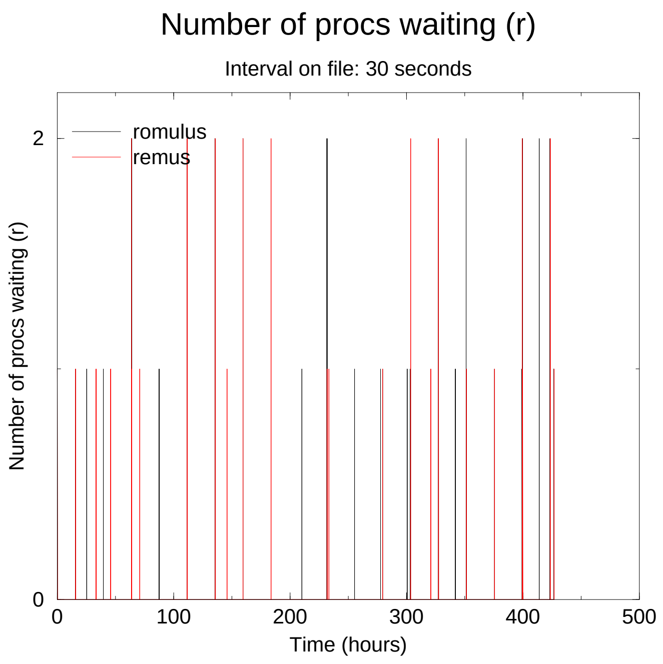


Figure B.1: This plot shows the number of precesses waiting.

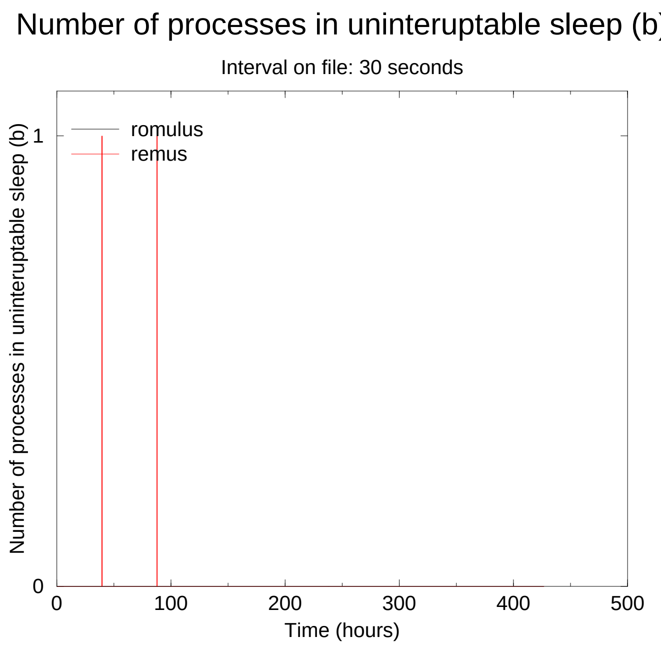


Figure B.2: This plot shows the number of processes in uninteruptable sleep. As we can see, there is not much activity in this variable.

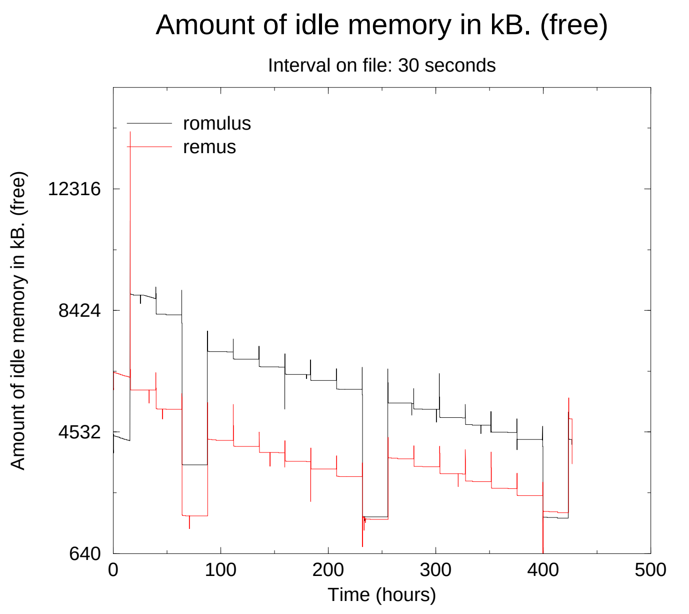


Figure B.3: Here we see the amount of idle memory in kB. The wavy pattern shows the effect of scheduled jobs every week. The decrease of idle memory is because the machines where rebooted right before this trial and therefore had initially almost only idle memory.

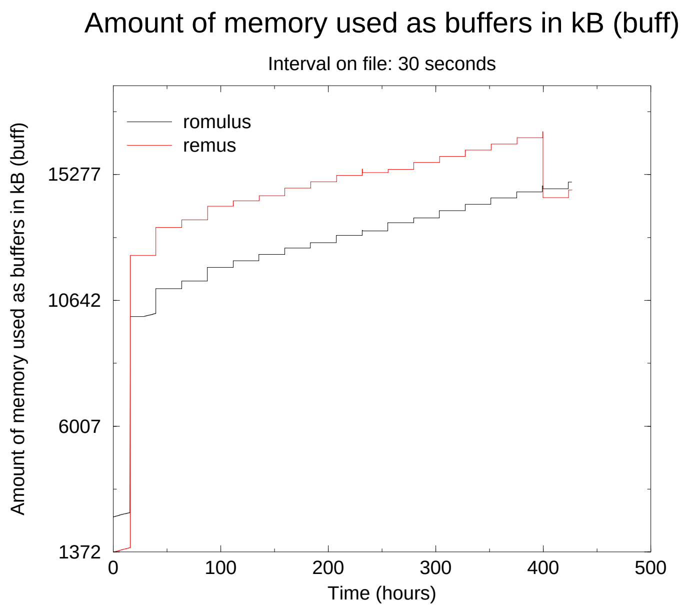


Figure B.4: This plot shows the amount of memory used as buffers. This amount increases due to a system reboot right before the trial started.

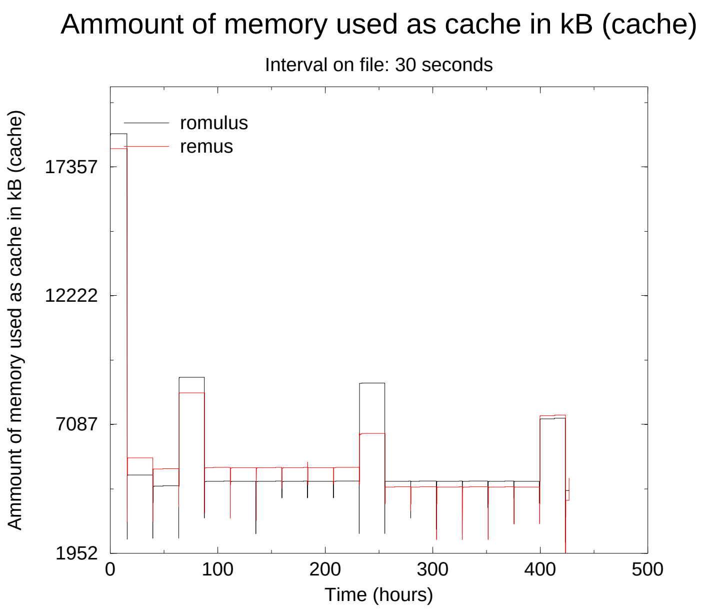


Figure B.5: In this plot we see the same pattern as in the other plots regarding memory usage.

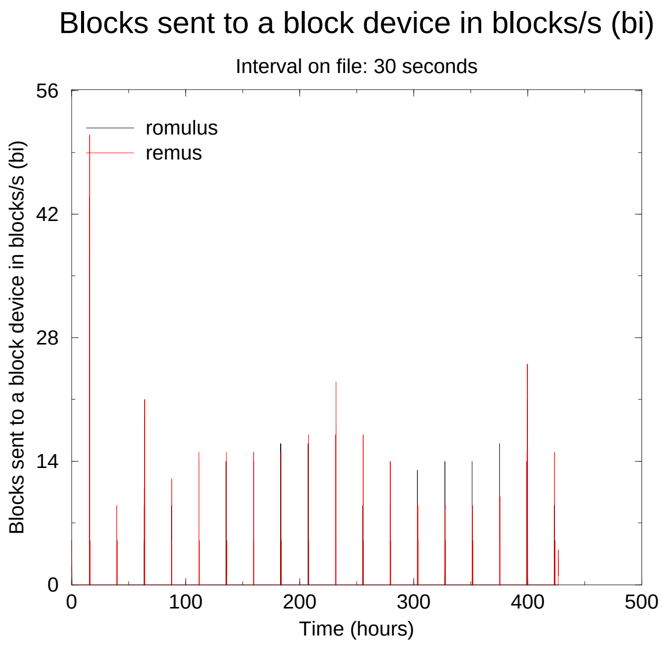


Figure B.6: This plot shows the amount of blocks sent to a block device. The plot may seem unorganised, but it shows a pattern similar to the plots regarding memory usage.

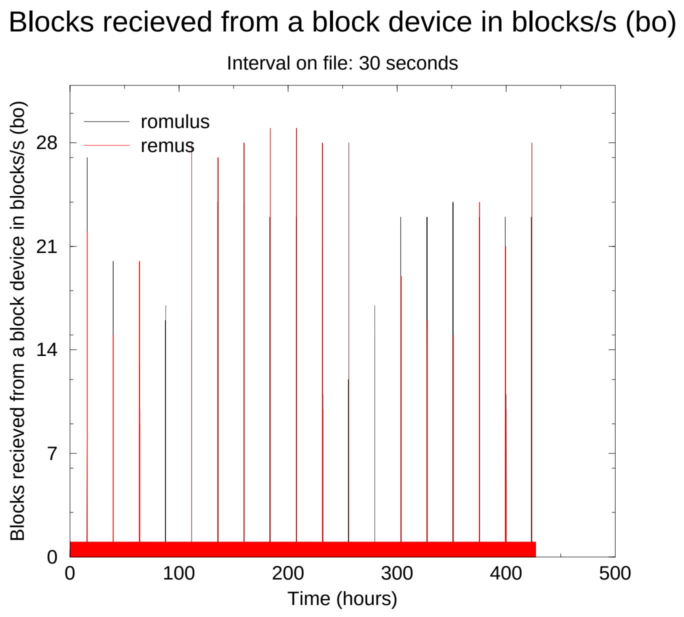
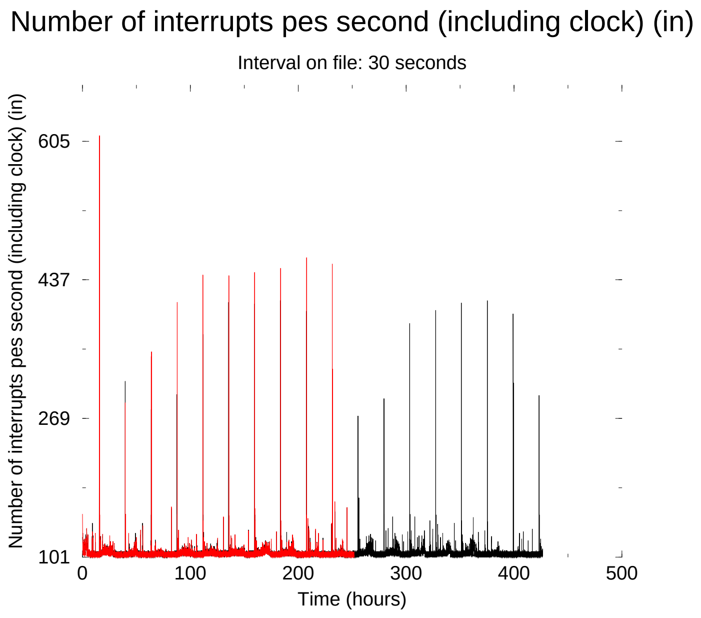


Figure B.7: In this plot we recognise a weekly pattern, although distorted.



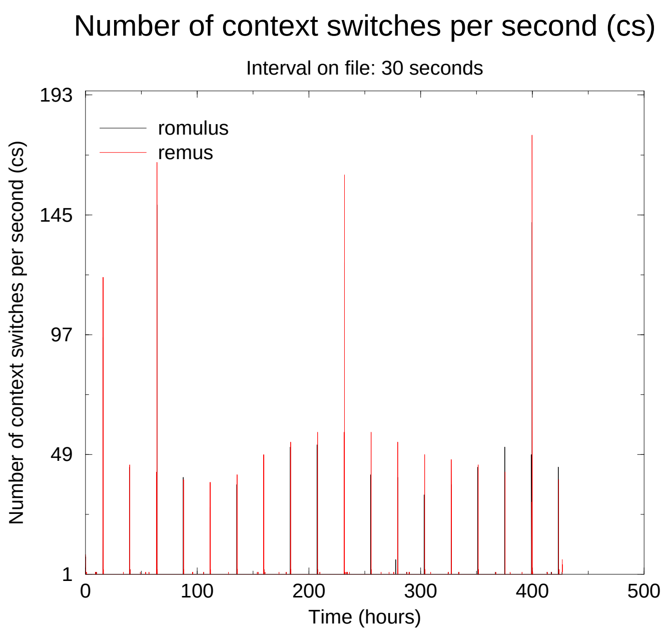


Figure B.9: Here we see the number of context switches per second. Both machines show the same behaviour.

