Enhancing Intrusion Resilience in Publicly Accessible Distributed Systems

Dylan James Clarke

In Partial Fulfilment of the Requirements for the Degree of Doctor of Philosophy

School of Computing Science

Newcastle University

July 2012

Acknowledgements

I would like to thank everyone who has offered me support and advice during my PhD, especially my supervisor Dr. Paul Ezhilchelvan, Professor Isi Mitrani and my thesis committee Professor Santosh Shrivastava and Dr. Nigel Thomas.

Abstract

The internet is increasingly used as a means of communication by many businesses. Online shopping has become an important commercial activity and many governmental bodies offer services online. Malicious intrusion into these systems can have major negative consequences, both for the providers and users of these services.

The need to protect against malicious intrusion, coupled with the difficulty of identifying and removing all possible vulnerabilities in a distributed system, have led to the use of systems that can tolerate intrusions with no loss of integrity. These systems require that services be replicated as deterministic state machines, a relatively hard task in practice, and do not ensure that confidentiality is maintained when one or more replicas are successfully intruded into.

This thesis presents FORTRESS, a novel intrusion-resilient system that makes use of proactive obfuscation techniques and cheap off-the-shelf hardware to enhance intrusion-resilience. FORTRESS uses proxies to prevent clients accessing servers directly, and regular replacement of proxies and servers with differently obfuscated versions. This maintains both confidentiality and integrity as long as an attacker does not compromise the system as a whole.

The expected lifetime until system compromise of the FORTRESS system is compared to those of state machine replicated and primary backup systems when confronted with an attacker capable of launching distributed attacks against known vulnerabilities. Thus, FORTRESS is demonstrated to be a viable alternative to building intrusion-tolerant systems using deterministic state machine replication.

The performance overhead of the FORTRESS system is also evaluated, using both a general state transfer framework for distributed systems, and a lightweight framework for large scale web applications. This shows the FORTRESS system has a sufficiently small performance overhead to be of practical use.

Contents

List of Figures

List of Tables

Glossary of Notation and Abbreviations

SMR	A replicated system using state machine replication for intrusion tolerance.
PB	A replicated system using primary-backup replication for fault tolerance.
SO	Start-up only Obfuscation
PR	Proactive Recovery
PO	Proactive Obfuscation
EL	Expected Lifetime until System Compromise

Chapter 1

Introduction

Distributed systems have become an important part of many businesses, with the Internet being used as a primary means of communication. Online shopping is common both as an additional service offered by traditional businesses, and as a sole means of trading. Governmental bodies increasingly offer services online, both to the public and to other parts of the government, ranging from web forms allowing litter to be reported, through to systems allowing authorised staff to access sensitive personal information. Many organisations even access their own internal records through a distributed application, allowing mobile and off-site working.

This proliferation of distributed systems has resulted in a need for increased security. Malicious intrusion into one of these systems can have huge negative consequences for the organisation involved. An attacker who successfully compromises an online shopping site may be able to defraud the business of large amounts of merchandise, but this is likely to be one of the smaller concerns of the business owners. More seriously, an attacker can significantly impair the capability of the business to sell products. This has the potential to ultimately result in the failure of the business.

For example, an attacker who corrupts the contents of an online shopping site may cost the business a significant number of sales while the site is unavailable, destroy or alter records of orders in progress and steal the personal information of customers, thereby damaging the reputation of the business. Even more seriously, an intruder into a governmental system may be able to steal or alter health or criminal record information, potentially resulting in breaches of privacy, the commission of further crimes, or even the death of the subjects of the records if medical information is changed, or records of convictions for serious crimes are made public.

This need for increased security is amplified by the fact that the internet is used to access these systems. The nature of Internet communication means that the systems are accessed via TCP/IP and packets of data may take arbitrary routes between sender and receiver. This, coupled with the incredibly large numbers of machines connected to the Internet, and the public availability of Internet connections, means

that anyone who wishes to can attempt to attack these systems, from any location they choose.

The attacks that may be launched against a distributed system can be grouped into three categories based on the attributes of a dependable system they attempt to compromise. We note that originally six attributes of a dependable system were identified in [17], but only three of these are likely targets for malicious attackers. The first of these categories is made up of attacks that attempt to compromise the *integrity* of a system. These attacks aim to alter existing system data or code in such a way that the system no longer performs as intended, either due to functioning incorrectly, or functioning correctly with incorrect data. The second category consists of attacks that attempt to compromise the *confidentiality* of a system. These attacks aim to get unauthorised access to data.

Finally, the third category consists of attacks that attempt to compromise the *availability* of a system. Here an attacker will try and prevent the system being available for use. This can either be through an attack on system integrity that aims to be so damaging that the system is no longer usable, or by attempting to generate a sufficient number of malicious requests that the system is unable to handle any other requests that are sent to it. The latter is known as a *denial of service attack*.

We note here that the attacks against integrity and confidentiality involve gaining some sort of unauthorised control over the system being attacked, to change or read some aspect of the system state. Attacks against availability do not have to have this characteristic, and can purely be caused by overloading the system with what are otherwise valid requests.

This distinction results in a corresponding distinction in how these attacks can be dealt with. Attacks against availability can be mitigated against by increasing the amount of processing power available until all of the requests can be handled, by filtering out malicious requests, or by identifying and blocking the attacker. Once this has been accomplished, the system will continue to perform as before.

Attacks against integrity or confidentiality, in contrast, are not mitigated against by detecting and stopping them after they succeed. Once an attack is halted, any corrupted system state is still corrupted, and any stolen information has been stolen, and in the case of sensitive personal information may now be public knowledge.