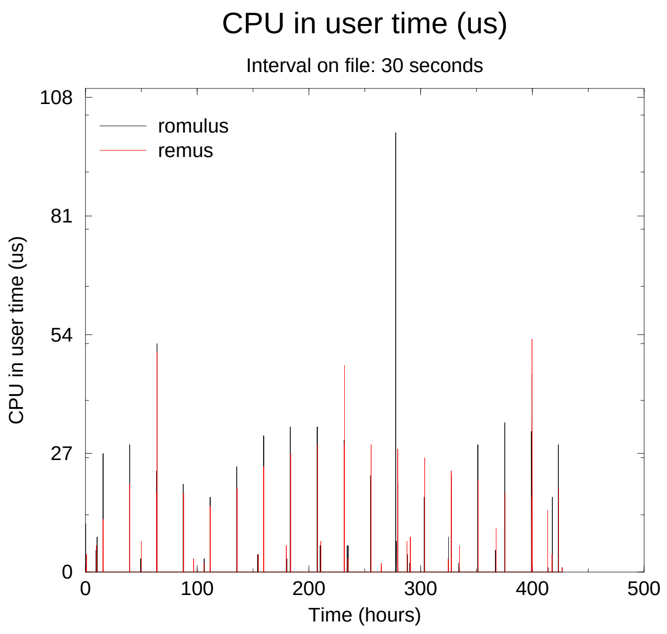


Figure B.10: CPU in user time. Note how both machines show similar behaviour. The peaks coincide here also.

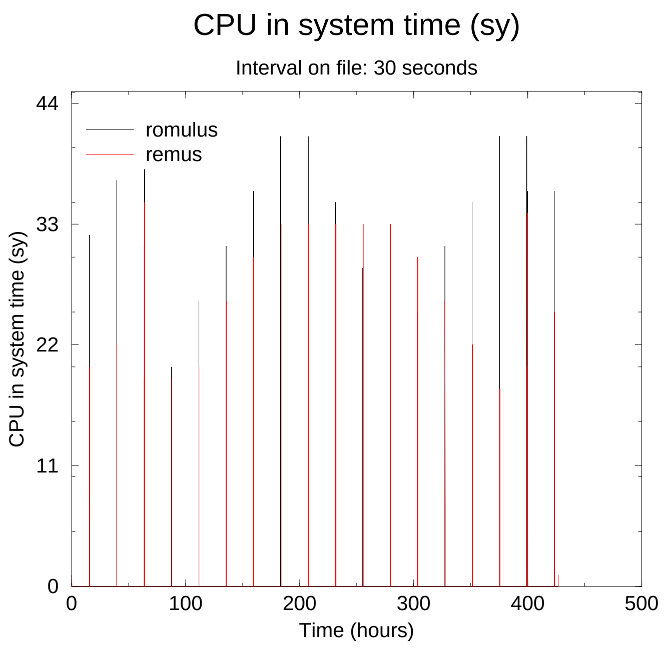


Figure B.11: CPU in system time. This plot is in many ways similar to the previous one in that it shows a different perspective on the same resource - CPU usage. A wavy pattern is recognisable on both machines.

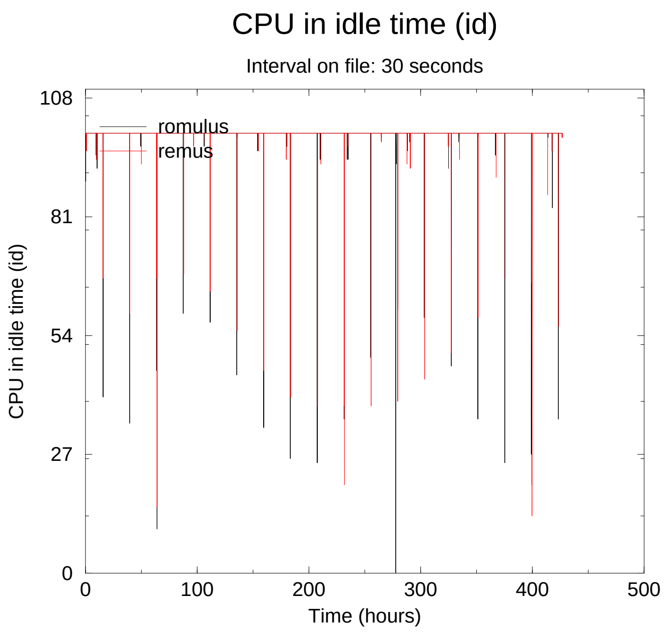


Figure B.12: This plot, as with the two previous ones, shows a periodic an constant behaviour on both machines.

variable	n	sum	max	average	stdDev
r	51173	47	2	0.001	0.037
b	51173	0	0	0.000	0.000
w	51173	0	0	0.000	0.000
swpd	51173	0	0	0.000	0.000
free	51173	276934696	11300	5411.735	1865.028
buff	51173	642502844	14992	12555.505	2292.143
cache	51173	301830452	18676	5898.236	2822.370
si	51173	0	0	0.000	0.000
so	51173	0	0	0.000	0.000
bi	51173	541	44	0.011	0.399
bo	51173	1359	27	0.027	0.441
in	51173	5301617	551	103.602	6.814
cs	51173	53025	149	1.036	1.653
us	51173	1869	100	0.037	1.474
sy	51173	926	41	0.018	0.741
id	51173	5114495	100	99.945	1.933

Table B.1: vmstat_phase2.romulus

variable	n	sum	max	average	stdDev
r	51189	35	2	0.001	0.032
b	51189	2	1	0.000	0.006
w	51189	0	0	0.000	0.000
swpd	51189	0	0	0.000	0.000
free	51189	179541960	14156	3507.432	1207.383
buff	51189	737534704	16856	14408.070	2753.149
cache	51189	303237420	18676	5923.879	2582.831
si	51189	0	0	0.000	0.000
so	51189	0	0	0.000	0.000
bi	51189	593	51	0.012	0.442
bo	51189	1435	29	0.028	0.491
in	51189	5302075	612	103.578	7.446
cs	51189	53213	177	1.040	1.778
us	51189	840	100	0.016	0.627
sy	51189	759	41	0.015	0.598
id	51189	5117302	100	99.969	1.206

Table B.2: vmstat_phase2.remus

variable	average of both
r	0.001
b	0.000
w	0.000
swpd	0.000
free	4459.584
buff	13481.787
cache	5911.057
si	0.000
so	0.000
bi	0.011
bo	0.027
in	103.590
cs	1.038
us	0.026
sy	0.016
id	99.957

Table B.3: average

variable	r
r	0.0159521
b	0.0000000
w	0.0000000
swpd	0.0000000
free	0.7681241
buff	0.9561756
cache	0.9689900
si	0.0000000
so	0.0000000
bi	0.0098541
bo	-0.0001834
in	0.0867931
cs	0.0006196
us	-0.0006484
sy	-0.0006052
id	0.0000000

Table B.4: Pearson-Bravai Correlation

variable	Spearman's rho
r	0.998
b	1.000
w	1.000
swpd	1.000
free	0.808
buff	0.844
cache	0.722
si	1.000
so	1.000
bi	0.997
bo	0.948
in	0.673
cs	0.995
us	0.996
sy	0.998
id	0.996

Table B.5: Spearman's rho

Appendix C

Statistical summary for `pH-Active`

This appendix contains the results from the third trial in the rst phase of this project. Some of the variables where zero throughout the trial, and produced therefore no plot. These empty plots have been removed from this appendix.

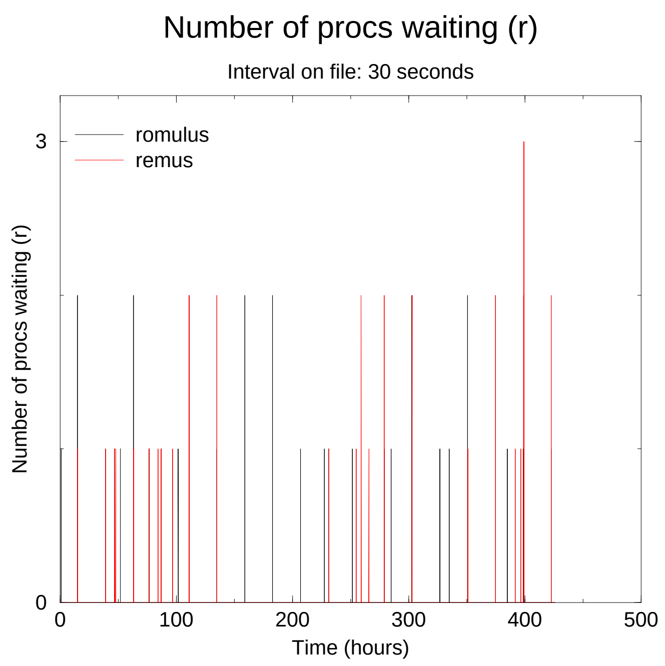


Figure C.1: This plot shows the number of precesses waiting.

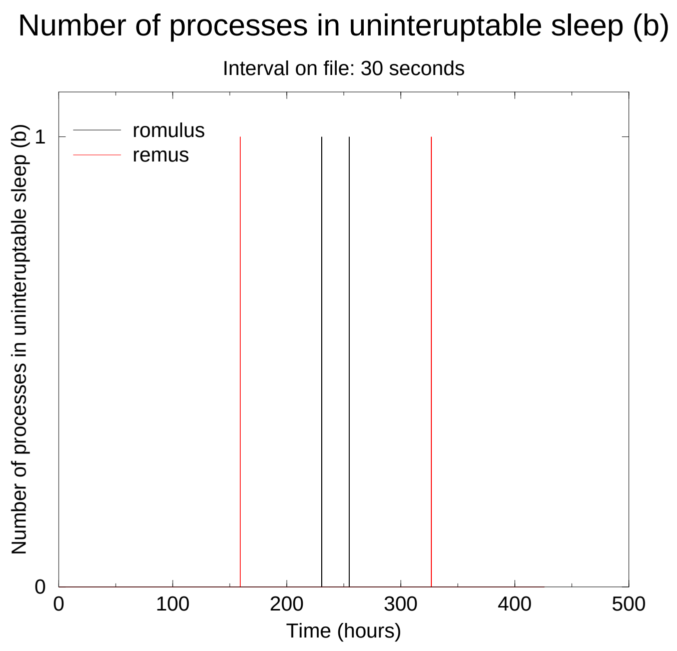


Figure C.2: This plot shows the number of processes in uninteruptable sleep. As we can see, there is not much activity in this variable.

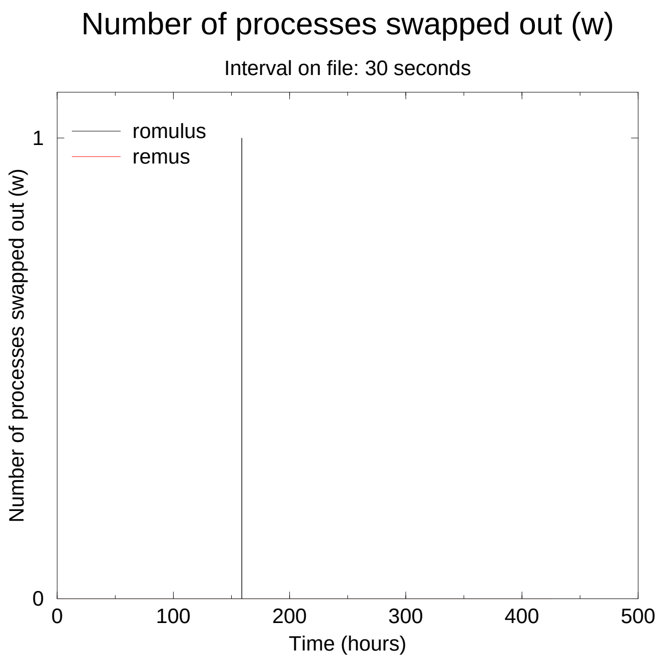


Figure C.3: Here we see the number of processes swapped out. As you can see, this only happened once.

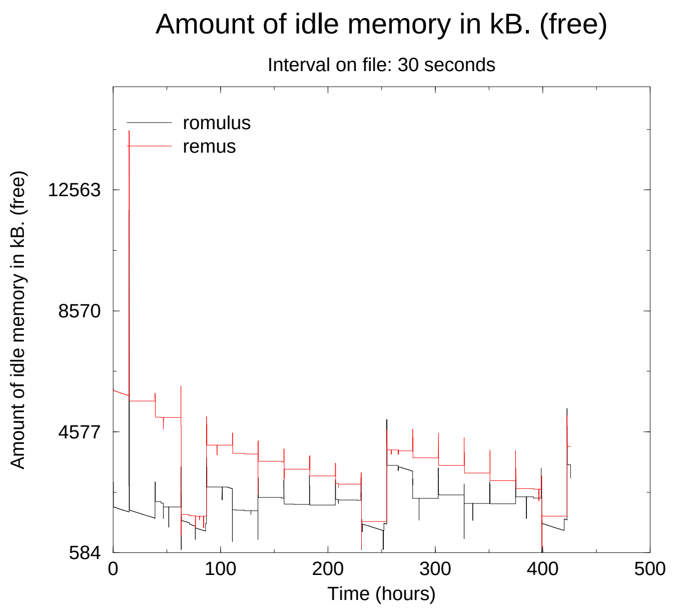


Figure C.4: Here we see the amount of idle memory in kB. The wavy pattern shows the effect of scheduled jobs every week. The decrease of idle memory is because the machines where rebooted right before this trial and therefore had initially almost only idle memory. In romulus, the amount is used faster. We also see a greater difference between the systems.

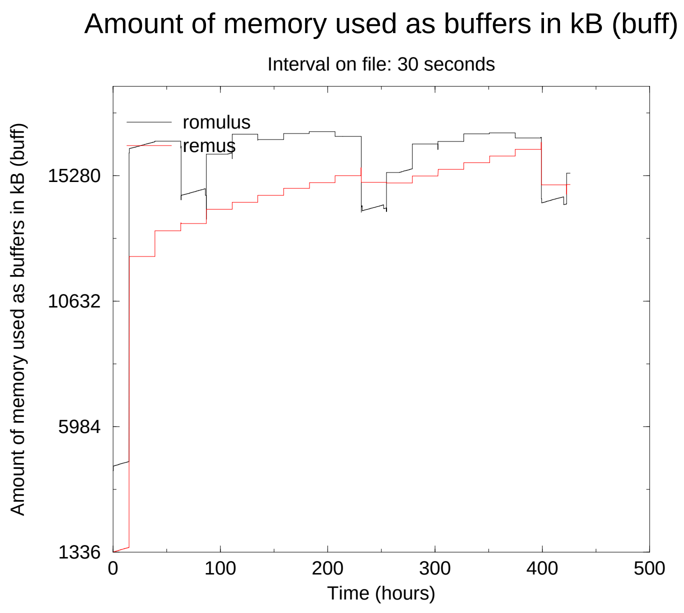


Figure C.5: This plot shows the amount of memory used as buffers. This amount increases due to a system reboot right before the trial started. A greater difference here also.

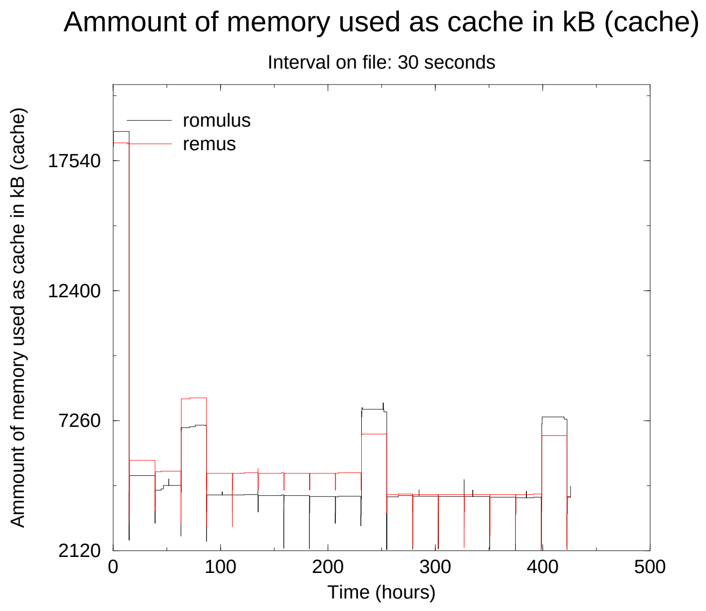


Figure C.6: In this plot we see the same pattern as in the other plots regarding memory usage.

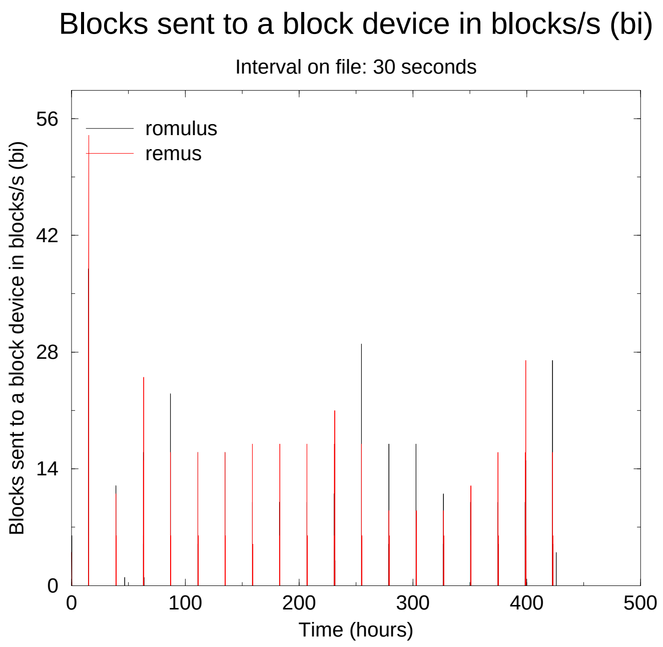


Figure C.7: This plot shows the amount of blocks sent to a block device. The plot seem more unorganised than in previous trials.

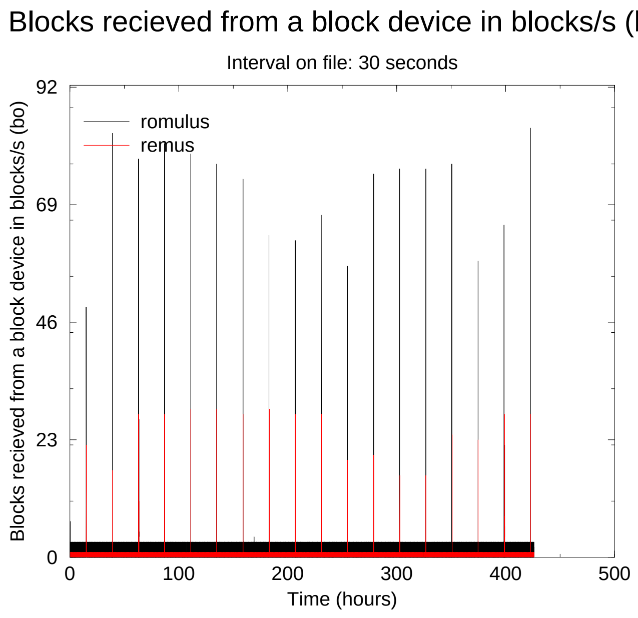
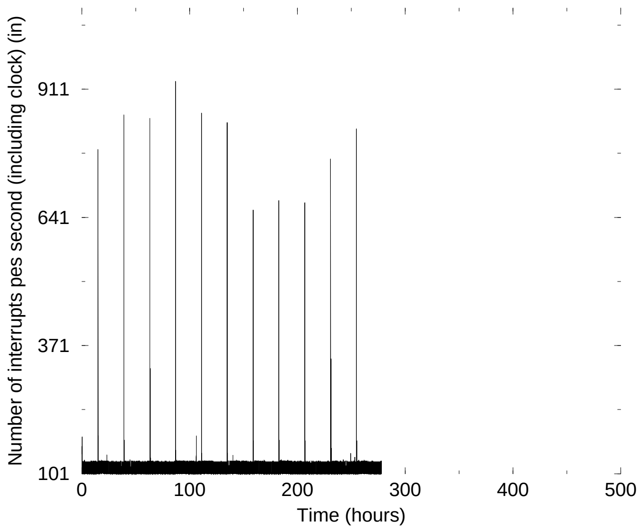


Figure C.8: In this plot we recognise a weekly pattern, although distorted.

Number of interrupts pes second (including clock) (in)

Interval on file: 30 seconds



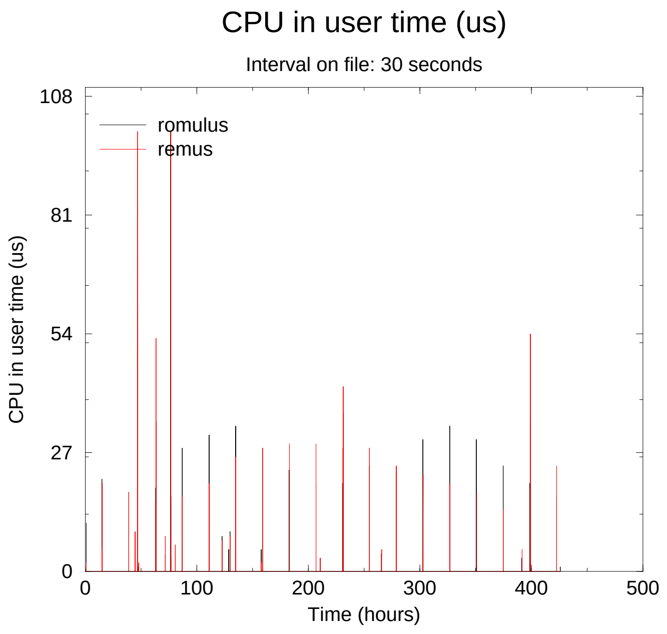


Figure C.11: CPU in user time. Note how both machines show more different behaviour. The peaks coincide here also.

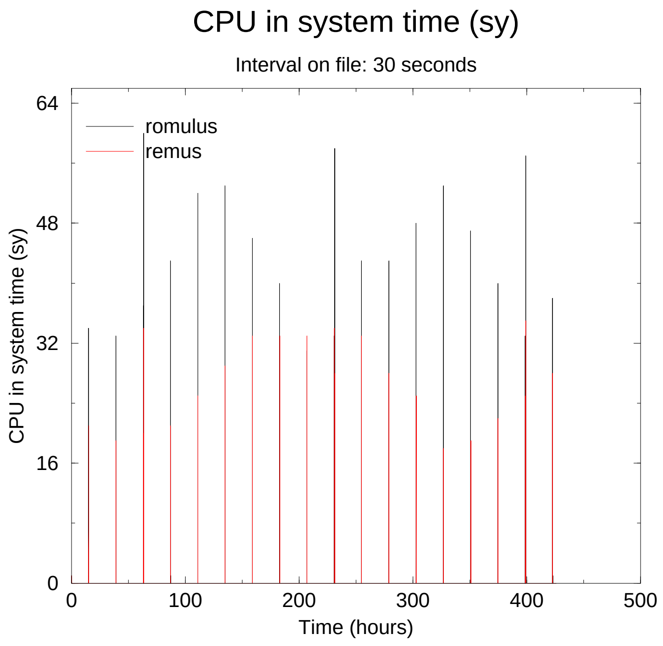


Figure C.12: CPU in system time. This plot is in many ways similar to the previous one in that it shows a different perspective on the same resource - CPU usage. A wavy pattern is recognisable on both machines.