This leads to a different strategy having to be used in the case of attacks against integrity and confidentiality; prevention rather than mitigation once an attack has succeeded. However, this is complicated by the extreme difficulty of making sure that software is free of vulnerabilities that may be exploited, and the sheer number of potential attackers that a distributed system on the Internet may be exposed to. This results in a need for *intrusion tolerance*; that is, a system that can withstand the situation where a vulnerability is exploited and some part of the system is maliciously

intruded into, without integrity or confidentiality being compromised.

One key issue with intrusion tolerance is that, as the system needs to be tolerant of intrusions making use of arbitrary and unknown vulnerabilities, the general assumption is that intrusion involves the attacker managing to take complete control of the machine intruded into. This includes access to any data held on the machine, and the ability to do anything that the machine was capable of doing, including altering any system data. Hence, intrusion tolerant systems require *agreement* among a number of diverse machines before any action can be taken, allowing them to survive some finite number of malicious intrusions.

The number of messages exchanged in the protocols used to make these decisions grows rapidly as the number of machines increases, so there is a need to keep the number of machines reasonably small. This is compounded by the cost and difficulty of writing many diverse versions of the same software, and acquiring sufficient diverse operating systems and hardware on which to run them.

The need for agreement also results in a need for each of these diverse machines to produce identical states when given a set of identically ordered inputs. This leads to a requirement to either remove, or handle the outcomes of all sources of non-determinism in the system.

1.1 Our Approach

The availability of cheap off-the-shelf hardware has made hardware costs a relatively small part of IT system costs in general. One way of exploiting this trend to aid in preserving integrity and confidentiality is to use many in place of one and make it harder for an attacker to intrude more than a threshold.

Obviously, the many replicas should be non-identical; otherwise the attacker would succeed using the same strategy for all, once that strategy is worked out. Randomisation techniques such as those detailed in sections 2.8.3 and 2.8.4 help to generate diverse replicas with minimal effort. Even then, an attacker, given time, could work out appropriate strategies to intrude more than the threshold. To prevent this, randomisation ought to be changed regularly for diversity to be replenished.

This technique is termed as proactive obfuscation [45].

There are however two barriers to the use of large amounts of proactively obfuscated hardware to produce systems with a high degree of intrusion resilience. Firstly, the number of messages exchanged between replicas will become prohibitively high as the number of replicas increases. Secondly, the comparison of states between a large number of differently randomised systems may result in a significant overhead in marshalling and unmarshalling system states.

We attempt to circumvent these issues, and other practical issues around the use of replication strategies for intrusion tolerance, by instead considering intrusion resilience schemes that separate machines into two types of node, tolerating intrusions in one type, while protecting the other type from direct attack. This, coupled with the use of proactive obfuscation and additional cheap off-the-shelf hardware will be used to design a protocol that decreases the likelihood of a system being maliciously intruded into to the point that confidentiality or integrity are compromised.

1.2 Thesis Objectives

The objectives of this thesis are:

- To design an intrusion resilience protocol that can leverage proactive obfuscation techniques and the availability of cheap off-the-shelf hardware. The protocol should be able to be used to augment existing systems without a need to significantly re-write these systems to remove determinism. The protocol should also be usable in addition to any existing crash or intrusion tolerance measures.
- To statistically evaluate the intrusion resilience of systems using this protocol across a variety of likely conditions, and compare it to the existing state-of -art crash and intrusion tolerant protocols. This will allow us to
  - (i) identify situations where this protocol is the best choice for intrusion resilience,
  - (ii) identify the trade-offs present in other situations where there is no overall best protocol
  - (iii) identify any situations where this protocol would not be a viable choice.
- To develop an implementation framework for this protocol which will be as lightweight as possible in terms of overhead and development time required to use it with a particular system. It will be designed to allow customisation to any given legacy system.
- To measure the efficiency overhead of this protocol both in relatively small scale distributed applications and large scale web applications.

These objectives can be summarised in the following thesis statement.

1.2.1 Thesis Statement

This thesis designs an intrusion resilience protocol utilising proactive obfuscation techniques to augment intrusion resilience in existing systems and demonstrates the applicability of this protocol in a variety of situations through assessing the degree of

intrusion resilience and determining the efficiency overhead using an implementation framework and two specific implementations of test systems.

1.3 Thesis Contribution

The contributions made by this thesis are 4-fold.

The first contribution is to design a novel technique for adding intrusion resilience to distributed systems. This technique, proactive fortification, which implements proactive obfuscation, can be used to augment the intrusion resilience of systems without the necessity to remove all sources of non-determinism. It can also be added to systems that already make use of another replication strategy to provide fault tolerance or intrusion tolerance. Furthermore, proactive fortification goes beyond the usual goal of intrusion tolerance, and helps to guard against losses of confidentiality, as well as losses of integrity.

The second contribution is a corollary of the first contribution. We show that intrusion resilience and crash tolerance in distributed systems can be orthogonal issues. It is possible to augment intrusion resilience with a scheme such as proactive fortification, allowing the system to survive intrusions into publicly accessible nodes, which may have already been designed, and even deployed, as a fault-tolerant system. So, what is on offer is a free choice as to which fault-tolerant replication strategies may be used.

The third contribution is a general comparison of the degree of protection that passive replication, active replication and proactive fortification can give to a distributed system. This comparison is performed in such a way that it is abstract enough to be relevant for a large spectrum of real world attacks. A large range of probabilities of successful intrusion is considered, without reference to the combination of local defences and attack techniques that lead to these probabilities. This leads to a model that is more general than those in [34, 39], allowing replication schemes to be compared separately from the defences that may be deployed on the individual machines used in the scheme.