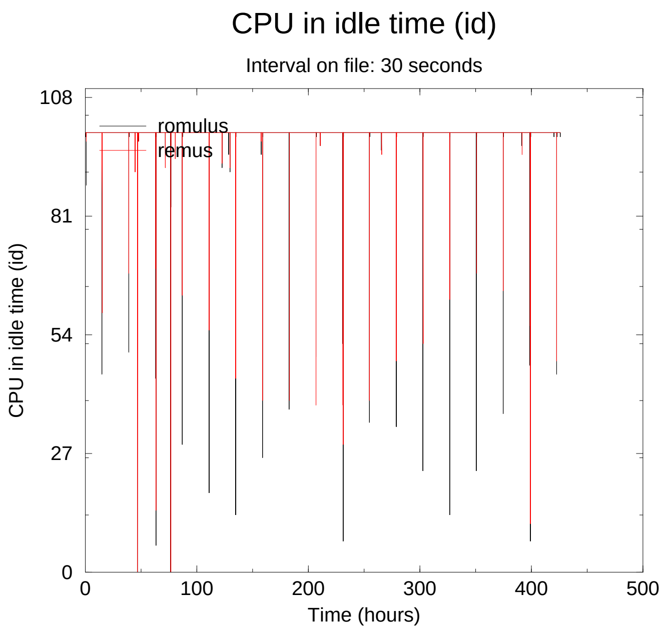


Figure C.13: This plot, as with the two previous ones, shows a more unorganised behaviour.

variable	n	sum	max	average	stdDev
r	51091	64	2	0.001	0.040
b	51091	2	1	0.000	0.006
w	51091	1	1	0.000	0.004
swpd	51091	0	0	0.000	0.000
free	51091	112745044	11920	2206.750	487.981
buff	51091	805414624	16904	15764.315	2299.953
cache	51091	273445032	18692	5352.117	2783.306
si	51091	0	0	0.000	0.000
so	51091	0	0	0.000	0.000
bi	51091	708	38	0.014	0.454
bo	51091	4660	84	0.091	1.566
in	51091	5241945	984	102.600	15.402
cs	51091	53209	115	1.041	1.603
us	51091	2756	100	0.054	1.947
sy	51091	1493	60	0.029	1.085
id	51091	5104823	100	99.916	2.519

Table C.1: vmstat.log.romulus.Thu_Feb__7_15-32-56_CET_2002

variable	n	sum	max	average	stdDev
r	51106	53	3	0.001	0.038
b	51106	2	1	0.000	0.006
w	51106	0	1	0.000	0.000
swpd	51106	0	0	0.000	0.000
free	51106	176055384	14520	3444.906	1113.676
buff	51106	730239680	16904	14288.727	2636.857
cache	51106	291608324	18692	5705.951	2590.075
si	51106	0	0	0.000	0.000
so	51106	0	0	0.000	0.000
bi	51106	615	54	0.012	0.462
bo	51106	1432	84	0.028	0.510
in	51106	5211513	984	101.975	7.255
cs	51106	53953	179	1.056	1.774
us	51106	2593	100	0.051	1.942
sy	51106	754	60	0.015	0.596
id	51106	5107255	100	99.935	2.196

Table C.2: vmstat.log.remus.Thu_Feb__7_15-32-09_CET_2002-Mon_Feb_25_09-41-47_CET_2002

variable	average of both
r	0.001
b	0.000
w	0.000
swpd	0.000
free	2825.828
buff	15026.521
cache	5529.034
si	0.000
so	0.000
bi	0.013
bo	0.060
in	102.287
cs	1.049
us	0.052
sy	0.022
id	99.925

Table C.3: average

variable	r
r	0.1523021
b	-0.0000391
w	0.0000000
swpd	0.0000000
free	0.3722860
buff	0.8795790
cache	0.9769591
si	0.0000000
so	0.0000000
bi	0.0359607
bo	0.0007226
in	0.0205215
cs	0.0052086
us	0.6653252
sy	0.0011794
id	0.0000000

Table C.4: Pearson-Bravai Correlation

variable	Spearman's rho
r	0.998
b	1.000
w	1.000
swpd	1.000
free	0.323
buff	0.410
cache	0.889
si	1.000
so	1.000
bi	0.996
bo	0.947
in	0.435
cs	0.971
us	0.996
sy	0.997
id	0.996

Table C.5: Spearman's rho

Appendix D

Statistical summary for pH-Load

This appendix contains the results from the fourth trial in the rst phase of this project. During this trial, the machines where running a webserver. We stressed the webserver three times during this trial. Some of the variables where zero throughout the trial, and produced therefore no plot. These empty plots have been removed from this appendix.

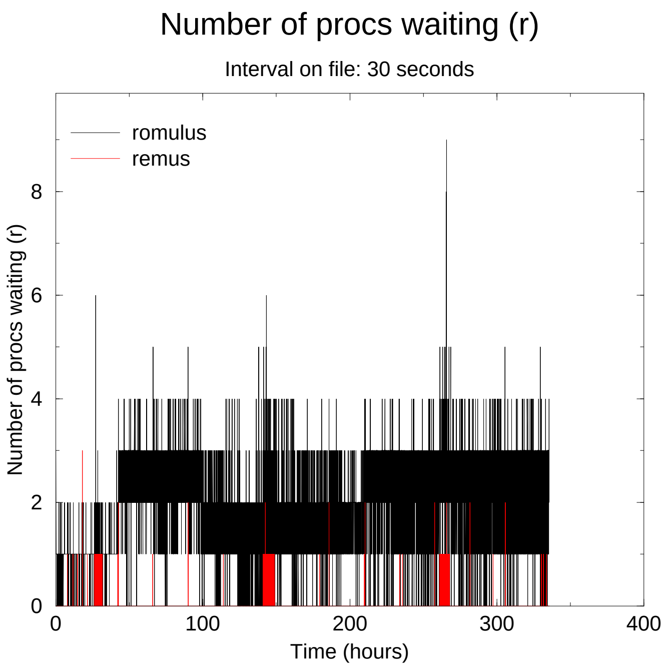


Figure D.1: This plot shows the number of precesses waiting. There is a lot more activity here than in the previous trials.

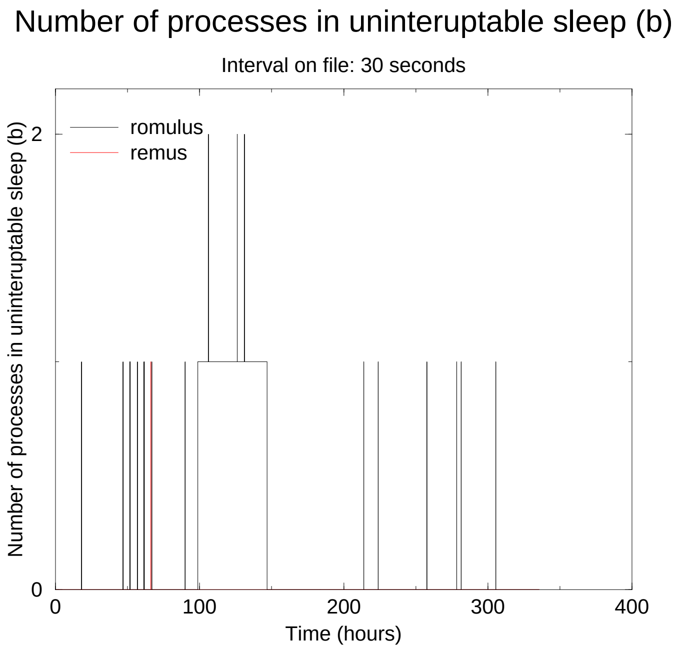


Figure D.2: This plot shows the number of processes in uninteruptable sleep. As we can see, there is more activity here.

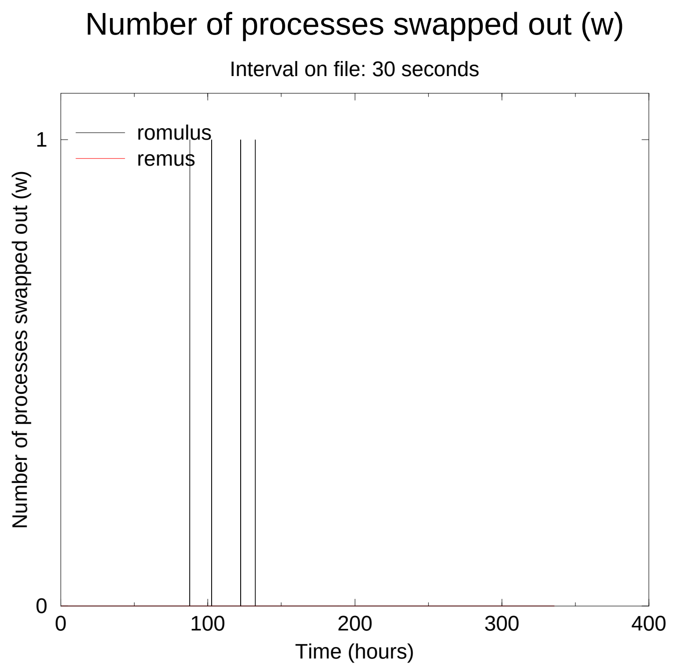


Figure D.3: Here we see the number of processes swapped out. As you can see, this only happened four times.

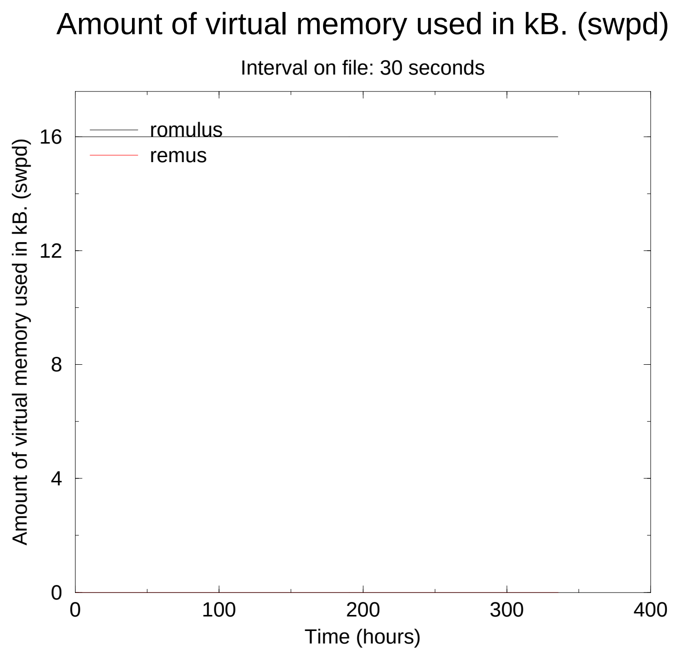


Figure D.4: This plot shows the amount of virtual memory used. Only romulus used virtual memory during this trial.

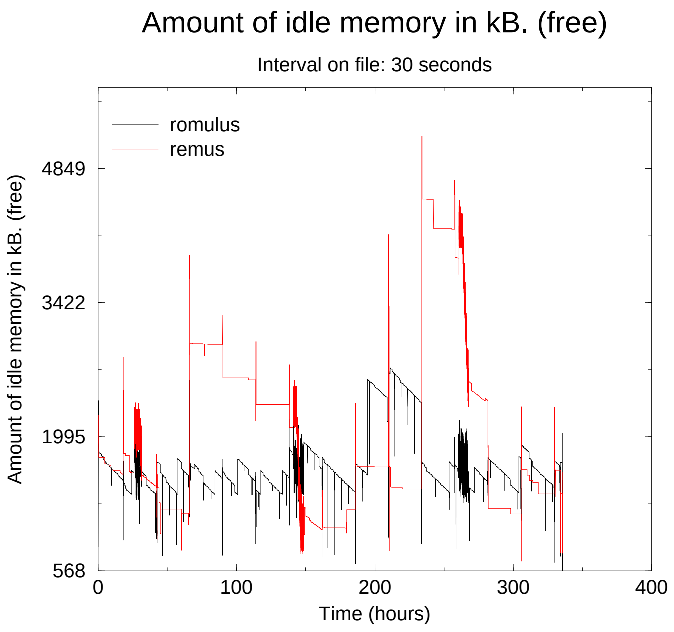


Figure D.5: Here we see the amount of idle memory in kB. The wavy pattern shows the effect of scheduled jobs every week. The pattern is much more distorted in this trial. In romulus, We see a greater difference between the systems. The three occasions when we stressed the webserver are clearly visible.

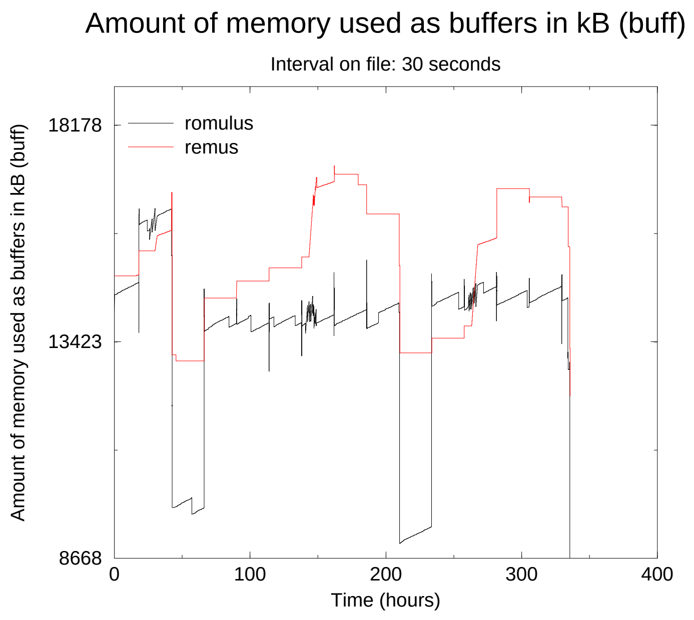


Figure D.6: This plot shows the amount of memory used as buffers. A greater difference than before here also.