The fourth contribution is an efficiency evaluation of proactive fortification systems. This consists of two cases. First we evaluate the latency and throughput of a proactive fortification framework with a sample application running on a common middleware platform and compare this to the latency and throughput of the same application running without proactive fortification. Then, we evaluate the latency and throughput changes when we augment an online shopping application running on Apache Tomcat web servers with proactive fortification. This second evaluation involves the use of clustering features available in Apache Tomcat to provide state transfer and heartbeat messages. This provides a lightweight implementation of proactive fortification,

showing that the introduction of proactive fortification can make use of beneficial features of existing software.

These efficiency evaluations indicate that proactive fortification is a practical solution to increasing intrusion resilience which can be easily added to a variety of real-world systems.

1.4 Thesis Structure

The thesis consists of four main parts:

1. Background and related work are considered in Chapter 2.
2. Central concepts are introduced and theoretical performance analysis is carried out in Chapters 3, 4 and 5.
3. Implementation requirements are identified and system architecture is specified in Chapter 6.
4. Performance testing is presented in Chapters 7 and 8.

A more detailed summary of the sections is as follows:

- Chapter 2 considers the **key issues** in system replication and intrusion tolerance, then examines common attacks used to intrude into distributed systems. Current defences to these attacks are considered, as are counter-attacks against these defences. Protection of integrity is shown to be the main concern of intrusion tolerance, with secret sharing schemes providing a method of improving confidentiality that also has some drawbacks. Finally, the use of proxy servers as an intrusion resilience mechanism and the costs of using additional hardware to increase intrusion resilience are considered.
- Chapter 3 begins by considering common design patterns used in building distributed applications and how systems using these patterns can be proactively fortified. It then goes on to present the design of the Fortress system, **the conceptual model** for a proactively fortified system. The modelling assumptions and system requirements are specified, along with an analysis of the potential threats introduced by these system requirements.
- Chapter 4 describes our model for assessing the intrusion resilience of a distributed system. We begin by modelling attacks on an single server system, and then expand this into models for attacks on the FORTRESS system, SMR system and PB system. **Evaluation techniques** such as Markov Chain Analysis, Time-Dependent Stochastic Process Analysis and Monte Carlo Simulation are discussed.

- Chapter 5 Uses the models and techniques in Section 4 to compare the intrusion resilience of the FORTRESS, SMR and PB system. An analytical result is proved about the **relative performance** of the active replication systems and proactive fortification systems under certain conditions. Then, the intrusion resilience of the three systems is compared more generally. Next, the assumption that attacks are only possible against publicly accessible nodes is relaxed to allow attacks to be made against all nodes that handle client requests. The intrusion resilience of the FORTRESS system is calculated in this case and compared to the SMR and PB systems analysed previously. Finally, the differences in intrusion resilience are discussed, illustrating the advantages and disadvantages of the systems in a variety of different circumstances.

- Chapter 6 examines the requirements for system checkpointing in systems with short or long lived computations, and hence the **requirements for state transfer**. It goes on to define four state transfer schemes and consider the advantages and disadvantages of each. It then presents the **design** of a state transfer framework for proactive fortification. The software processes required for the controller unit, proxies and server nodes are defined, along with the interactions between them. Then, this design is refined to the level of individual objects within these processes. The implementation environment in which our FORTRESS framework was produced is discussed, along with **implementation** decisions made for each of the components.

- Chapter 7 presents the server application and client components designed for **evaluating the overhead** of our FORTRESS framework. Then the evaluation strategy used for determining the overhead of the proactive fortification framework is detailed. It then presents an analysis of the change in latency and throughput caused by proactive fortification as the intervals at which state transfer, system heartbeat and migration occur are varied. Finally, the findings about the efficiency and practicality of proactive fortification for distributed applications are summarised.

- Chapter 8 examines the possibility of applying proactive fortification to a **large scale web application**. The differences in system architecture between such an application and the smaller distributed systems we have considered previously are outlined. The changes these differences may make to our previous calculations of intrusion resilience are analysed and the likelihood of intrusion resilience being equal or greater to that of the FORTRESS systems considered previously is shown. It then presents the design for a proactive fortification of an Apache Tomcat based online shopping system. This proactive fortification makes use of the clustering features of Apache Tomcat, and the load balancing features of Apache HTTP server. Efficiency testing is presented for the online

shopping system; latency and throughput are measured and compared to an unfortified system, both for simple web pages and shopping cart pages that contain personalised dynamic content. Finally, the findings about the efficiency and practicality of proactive fortification for large scale web applications are summarised.

- Chapter 9 summarises the findings of this thesis, and draws conclusions as to the practicality of proactive fortification as an intrusion resilience mechanism. Finally, possible future work is discussed.

Chapter 2

Background and Related Work

This chapter begins by examining intrusion tolerance techniques and the theory that underpins them. First, system models using synchronous and asynchronous timing assumptions are examined (Section 2.1). Next, the nature of Byzantine behaviour and consensus in the face of possible Byzantine failure is considered (Section 2.2), then state machine replication is introduced (Section 2.3), and the combination of the two to provide intrusion tolerance is detailed (Section 2.4). The need for proactive recovery to bound the time-period in which an attacker has to complete an attack is then considered (Section 2.5), along with the temporal requirements this adds to system assumptions, and the need for schemes like proactive recovery wormholes to provide these requirements (Section 2.6).

The nature of common system attacks and the tools attackers have at their disposal are considered next. The use of distributed attacks as means to increase attacker power are considered (Section 2.7) , followed by code injection attacks as an example of a common specific attack encountered by distributed systems (Section 2.8).

Section 2.8 then catalogues defences against buffer overflow attacks, and the attacks that in turn have been developed to defeat these defences. These defences are then summarised, showing a general picture of more uncertainty being added into attacks as more defences are added.

The concept of proactive obfuscation is shown to be a generalisation of the use of defences that involve randomisation, and also a way of producing large numbers of artificially diverse versions of a system (Section 2.9).

The fact that the intrusion tolerance methods considered so far only address integrity is noted, and secret sharing schemes are considered as a possible way of adding confidentiality to intrusion tolerant systems. The limitations of current secret sharing schemes are highlighted (Section 2.10).

The use of proxy servers as a way to protect servers from attack by malicious clients is considered (Section 2.11) . Finally, the use of additional hardware in providing

intrusion tolerance or resilience is considered, along with a discussion of where the major costs in running a system will be found (Section 2.12).

2.1 Synchronous and Asynchronous System Models

A synchronous system model is a distributed system model where all processes have access to a shared global time, all messages between processes are delivered within a known time bound, and all local processing operations are performed within a known time bound [61]. An asynchronous system model is a distributed system model where there is no shared global time and no timing assumptions are made about message delivery or local processing [61].

2.2 Byzantine Consensus

Byzantine behaviour is defined as a process performing arbitrary behaviour, which may include sending messages with arbitrary content to other processes [61]. The idea of a group of processes, some of which may exhibit Byzantine behaviour, needing to reach agreement was introduced in [40], although the term Byzantine Generals Problem, from which Byzantine behaviour is derived, was first used in [33].

Many algorithms have been presented to produce consensus among a group of processes in which some may exhibit Byzantine behaviour. Both [40] and [33] provide algorithms to produce Byzantine consensus.

In [22] it was proven that deterministic consensus is not possible in an asynchronous system in which one or more processes may crash; this is commonly referred to as the FLP impossibility. As Byzantine behaviour could include mimicking a crash, this means that deterministic consensus is also not possible in an asynchronous system when one or more processes may exhibit a Byzantine fault. This leads to Byzantine consensus algorithms falling into two main categories;

1. Randomised algorithms.
2. Oracle based algorithms.

Randomised algorithms circumvent the FLP impossibility by being probabilistic: they have a probability of terminating within any given time period, and this probability converges to 1 as the time period increases. Examples of randomised Byzantine consensus algorithms include those in [6, 7, 11, 12, 62, 42].

Oracle based algorithms require certain timing assumptions to hold for a sufficiently long period of time which may not be known a priori. This allows them to use timeouts to differentiate between non-responsive faulty processes and correct processes

that have yet to respond. This technique is known as using an oracle or failure detector. Using failure detectors to solve consensus is tricky and involves avoiding incorrect execution steps when the timing assumptions do not hold. The consensus algorithm was first presented in [15]. Other examples of oracle based algorithms for Byzantine consensus include those in [5, 19, 20, 24, 31, 35].

Wormholes, which we discuss in the context of intrusion tolerance in Section 2.6, implement timing assumptions by using a synchronous network for special messages, alongside an asynchronous system for other messages.

2.3 State Machine Replication

State machine replication involves a service being replicated as several deterministic state machines [48, 49]. That is, several copies of the service are made that, given identical inputs, will reach identical states. These replicas may be running identical software, or may be running diverse executables that respond with logically identical states when given the same inputs.

All sources of non-determinism must be removed or their outcomes be resolved for replicas to produce identical states, given that an identically ordered set of inputs is supplied. This can be a relatively hard task in practice, as sources of non-determinism include time-stamps, results of system calls, multi-threading and task scheduling.

2.4 Intrusion Tolerance

Intrusion tolerance strategies combine the techniques shown in Sections 2.2 and 2.3 to prevent a loss of integrity when faced by a malicious intruder. The system to be protected is replicated as a set of deterministic state machines, and a Byzantine consensus algorithm is used to reach consensus on the order in which client requests are processed by these replicas. As Byzantine behaviour is completely arbitrary, the actions of a malicious intruder are simply a worst-case example of Byzantine behaviour.

Any requests sent to the system are executed by all replicas in an agreed order. The need for ordering requires the group to achieve consensus on the order in which the requests are to be executed. This allows the system to maintain integrity when up to some pre-set number n of replicas have come under the control of a malicious intruder, as, for small enough n, all correct replicas will still reach consensus on the correct order of processing, and all correct replicas will process these requests correctly. Similarly, while a replica under the control of a malicious intruder may return an incorrect response to a client, the client will receive enough correct, and

hence identical, responses to have a majority of correct responses. Hence the client can trust the majority value to be correct.

Notable among intrusion tolerant systems are those presented in [13, 30, 43]. Among these systems, there is a requirement for synchrony to hold, starting from an arbitrary moment for a sufficiently long duration, an assumption that may be unrealistic in real-world systems prone to attacks.

One major criticism of these intrusion tolerant systems is the danger of any assumption about the number of replicas that can be compromised failing to hold over a sufficiently long period of time. A determined attacker may be prepared to continue compromising replicas without performing any other malicious action until they have compromised more than the number of intrusions that the system can tolerate. Given a sufficiently long running system, an attacker that is capable of compromising one replica, an assumption without which intrusion tolerance would not be needed, will eventually be able to compromise enough replicas to compromise the system.

This criticism is addressed in [14] and the concept of proactive recovery is introduced. It is also noted in [64] that, while state machine replication techniques help to provide integrity they can actually impact confidentiality negatively. This happens because each replica contains the full system state, and hence one intrusion results in the attacker obtaining this system state.

A second practical problem with existing intrusion tolerant systems is the need for state machine replication. The requirement to remove all non-determinism from the system can add significant time, cost and complexity to a project, and may in some cases result in the abandonment of intrusion tolerance in favour of crash tolerance and an attempt to make the system as attack resistant as possible.