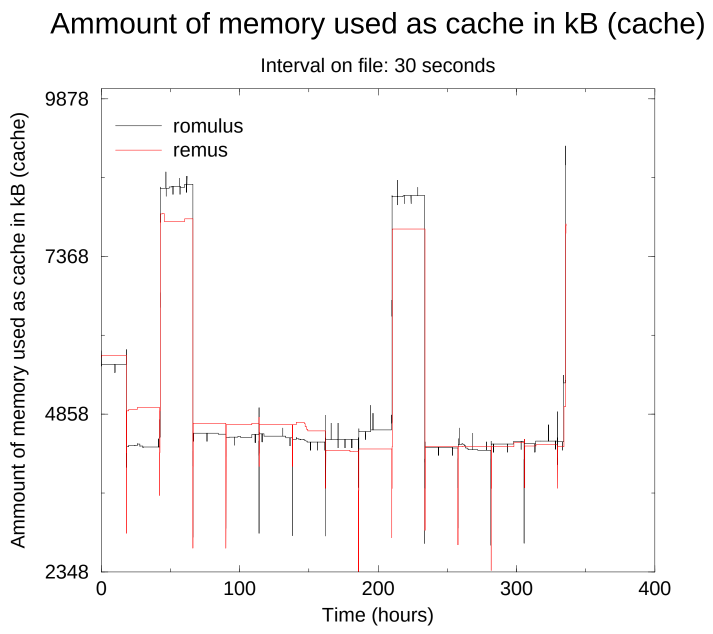


Figure D.7: In this plot we see the same pattern as in the other plots regarding memory usage. The difference is much smaller here than in the other plots from this trial.

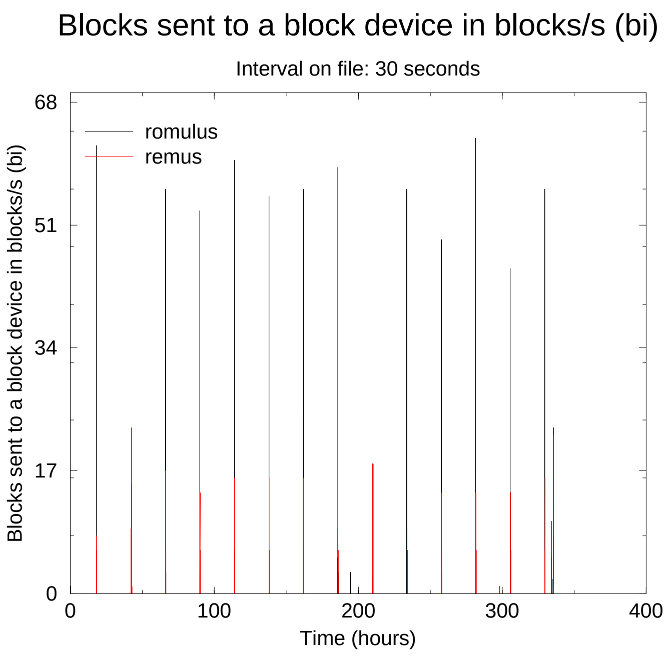


Figure D.8: This plot shows the amount of blocks sent to a block device. The plot seem more unorganised than in previous trials.

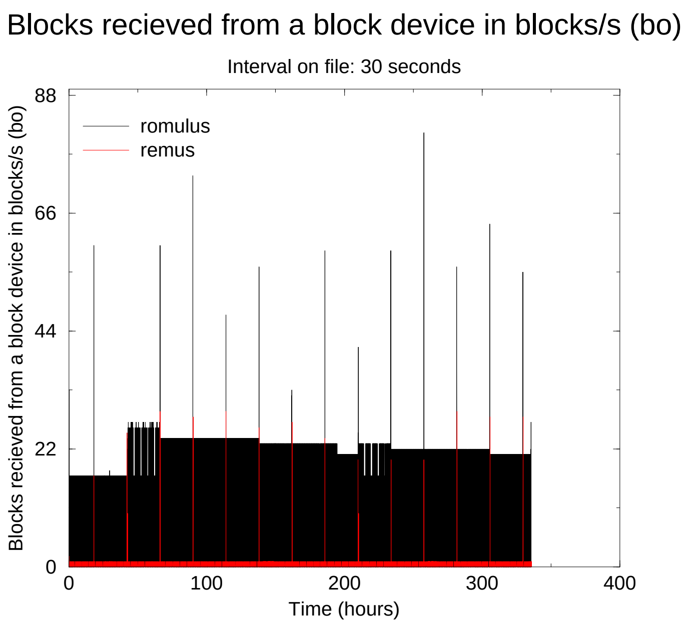
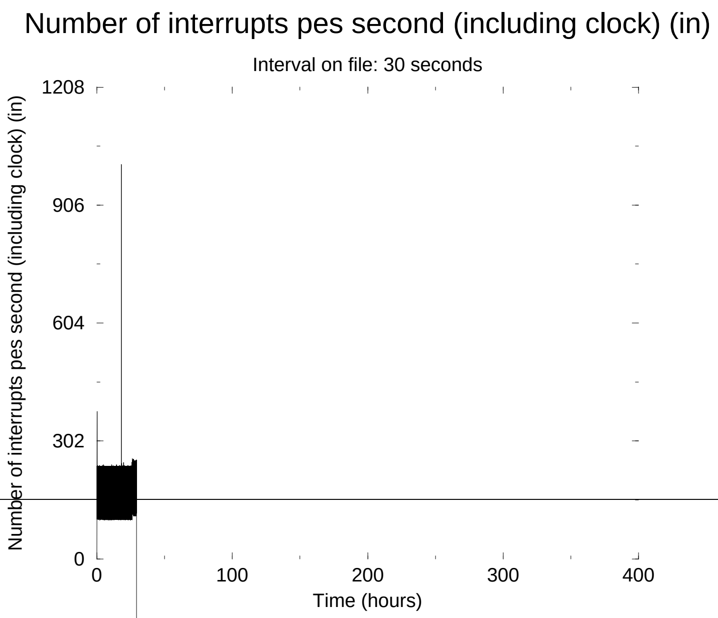
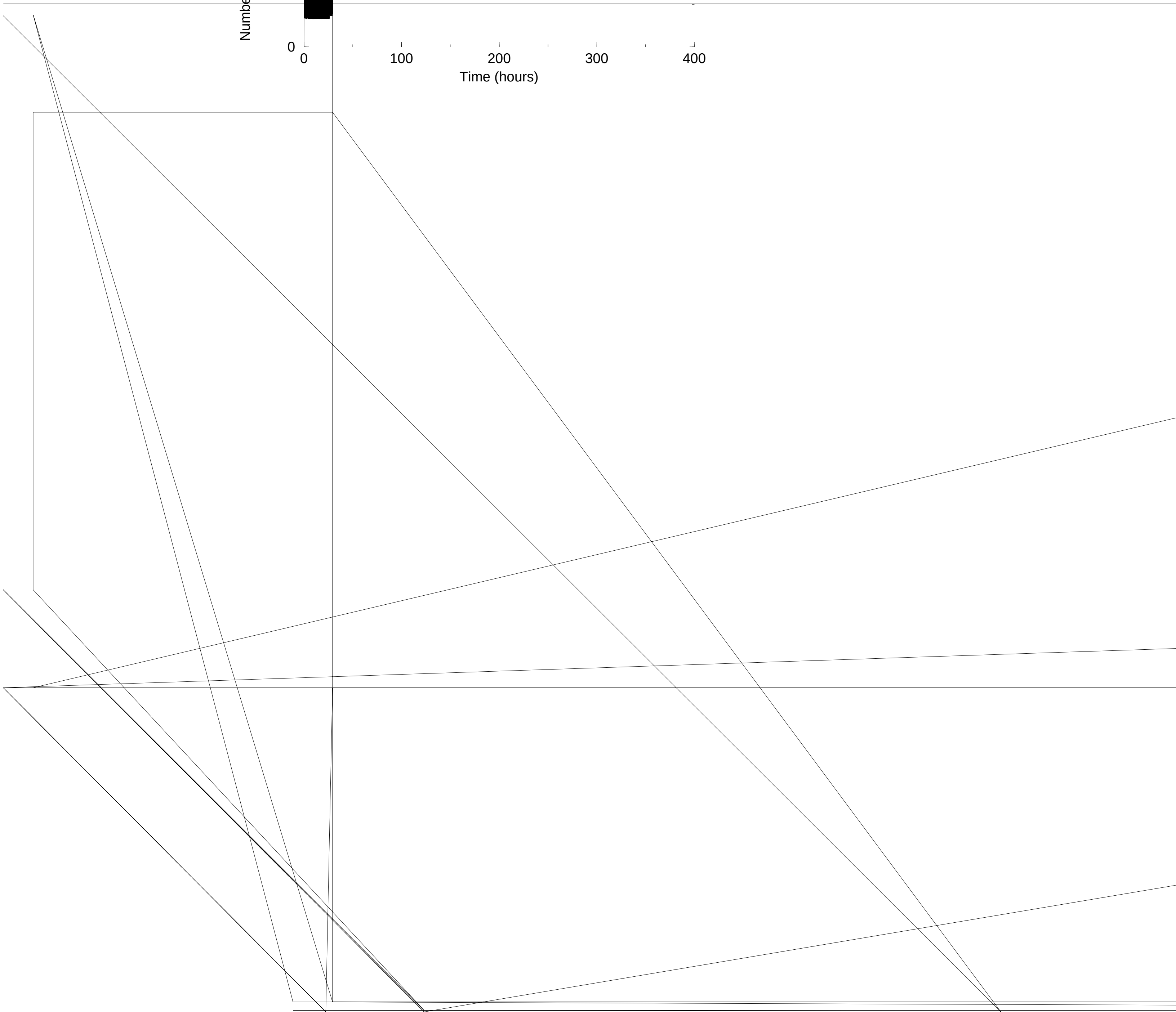


Figure D.9: In this plot we have greater dif culties to recognise the pattern from previous trials .


Number of interrupts pes second (including clock) (in)
Interval on file: 30 seconds
1208
906
604
302
0
0
100
200
300
400
Time (hours)
Number of interrupts pes second (including clock) (in)


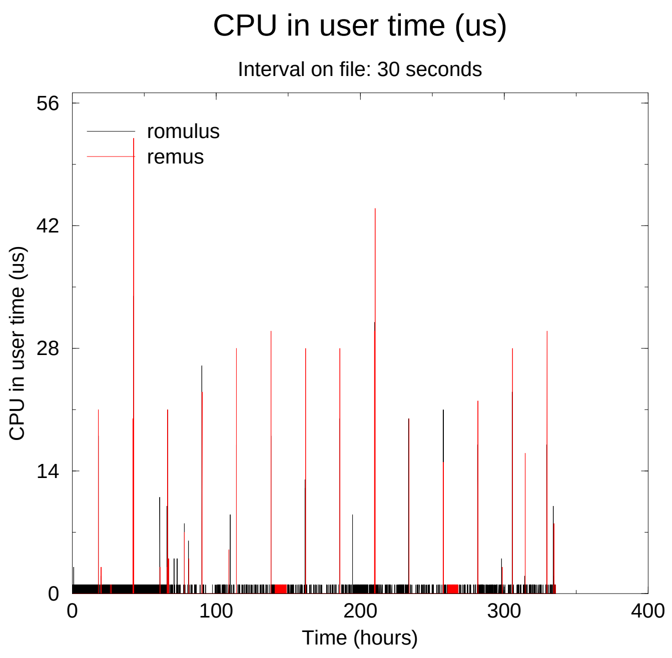
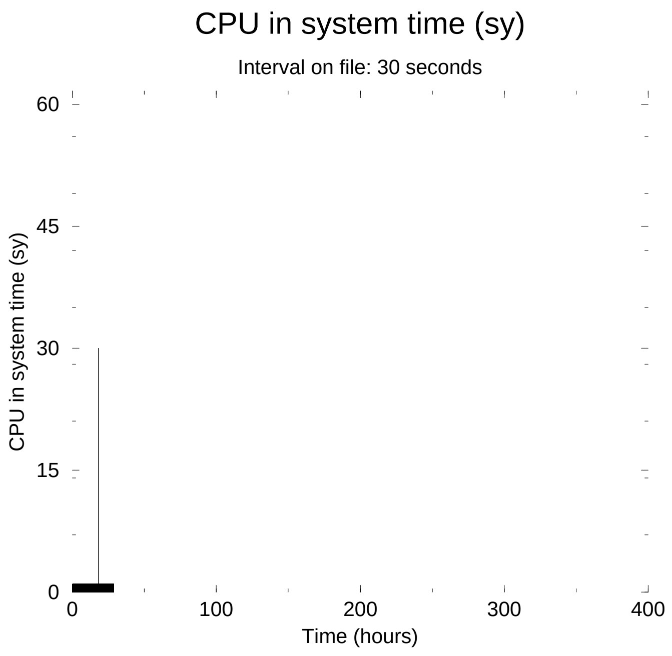
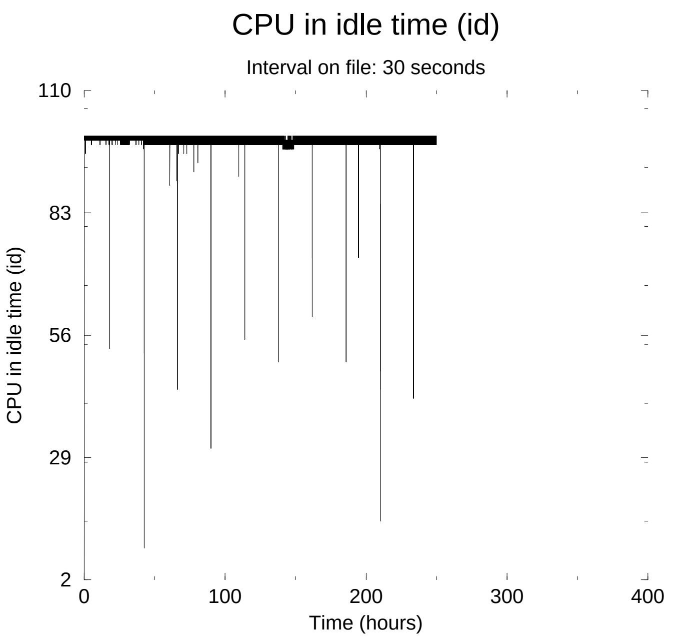


Figure D.12: CPU in user time. Note how both machines show more different behaviour. The peaks coincide less.