2.5 Proactive Recovery

Proactive recovery [14] involves replicas in an intrusion tolerant system being periodically recovered from a possibly compromised state. After a period of time, each replica is rebooted, terminating any malicious code and causing any malicious intruder who has access to the system to lose that access. This reboot may also involve a refresh of the system software to remove any alterations made to the system by an intruder. Then, system state is restored to the consensus of the other replicas, eliminating any past influence of the malicious intruder.

This process happens on a timer whether intrusion of the replica is suspected or not, removing any risk of an intruder evading the mechanism by concealing the intrusion. The use of proactive recovery makes an important change in the assumptions made about an intrusion tolerant system. Instead of the original, possibly unrealistic, assumption that the system will only have up to n replicas intruded in its operational

lifetime, the assumption now becomes that the system will have up to n replicas intruded during the period of time between proactive recoveries. As this period of time is bounded, this allows a sufficiently large number of replicas to be included in the system to ensure that an attacker will not be able to compromise enough of them to compromise the whole system.

An alternative approach to rebooting replicas and then immediately using them again is found in [66]. Here, a pool of spare replicas is maintained and when a replica is due to be rebooted a spare takes its place. The replica then reboots and becomes part of the pool of spare replicas.

There have been two key criticisms levelled at proactive recovery. Firstly, the assumption that an attacker has a fixed period of time in which to compromise replicas may be violated by attacks which prevent or slow down the recovery mechanism. This problem cannot be addressed in an asynchronous system model as time is explicitly excluded from the model [53, 54].

Secondly, reboot and refresh may not always be enough to return a replica to an uncompromised state. An attacker who has successfully compromised a replica may have gained sufficient knowledge in doing so that a simple replay of the attack will compromise the replica again. In this case, the system will essentially only be as intrusion tolerant as if proactive recovery was not used.

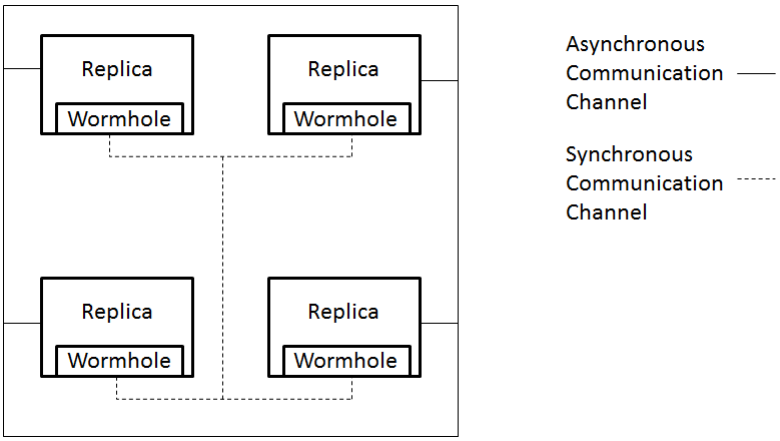
2.6 Proactive Recovery Wormholes

Wormhole based systems involve the use of a hybrid system model, where a payload system (that may be synchronous, partially synchronous or asynchronous) and a set of synchronous wormhole systems coexist, as shown in [56]. The wormhole systems may also have a synchronous method of communication with each other.

Proactive recovery wormholes [56] are a practical method of using wormholes to ensure that reboot and refresh in proactive recovery systems does occur within a bounded time from when it is expected. Each replica has a local proactive recovery wormhole, a tamper proof hardware module running a real time operating system with a clock separate from that of the replica. A secure private channel is provided between these wormholes such that there is timed point-to-point reliable communication between every pair of wormholes and timed reliable atomic broadcast from one wormhole to all others is possible. The wormhole clocks are synchronised within a known bound.

Proactive recovery wormholes are assumed only to fail by crashing and at most f wormholes can crash during the lifetime of the system, where f is the number of intrusions that can be tolerated by the system. The failure of a wormhole is assumed to cause the crash of the corresponding replica.

Figure 2.1: Proactive Recovery Wormhole System



The payload system performs the normal tasks of processing client requests and reaching consensus in a state machine replicated intrusion tolerant system. The proactive recovery wormholes govern the proactive recovery of each replica, preventing an adversary from increasing the recovery interval, and preserving availability by ensuring that the number of replicas rebooting at any given time is smaller than a pre-set bound.

Figure 2.1 illustrates the structure of a proactive recovery wormhole system with 4 replicas.

2.7 Distributed Attacks

Distributed attacks involve an attacker using multiple pieces of hardware to generate malicious requests for use in some other form of attack against a publicly accessible server. These attacks can use much larger numbers of machines than a typical attacker could be expected to buy, by using entities such as bot-nets [1]. Here an attacker first compromises a large number of easily compromisable machines and uses them to launch malicious requests. This is a relatively easy task, as the attacker can target any machines connected to the Internet, so has a very large pool of systems from which to find vulnerable ones. Another option for an attacker is to pay for access to a bot-net created by a third-party for the purpose of selling access.

The possibility of launching distributed attacks means that, except for incredibly large scale systems, an attacker will generally be able to launch sufficiently many attacks from a sufficiently diverse set of machines to send malicious requests to as many system nodes as are available, at a constant rate. For example, the rate at which an attacker can launch malicious requests against each of four publicly accessible nodes will be the same as the rate at which an attacker can launch malicious requests against each of eight publicly accessible nodes.

This means that merely increasing the number of nodes that have to fail before an

attacker has compromised the integrity of a system is not a sure way of increasing the intrusion-resilience. In general the attacker's rate of launching malicious requests will just increase as more nodes are made available.

This does not mean however that the attacker's rate of launching malicious requests towards each node will be arbitrarily high. In distributed denial of service attacks it is often the case that bot-nets are used to produce as many requests as possible [23]. Distributed denial of service attacks specifically attempt to disrupt availability of a system; generally this is done by using a sufficient volume of requests, TCP connections or UDP packets to exhaust the system's capacity to process them. The distributed attacks that we consider are attacks which attempt to breach the integrity or confidentiality of a system. Here an attacker does not want to flood the system with requests to the point that it shuts down. The attacker's goal actually requires the system to be sufficiently available to allow them to use it to read or alter system state.

This results in an upper limit to the number of requests sent in a given period of time. In practice this upper limit is likely to be set either by the number of requests the system can process, or the number of malicious requests that can be sent without causing suspicion among system administrators that will result in additional defences being applied or legal action being initiated.

2.8 Code Injection Attacks and Defences

Here we catalogue common code injection attacks against vulnerabilities in distributed systems, and defences that have been developed to stop them. This shows that, while the techniques developed so far have been unable to eliminate code injection attacks, they have caused the individual malicious requests launched during code injection attacks to succeed with a very small probability, necessitating an attacker to launch a large series of malicious requests before success is likely.

2.8.1 Code Injection Attacks

Code injection attacks involve an attacker generating malicious client requests in an attempt to run their own code on the machine being attacked. These requests are designed to take advantage of vulnerabilities in either the code processing the client requests or the software that runs this code such as a web server or operating system. Typically, the attacker will be trying to use a vulnerability to cause the operating system on the client machine to interpret part of the malicious request as being part of the code it is running. Two common examples of this are the buffer overflow attack and the SQL injection attack.

2.8.1.1 Buffer Overflow Attack

Buffer overflow attacks involve the attacker providing a parameter that is larger than the buffer allocated for it by the system. The expected behaviour when this happens would be some sort of error state and a refusal of the request. This is generally what happens with type-safe languages and defensively programmed systems.

However, in some cases the size of the parameter may not be checked and instead it is copied into a buffer that is too small for it. One example of where this is possible is with the string copy function *strcpy* when programming in C [38]. This function, for efficiency reasons, leaves bounds checking to be carried out by the programmer using it, when it is required. Similar efficiency concerns may result in programmers not performing bounds checking, as they believe that it will have already been performed on the data before their code receives it. This issue, or simple programmer error, can result in buffer overflow causing client specified data to be written past the end of the buffer.

Other common C functions that do not perform bounds checking and can cause this type of overflow include the string concatenation function *strcat*, the *gets* function which reads a line from the standard input and stores it in a buffer and the *sprintf* function which prints to a buffer [38].

When client data is written past the end of the buffer, this will cause other data on the stack to be overwritten. It is also possible, with a sufficiently long buffer overflow to overwrite the return pointer for the function handling the data. This then allows the attacker to include his own return pointer in the overflowed data, causing the execution of whatever code this pointer targets.

If an attacker includes malicious code in the data which will overrun the buffer and a new return pointer that points to this code then the code will be run, potentially allowing the attacker to do anything that the legitimate code had permission to do on the target system. This use of this procedure is demonstrated in [38].

The attacker does have to know the correct address of the malicious code to send the return pointer to. This is fairly easy to determine in practice however, using knowledge of the system architecture and trial and error over a relatively small set of possible values. The set of values can be further reduced by using a technique known as a NOP slide or NOP sled [57]. Here a large number of NOP instructions, instructions that have no effect on the system state, are included before the working part of the malicious code. Then, if the attacker's guess for the start address of the code is anywhere within this region of NOP instructions the malicious code will be executed just as if the attacker had successfully guessed the address of the start of the code.

Similar attacks can be produced by overflowing data structures on the heap such as in [4].

2.8.1.2 SQL Injection Attack

SQL injection attacks involve the attacker including control characters in a parameter so that an attempt to use this parameter in an SQL query will cause the part of the parameter after the control characters to be interpreted as a new SQL query, entirely defined by the attacker [26]. This can result in the modification or deletion of any of the contents of the database that the system has permission to modify.

Normally, SQL injection attacks require the attacker to know the structure of the database used by the target systems, as queries will only be successfully executed if they contain the correct names for tables and fields. However, a determined attacker can use blind SQL injection techniques to attempt to discover names of fields and tables. This involves crafting SQL injection attacks in such a way that a normal response will occur if the field or table name is guessed correctly, and a recognisably different response will occur if the field or table name is guessed incorrectly [26]. Then, a brute force or dictionary based scheme can be used to attempt to determine the actual table and field names.

We will mainly consider buffer overflow attacks in this thesis, but it is worth noting that there are sufficient similarities between buffer overflow attacks and SQL injection attacks to allow proactive fortification to be effective against SQL injection attacks.

2.8.2 Write or Execute Only Memory Pages

Write or execute only memory pages are a simple technique used to attempt to stop buffer overflow attacks. Every memory page is given a bit value to mark it as either writeable or executable [60]. Hence, when the attacker puts malicious code into a buffer, the memory pages containing this code will be marked as writeable. This means that when the overwritten return pointer directs control flow to this code it will not be executed and will instead cause a segmentation fault.

While write or execute only memory pages do stop the standard buffer overflow attack from succeeding, there is a modified type of buffer overflow attack that they do not prevent. This is the return-to-libc attack.

2.8.2.1 Return-to-libc Attack

The return-to-libc attack [18, 51] does not place malicious code in the overflowed buffer. Instead, it overwrites the return pointer with the address of the *system* function in the standard C library. This will cause the *system* function to be executed, in turn executing whatever kernel function is passed to the *system* function as a parameter. The *system* function will be marked as executable, and no attempt will be made to execute the malicious code in the buffer, simply to pass its contents to the *system*

function. This allows the attacker to bypass the protection provided by write or execute only memory pages.