CPU in idle time (id)
Interval on file: 30 seconds
110
83
56
29
2
0
100
200
300
400
Time (hours)
CPU in idle time (id)

variable	n	sum	max	average	stdDev
r	40269	79614	9	1.977	0.796
b	40269	5770	2	0.143	0.351
w	40269	4	1	0.000	0.010
swpd	40269	644304	16	16.000	0.000
free	40269	68642440	2728	1704.598	322.098
buff	40269	549043496	16352	13634.396	1772.924
cache	40269	203872812	9128	5062.773	1382.830
si	40269	0	0	0.000	0.000
so	40269	0	0	0.000	0.000
bi	40269	1497	63	0.037	1.195
bo	40269	92009	81	2.285	6.763
in	40269	4901724	1099	121.725	56.137
cs	40269	13897100	519	345.107	69.879
us	40269	2798	34	0.069	0.581
sy	40269	5975	56	0.148	0.924
id	40269	4015408	100	99.715	1.582

Table D.1: vmstat.log.romulus.Fri_May_10_12-19-44_CEST_2002-Fri_May_24_12-26-38_CEST_2002

variable	n	sum	max	average	stdDev
r	40293	225	9	0.006	0.078
b	40293	1	2	0.000	0.005
w	40293	0	1	0.000	0.000
swpd	40293	0	16	0.000	0.000
free	40293	81186676	5192	2014.908	928.007
buff	40293	612531644	17292	15201.937	1323.788
cache	40293	203548560	9128	5051.710	1197.592
si	40293	0	0	0.000	0.000
so	40293	0	0	0.000	0.000
bi	40293	435	63	0.011	0.368
bo	40293	1179	81	0.029	0.509
in	40293	4169893	1099	103.489	8.672
cs	40293	181578	519	4.506	6.053
us	40293	946	52	0.023	0.639
sy	40293	609	56	0.015	0.608
id	40293	4026518	100	99.931	1.332

Table D.2: vmstat.log.remus.Fri_May_10_12-19-42_CEST_2002-Fri_May_24_12-25-55_CEST_2002

variable	average of both
r	0.991
b	0.072
w	0.000
swpd	8.000
free	1859.753
buff	14418.166
cache	5057.242
si	0.000
so	0.000
bi	0.024
bo	1.157
in	112.607
cs	174.807
us	0.046
sy	0.082
id	99.823

Table D.3: average

variable	r
r	-0.0130690
b	-0.0020368
w	0.0000000
swpd	0.0000000
free	-0.2097361
buff	0.6135067
cache	0.9787002
si	0.0000000
so	0.0000000
bi	0.0390587
bo	0.0134169
in	0.0715270
cs	0.0016713
us	0.0437398
sy	0.0377212
id	0.0000000

Table D.4: Pearson-Bravai Correlation

variable	Spearman's rho
r	0.550
b	0.816
w	1.000
swpd	1.000
free	-0.147
buff	0.426
cache	0.527
si	1.000
so	1.000
bi	0.995
bo	0.812
in	0.424
cs	0.466
us	0.921
sy	0.846
id	0.833

Table D.5: Spearman's rho

Appendix E

Statistical summary for `Default` (smoothed to 30 min)

This appendix contains the results from the rst trial in the rst phase of this project. The data has been smoothed by using local average for every 30 minutes. As a result, the plots look less jagged and the patterns due to the scheduled jobs are more visible.

Some of the variables where zero throughout the trial, and produced therefore no plot. These empty plots have been removed from this appendix.

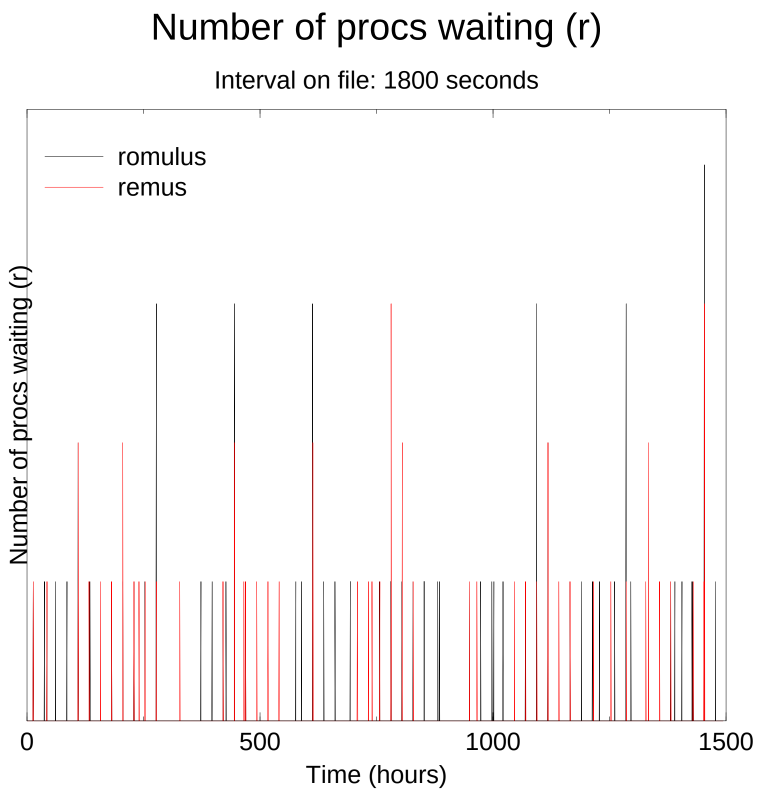


Figure E.1: This plot shows the number of precesses waiting.

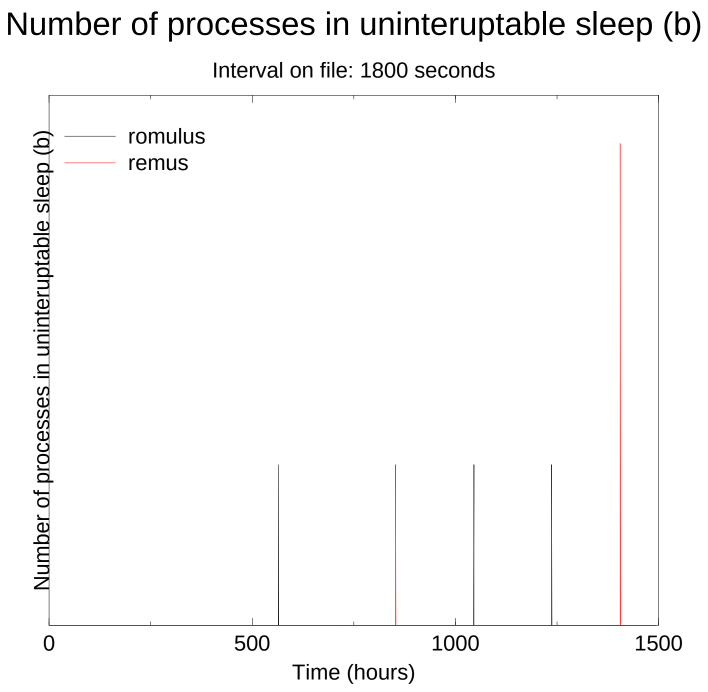


Figure E.2: This plot shows the number of processes in uninteruptable sleep. As we can see, there is not much activity in this variable.

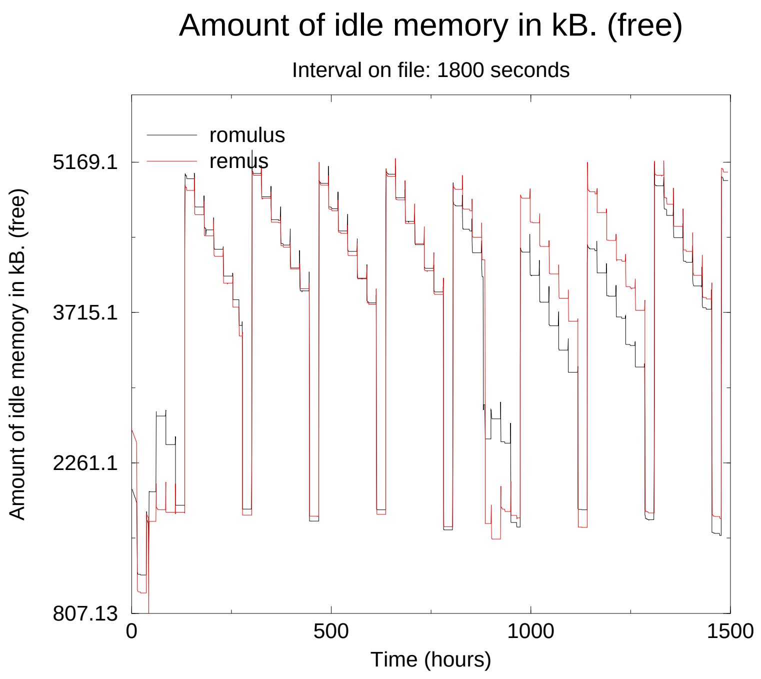


Figure E.3: Here we see the amount of idle memory in kB. The wavy pattern shows the effect of scheduled jobs every week.

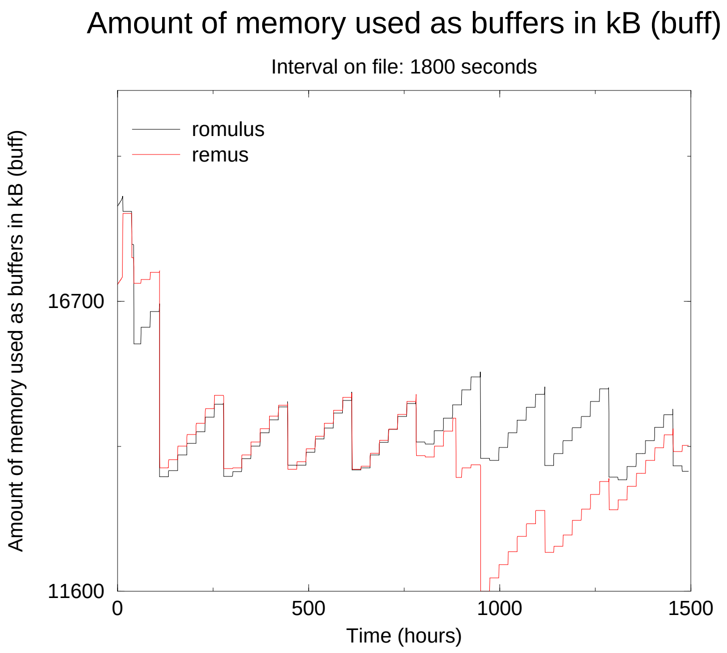


Figure E.4: This plot shows the amount of memory used as buffers. The same waves as in the previous plot are recognisable here.

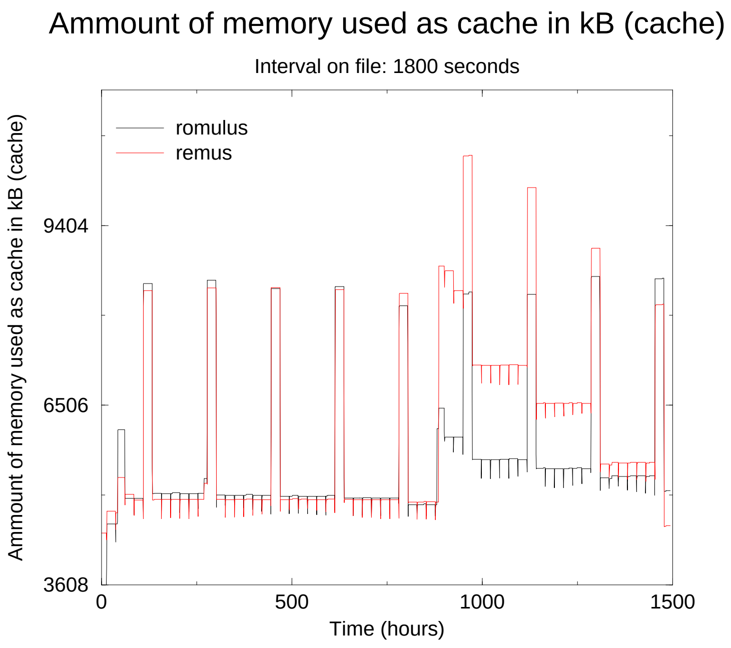


Figure E.5: In this plot we see the same pattern as in the other plots regarding memory usage.