This technique requires the attacker to know or guess the address of the standard C library. However, this is fairly easy in practice using knowledge of standard system architecture combined with trial and error over a relatively small set of values.

2.8.3 Address Space Layout Randomisation

Address space layout randomisation (ASLR) involves randomising the positions of important areas of data within the execution space for a process [59]. This results in an attacker having to guess the positions of the stack and standard libraries when attempting a buffer overflow or return-to-libc attack. This means that any given buffer overflow or return-to-libc attack has a very small chance of success, and can generally be expected to cause a system crash rather than the execution of malicious code.

[51] evaluates the effectiveness of ASLR. This evaluation shows that address space layout randomisation is relatively easy to defeat in practice using brute force methods when considering 32-bit system architectures. The attack method presented in [51] makes use of the fact that, in systems designed for high availability, a new process is usually spawned when an existing process crashes. For example, the Apache HTTP web server application has a daemon which forks a new process every time a process crashes. This allows the attacker to launch attacks using each possible randomisation key in turn until the one that results in the malicious code running is found. Every unsuccessful guess simply results in the process handling the request crashing and a new process being spawned.

The average time until a machine using ASLR was compromised in the experiments performed in [51] was 210 seconds. This shows both that ASLR is insufficient as a defence mechanism in itself, at least where 32-bit system architectures are concerned, but is capable of significantly increasing both the time taken for return-to-libc attacks to succeed and the number of malicious requests required.

A similar method to address space layout randomisation can be used as a defence against SQL injection attacks. We mentioned in Section 2.8.1.2 that launching a successful SQL injection attack requires the names of database tables and fields to be known or guessed. This can be made both more difficult, and unique to each particular execution of a process, by randomising the names of tables and fields at run-time. Both the table and field names used in the database and in the code of the process are identically randomised, so that SQL queries in the code work as normal, but an attacker has to guess a set of names unique to this process. An alternative to randomising the database can be used if multiple processes need to access the same database. A database proxy can be used that makes use of a SQL parser to extract

the table and field names from the query and applies the de-randomisation algorithm to them. Any Table or field names that has been injected into the query will be de-randomised and, unless the attacker has correctly determined the randomisation key, will be transformed into invalid table or field names.

2.8.4 Instruction Set Randomisation

Instruction set randomisation [29] involves using an encryption key to encrypt the machine code instructions that make up each process when it is started. These instructions are then decrypted as they are executed. This results in a situation where anybody wishing to generate valid instructions for a process will need to know the encryption key. Hence an attacker is not able to inject a runnable code without determining the encryption key.

One limitation of instruction set randomisation is that it is vulnerable to return-to-libc attacks, as the return-to-libc attack does not involve injecting code. When a return-to-libc attack takes place, a *system* call will be made with the name of a kernel function, typically one to open a shell, as a parameter. This does not involve the use of machine code instructions, so will not be affected by instruction set randomisation.

Another major vulnerability of instruction set randomisation are carefully crafted incremental attacks such as the one detailed in [57]. Here, brute force methods are used with a 1-byte or 2-byte instruction to discover the part of the key relating to these one or two bytes. Once this is found, the attacker can then move on to the next byte, optimising the process by using longer instructions when the keys for a number of bytes are already known. The experiments performed in [57] showed that even large randomisation keys resulted in intrusion times of less than an hour using this technique.

We note that this technique, as with that in [51], significantly increases both the time taken for an attack to succeed and the number of malicious requests needed. There is also a requirement for the attacker to be able to determine whether an attempt has been successful or not, in this case by the monitoring of an open TCP connection.

A method analogous to instruction set randomisation is available as a defence against SQL injection attacks. SQL instructions can be separately randomised for each executable and de-randomised before being executed. The de-randomisation mechanism will apply the de-randomisation algorithm to every value that is not semantically identified as a variable in the SQL statement. Therefore any malicious requests generated by a SQL injection attack will be de-randomised, and, unless the attacker has successfully guessed their randomisation key, this will result in them being invalid SQL commands, and hence rejected. Alternatively, as in [9], a database proxy can be used with a SQL parser that has the standard SQL keywords replaced by the randomised ones. This parser will reject any query that it sees as invalid, including any SQL

statement that has been injected without the correct randomisation key being used. If it receives a valid query then it will simply replace every randomised keyword with the un-randomised equivalent and pass it to the database.

2.8.5 Canary Values

Canary values (e.g. StackGuard [63]) are a technique to prevent buffer overflow attacks: A value is inserted before each return pointer, and the system checks that this value is still correct before allowing the function to return. If the value is not correct then a segmentation fault occurs. This prevents an attacker from overwriting the return pointer, as they are unable to overwrite the values before it.

There are however two common ways to defeat canary values. The first is to overwrite the canary value with itself, usually by including the value as part of a string or array. This does however require the attacker to know the canary value, which can itself be overcome by using random canary values.

The second method is to overwrite a pointer before the canary value in such a way that it points to the return pointer, and use this pointer to alter the return pointer. This can, in turn, be prevented by making the canary value a function of the return address. Then, any change in the return address would result in the canary value becoming invalid.

One attack that has been suggested in [10] against canary values that are a function of the return address is to use overwritten function pointers to determine how the canary value is calculated from the return address and then overwrite the canary value with a canary value calculated for the new address.

Other attacks against canary value based systems are demonstrated in [44].