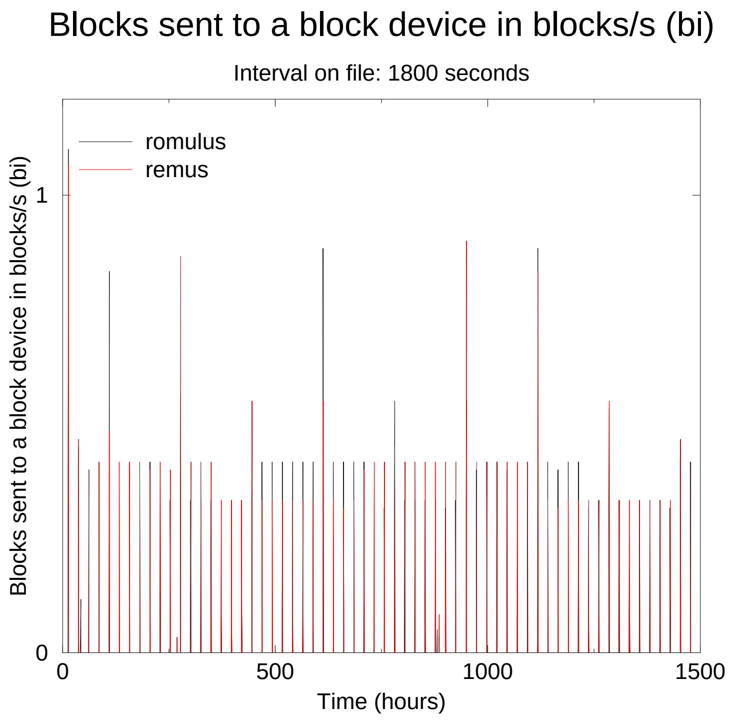


Figure E.6: This plot shows the amount of blocks sent to a block device. The peaks coincide, but it is hard to see any pattern.

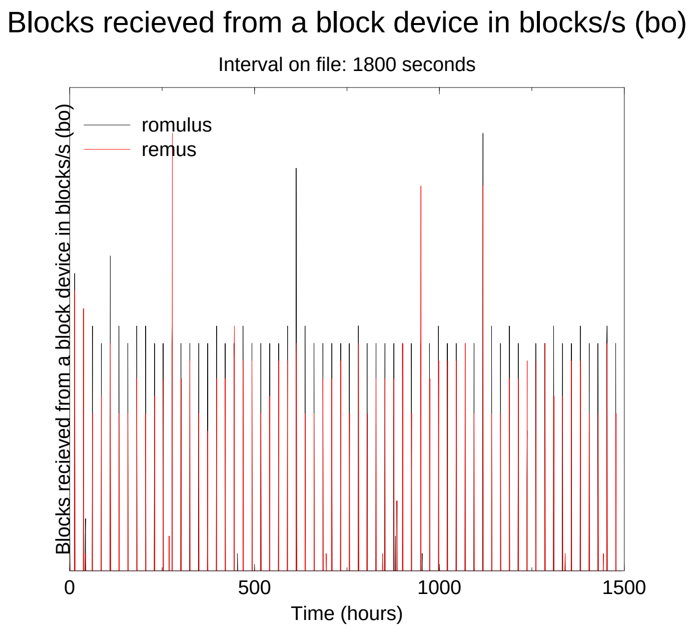
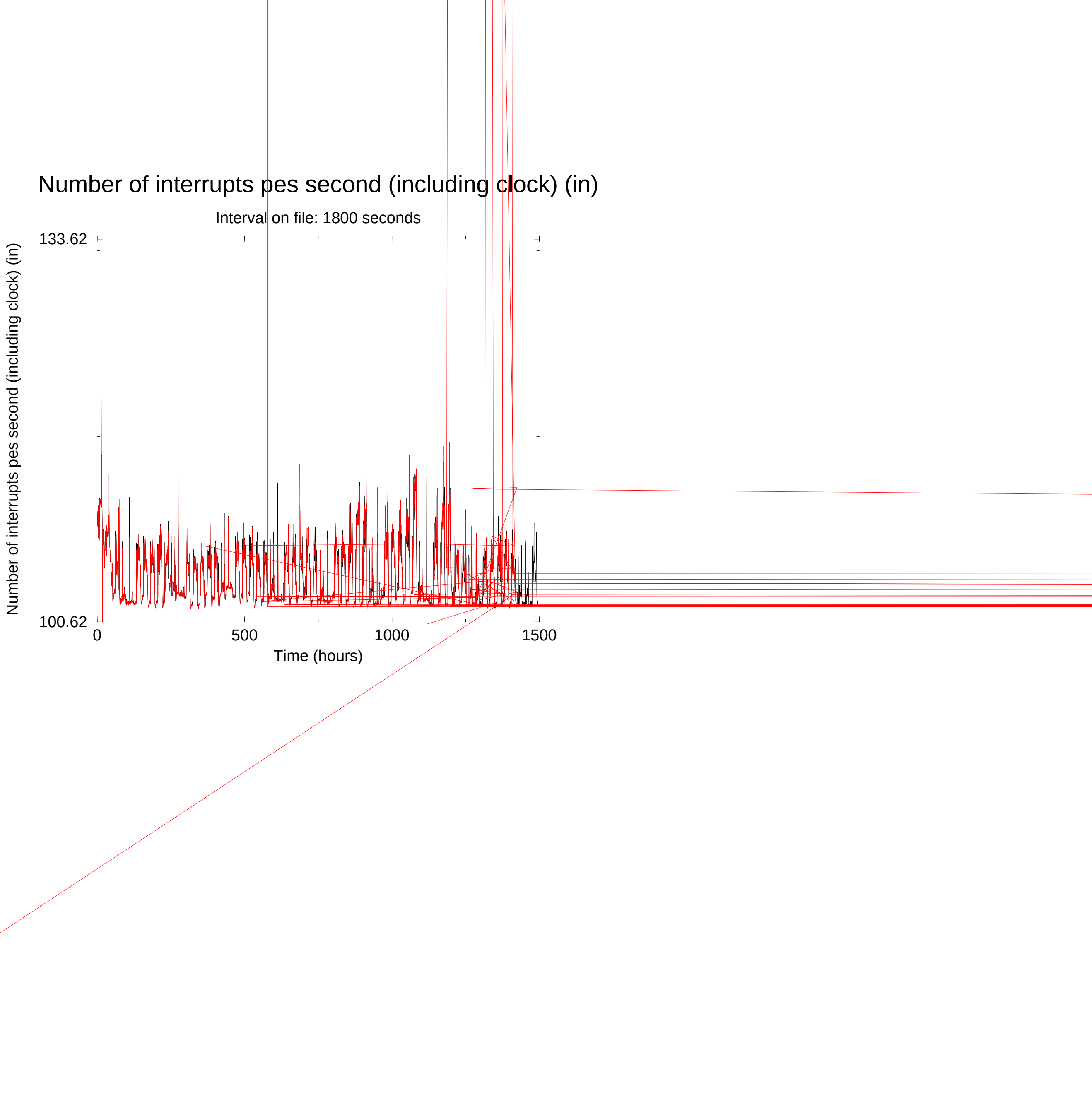


Figure E.7: In this plot it's hard recognise a weekly pattern, but the peaks seem to coincide.

Number of interrupts pes second (including clock) (in)

Interval on file: 1800 seconds



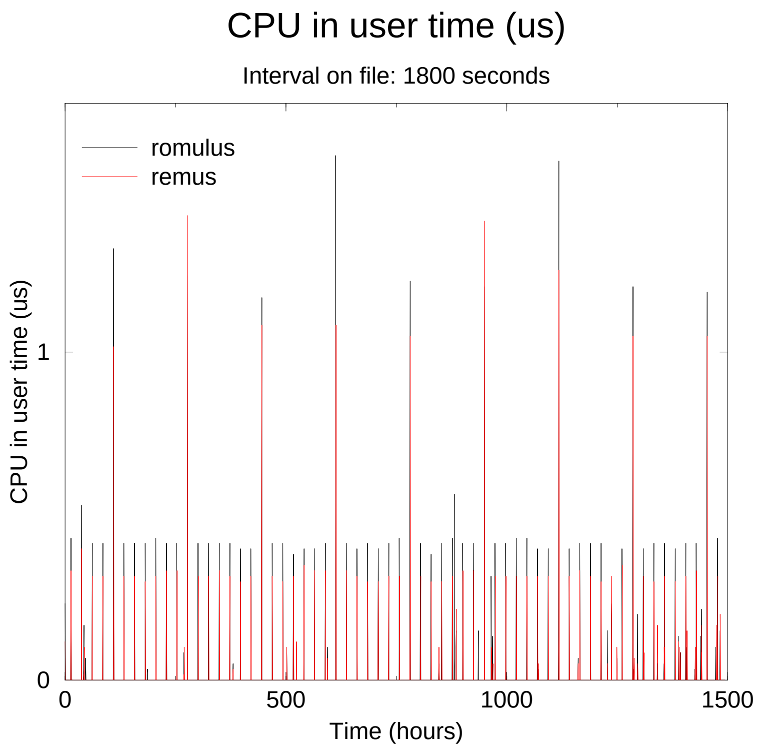


Figure E.10: CPU in user time. Note how both machines show similar behaviour. The peaks coincide here also.

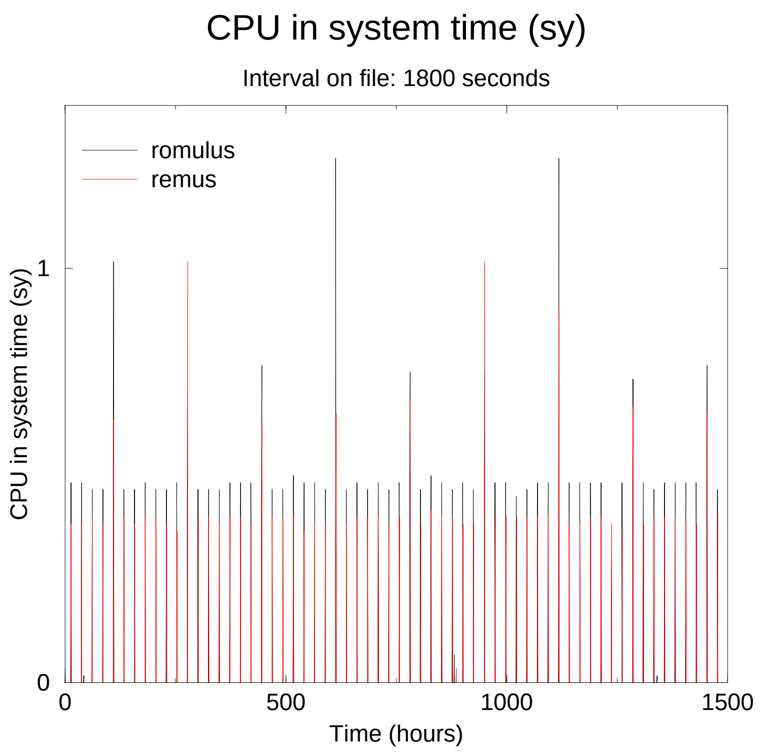


Figure E.11: CPU in system time. This plot is in many ways similar to the previous one in that it shows a different perspective on the same resource - CPU usage. A wavy pattern is hardly recognisable.

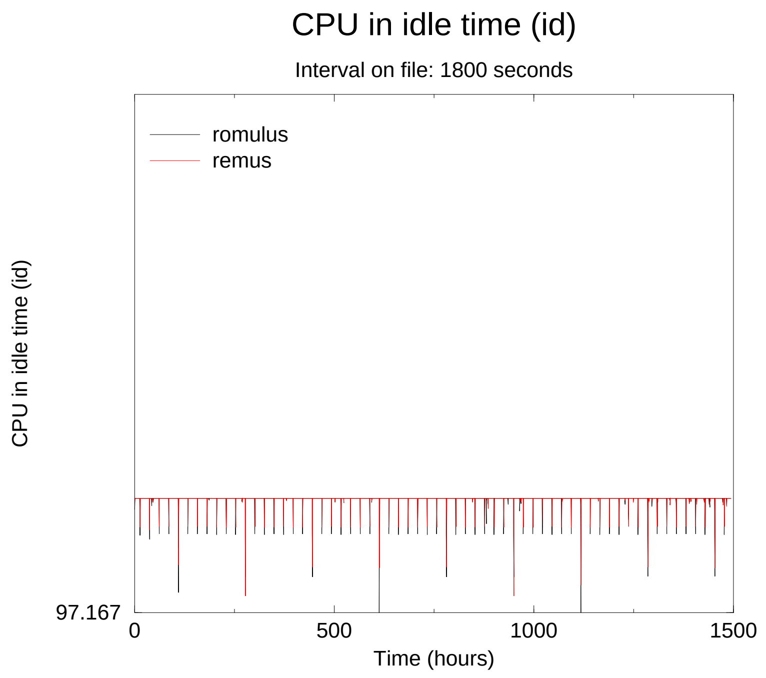


Figure E.12: This plot, as with the two previous ones, shows a periodic, stable and constant behaviour on both machines.

variable	n	sum	max	average	stdDev
r	2987	1	0	0.000	0.003
b	2987	0	0	0.000	0.001
w	2987	0	0	0.000	0.000
swpd	2987	0	0	0.000	0.000
free	2987	10933906	5289	3660.498	1116.362
buff	2987	43342498	18545	14510.378	881.701
cache	2987	16966811	8584	5680.218	1159.402
si	2987	0	0	0.000	0.000
so	2987	0	0	0.000	0.000
bi	2987	30	1	0.010	0.067
bo	2987	65	0	0.022	0.034
in	2987	311833	121	104.397	2.460
cs	2987	3071	5	1.028	0.238
us	2987	40	1	0.014	0.093
sy	2987	36	1	0.012	0.083
id	2987	298622	100	99.974	0.174

Table E.1: vmstat_phase1.romulus.smoothe1800

variable	n	sum	max	average	stdDev
r	2988	1	0	0.000	0.003
b	2988	0	0	0.000	0.001
w	2988	0	0	0.000	0.000
swpd	2988	0	0	0.000	0.000
free	2988	11089195	5289	3711.243	1236.121
buff	2988	41946799	18545	14038.420	1235.337
cache	2988	18192895	10540	6088.653	1500.546
si	2988	0	0	0.000	0.000
so	2988	0	0	0.000	0.000
bi	2988	30	1	0.010	0.066
bo	2988	63	0	0.021	0.030
in	2988	311808	121	104.353	2.440
cs	2988	3068	5	1.027	0.231
us	2988	32	1	0.011	0.078
sy	2988	30	1	0.010	0.069
id	2988	298737	100	99.979	0.145

Table E.2: vmstat_phase1.remus.smoothe1800

variable	average of both
r	0.000
b	0.000
w	0.000
swpd	0.000
free	3685.871
buff	14274.399
cache	5884.436
si	0.000
so	0.000
bi	0.010
bo	0.022
in	104.375
cs	1.028
us	0.012
sy	0.011
id	99.977

Table E.3: average

variable	r
r	0.2975995
b	-0.0007340
w	0.0000000
swpd	0.0000000
free	0.9558609
buff	0.7199619
cache	0.8594925
si	0.0000000
so	0.0000000
bi	0.6123918
bo	0.5233057
in	0.9164983
cs	0.4691590
us	0.4940068
sy	0.5262725
id	0.0000000

Table E.4: Pearson-Bravai Correlation

variable	Spearman's rho
r	1.000
b	1.000
w	1.000
swpd	1.000
free	0.937
buff	0.462
cache	0.916
si	1.000
so	1.000
bi	1.000
bo	1.000
in	0.914
cs	0.993
us	0.994
sy	0.997
id	0.947

Table E.5: Spearman's rho

Appendix F

Installed software on `romulus` and `remus`

The following list is the result of the following command:

```
dpkg -l \* | grep ''^ii''
```

```
ii  adduser        3.11.1         Add users and groups to the system.
ii  ae             962-26         Anthony's Editor -- a tiny full-screen edito
ii  apache         1.3.9-14       Versatile, high-performance HTTP server
ii  apache-common  1.3.9-14       Support files for all Apache webservers
ii  apt            0.3.19         Advanced front-end for dpkg
ii  at             3.1.8-10       Delayed job execution and batch processing
ii  base-config    0.33.2         Debian base configuration package
ii  base-files     2.2.0          Debian base system miscellaneous files
ii  base-passwd    3.1.10         Debian Base System Password/Group Files
ii  bash           2.03-6         The GNU Bourne Again SHell
ii  bc             1.05a-11       The GNU bc arbitrary precision calculator la
ii  bin86          0.14.9-3       16-bit assembler and loader
ii  binutils       2.9.5.0.37-1   The GNU assembler, linker and binary utiliti
ii  bsdmainutils   4.7.1          More utilities from 4.4BSD-Lite.
ii  bsdutils       2.10f-5.1      Basic utilities from 4.4BSD-Lite.
ii  bzip2          0.9.5d-2       A high-quality block-sorting file compressor
ii  console-data   1999.08.29-11. Keymaps, fonts, charset maps, fallback table
ii  console-tools  0.2.3-10.3     Linux console and font utilities.
ii  console-tools- 0.2.3-10.3     Shared libraries for Linux console and font
ii  cpio           2.4.2-32       GNU cpio -- a program to manage archives of
ii  cpp            2.95.2-13      The GNU C preprocessor.
ii  cron           3.0pl1-57.2    management of regular background processing
ii  dc             1.05a-11       The GNU dc arbitrary precision reverse-polis
ii  debconf-tiny   0.2.80.17      Tiny subset of debconf for the base system
ii  debianutils    1.13.3         Miscellaneous utilities specific to Debian.
ii  diff           2.7-21         File comparison utilities
ii  dpkg           1.6.15         Package maintenance system for Debian
ii  dpkg-dev       1.6.15         Package building tools for Debian
ii  e2fsprogs      1.18-3.0potato The EXT2 file system utilities and libraries
ii  ed             0.2-18.1       The classic unix line editor
ii  elvis-tiny     1.4-11         Tiny vi compatible editor for the base syste
ii  exim           3.12-10        Exim Mailer
ii  fbset          2.1-6          Framebuffer device maintenance program.
ii  fdflush        1.0.1-5        A disk-flushing program.
ii  fdutils        5.3-3          Linux floppy utilities
ii  fileutils      4.0l-8         GNU file management utilities.
ii  findutils      4.1-40         utilities for finding files--find, xargs, an
ii  ftp            0.10-3.1       The FTP client.
ii  gcc            2.95.2-13      The GNU C compiler.
ii  gettext-base   0.10.35-13     GNU Internationalization utilities for the b
ii  grep           2.4.2-1        GNU grep, egrep and fgrep.
ii  groff          1.15.2-1       GNU troff text-formatting system.
ii  gzip           1.2.4-33       The GNU compression utility.
ii  hostname       2.07           A utility to set/show the host name or domai
ii  info           4.0-4          Standalone GNU Info documentation browser
ii  isapnptools    1.21-2         ISA Plug-And-Play configuration utilities.
ii  jed            0.99.9-14      Editor for programmers. (textmode version)
ii  jed-common     0.99.9-14      Byte compiled SLang runtime files for jed an
ii  kernel-image-2 Custom.1.00    Linux kernel binary image for version 2.2.19
ii  kernel-package 7.04.potato.3  Debian Linux kernel package build scripts.
ii  kernel-source- 2.2.19.1-2     Linux kernel source for version 2.2.19
ii  ldso           1.9.11-9       The Linux dynamic linker, library and utilit
ii  libbz2         0.9.5d-2       A high-quality block-sorting file compressor
ii  libc6          2.1.3-19       GNU C Library: Shared libraries and Timezone
ii  libc6-dev      2.1.3-19       GNU C Library: Development Libraries and Hea
ii  libdb2         2.4.14-2.7.7.1 The Berkeley database routines (run-time fil
ii  libgdbmg1      1.7.3-26.2     GNU dbm database routines (runtime version).
ii  libgpmg1       1.17.8-18      General Purpose Mouse Library [libc6]
ii  libident       0.22-2         simple RFC1413 client library - runtime
ii  liblockfile1   1.01           Shared library with NFS-safe locking functio
ii  libncurses4    4.2-9          Shared libraries for terminal handling
ii  libncurses5    5.0-6.0potato1 Shared libraries for terminal handling
ii  libncurses5-de 5.0-6.0potato1 Developer’s libraries and docs for ncurses
ii  libnewt0       0.50-7         Not Erik’s Windowing Toolkit - text mode win
ii  libopenldap-ru 1.2.11-1       OpenLDAP runtime files for libopenldap
ii  libopenldap1   1.2.11-1       OpenLDAP libraries.
ii  libpam-modules 0.72-9         Pluggable Authentication Modules for PAM
ii  libpam-runtime 0.72-9         Runtime support for the PAM library
ii  libpam0g       0.72-9         Pluggable Authentication Modules library
ii  libpcre2       2.08-1         Philip Hazel’s Perl Compatible Regular Expre
ii  libpopt0       1.4-1.1        lib for parsing cmdline parameters
ii  libreadline4   4.1-1          GNU readline and history libraries, run-time
ii  libssl09       0.9.4-5        SSL shared libraries
ii  libstdc++2.10  2.95.2-13      The GNU stdc++ library
ii  libwrap0       7.6-4          Wietse Venema’s TCP wrappers library
ii  lilo           21.4.3-2       LInux LOader - The Classic OS loader can loa
ii  locales        2.1.3-19       GNU C Library: National Language (locale) da
ii  lockfile-progs 0.1.7          Programs for locking and unlocking files and
ii  login          19990827-20    System login tools
ii  logrotate      3.2-11         Log rotation utility
ii  mailx          8.1.1-11       A simple mail user agent.
ii  make           3.79.1-1.potat The GNU version of the "make" utility.
ii  makedev        2.3.1-46.2     Creates special device files in /dev.
ii  man-db         2.3.16-1.1     Display the on-line manual.
ii  manpages       1.29-2         Man pages about using a Linux system.
ii  mawk           1.3.3-5        a pattern scanning and text processing langu
ii  mbr            1.1.2-1        Master Boot Record for IBM-PC compatible com
ii  mime-support   3.9-1          MIME files ’mime.types’ & ’mailcap’, and sup
ii  modconf        0.2.26.14      Device Driver Configuration
ii  modutils       2.3.11-13.1    Linux module utilities.
ii  mount          2.10f-5.1      Tools for mounting and manipulating filesyst
ii  ncftp          3.0beta21-1    A user-friendly and full-featured FTP client
ii  ncurses-base   5.0-6.0potato1 Descriptions of common terminal types
ii  ncurses-bin    5.0-6.0potato1 Terminal-related programs and man pages
ii  netbase        3.18-4         Basic TCP/IP networking binaries
ii  ntpdate        4.0.99g-2      The ntpdate client for setting system time f
ii  nvi            1.79-16a       4.4BSD re-implementation of vi.
ii  passwd         19990827-20    Change and administer password and group dat
ii  patch          2.5-2.2        Apply a diff file to an original
ii  pciutils       2.1.2-2        Linux PCI Utilities (for 2.[123].x kernels)
ii  perl-5.005     5.005.03-7.1   Larry Wall’s Practical Extracting and Report
ii  perl-5.005-bas 5.005.03-7.1   The Pathologically Eclectic Rubbish Lister
ii  perl-base      5.004.05-1.1   Fake package assuring that one of the -base
ii  ppp            2.3.11-1.4     Point-to-Point Protocol (PPP) daemon.
ii  pppconfig      2.0.5          A text menu based utility for configuring pp
ii  procps         2.0.6-5        The /proc file system utilities.
ii  psmisc         19-2           Utilities that use the proc filesystem
ii  pump           0.7.3-2        Simple DHCP/BOOTP client for 2.2.x kernels
ii  screen         3.9.5-9        A screen manager with VT100/ANSI terminal em
ii  sed            3.02-5         The GNU sed stream editor.
ii  setserial      2.17-16        Controls configuration of serial ports.
ii  shellutils     2.0-7          The GNU shell programming utilities.
ii  slang1         1.3.9-1        The S-Lang programming library - runtime ver
ii  ssh            1.2.3-9.3      Secure rlogin/rsh/rcp replacement (OpenSSH)
ii  sysklogd       1.3-33.1       Kernel and system logging daemons
ii  syslinux       1.48-2         Bootloader for Linux/i386 using MS-DOS flopp
ii  sysvinit       2.78-4         System-V like init.
ii  tar            1.13.17-2      GNU tar
ii  tasksel        1.0-10         New task packages selector
ii  tcpd           7.6-4          Wietse Venema’s TCP wrapper utilities
ii  telnet         0.16-4potato.1 The telnet client.
ii  textutils      2.0-2          The GNU text file processing utilities.
ii  update         2.11-1         daemon to periodically flush filesystem buff
ii  util-linux     2.10f-5.1      Miscellaneous system utilities.
ii  wget           1.5.3-3        utility to retrieve files from the WWW via H
ii  whiptail       0.50-7         Displays user-friendly dialog boxes from she
ii  whois          4.4.14         whois client
ii  xviddetect     0.3-4          XFree86 installation helper
ii  zlib1g         1.1.3-5        compression library - runtime
```

Appendix G

A scaled, immunological approach to anomaly countermeasures