2.8.6 Return Address Cloning

This technique involves copying the return address of a function to a special area of memory before the function is entered. When the function is about to return there are two possible actions, depending on the particular implementation of return address cloning. The first possibility is that the return address in the table is used rather than the return address on the stack, negating any effect of the return address on the stack being changed. The second possibility is that the two return addresses are checked, and if they do not match then execution is terminated.

One way of defeating the first possibility is for the attacker to use a pointer based attack similar to the ones in [10] to overwrite the values in the return address table, as suggested in [44]. This will then have the same result as overwriting the return pointer in an unprotected system. Similarly, the second possibility may be defeated

by overwriting both the values in the return address table and the return pointer. Another attack that is suggested in [44] is making use of the way that execution is terminated in some implementations of return address cloning to run malicious code as part of the termination process.

2.8.7 Summary of Code Injection Defences

The defences against code injection considered in Sections 2.8.2-2.8.6 and the attacks devised to circumvent them show two key trends. As more defences are added to a system, attacking that system becomes more complicated and more uncertainty is added into the outcome of an individual attack attempt. Techniques such as Address Space Layout Randomisation and Instruction Set Randomisation explicitly add randomised elements into the system which have to be determined by the attacker. Techniques such as Canary Values, Return Address Cloning and Write or Execute Only pages constrain the types of attacks that can be launched, requiring more complicated attacks that in turn require more of the randomised structure to be determined.

2.9 Proactive Obfuscation

In [45], schemes such as address space layout randomisation and instruction set randomisation are generalised to produce the concept of proactive obfuscation.

Proactive obfuscation combines two techniques: program obfuscation and periodic reobfuscation. An obfuscater takes two inputs, a program P and a secret obfuscation key k and outputs a program P' that is semantically equivalent to P. A vulnerability in P is also present in P'; it cannot however be exploited without knowing k.

Program obfuscation is essentially randomization of executables obtained through a variety of techniques, such as address space layout randomization (ASLR) and instruction set randomization (ISR), or a combination of them.This then allows artificially diverse executables to be produced from one piece of code, by applying the chosen set of randomisations.

Program obfuscation within an SMR system works as follows. All server replicas have obfuscated executables obtained from a common software P, but each replica is obfuscated with a distinct, randomly selected obfuscation key. Consequently, if the SMR system is designed to tolerate at most f intruded replicas, then at least $(f + 1)$ of the keys used must be determined by the attacker before the vulnerability in P can be exploited and the system intruded into. Thus, given that the key selection process is securely carried out, the amount of work required of an attacker to compromise the SMR system is $(f + 1)$ times the work needed to intrude a single replica.

Even though the key-space is large, an attacker can deduce the keys used within an SMR system over time by launching a series of de-randomization attacks (discussed in Sections 2.8.3 and 2.8.4). This is mitigated against by the use of periodic re-obfuscation; replicas are periodically shut down, rebooted, randomized with a different, newly selected k, and initialized with the correct service state. Thus, if an attacker manages to deduce the key used in a replica, then that advantage is erased once that replica is re-obfuscated with a different key.

The actual degree of diversity produced by program obfuscation may vary relative to particular attacks that are attempted against the system.

For example, if address space layout randomisation with δ possible randomisation keys and instruction set randomisation with ϵ randomisation keys are used then, from the point of view of an attacker using a buffer overflow attack where the return pointer is redirected to malicious code on the stack, there are $\delta\varepsilon$ possible executables as this attack will need to determine both keys.

However, from the point of view of an attacker using a return-to-libc attack there are δ possible executables. This is because the attack will require determining the correct address space layout randomisation key to succeed, but will be unaffected by instruction set randomisation.

The availability of brute force and incremental attacks against randomised systems suggests that Proactive Obfuscation cannot guarantee complete protection against buffer overflow attacks. Instead, Proactive Obfuscation has two key aims; to increase the time taken to compromise a particular executable, and to make the compromise of different executables independent from each other.

This then allows an active replication system to be produced where individual executables can be replaced by new executables frequently enough that an attacker is unlikely to be able to compromise sufficient replicas between replacements to compromise the system.

While we have considered randomisation as a scheme to reduce the effectiveness of buffer overflow attacks in this thesis, it is worth noting that randomisation can also be applied to the structure and naming of SQL tables. This means that Proactive Obfuscation can also be used as a valid defence against SQL injection attacks.

2.10 Secret Sharing Schemes

Secret sharing, introduced in [8, 52], is a method for maintaining the confidentiality of a piece of information when malicious intrusion is possible. We noted in Section 2.4 that the state machine replication techniques do not generally improve the likelihood of confidentiality and in many cases reduce it due to an intrusion into just one replica

exposing the full system data to the intruder. In contrast, secret sharing schemes distribute data in such a way that the data of several replicas will be required before any meaningful information can be extracted.

Secret sharing has been used in a quorum system to improve the confidentiality of time-stamped files in [28]. However, as noted in [64], this technique is not generalisable to storing data that requires processing. If processing is required then, normally, the data would have to be recovered from the shares, processed and then re-shared. Hence an attacker could simply compromise one machine and then make a request to process data. We note that, in some cases, techniques have been produced to allow encrypted data to be processed without decryption, such as those detailed in [2, 47], suggesting that techniques for processing shares of data without recovering the data may be possible. However, the techniques currently available for processing encrypted data do not allow arbitrary processing, and instead are very limited in the range of operations that can be performed. e.g. It is possible to perform arbitrary arithmetic operations on two pieces of encrypted data, resulting in a third piece of data that is an encryption of the result, but this requires that homomorphic encryptions of these two pieces of data are individually supplied, rather than being part of a bigger document.

There may also be problems in some application domains with the right to modify data when secret sharing is used. The quorum system considered in [28] has authorised clients retrieving files and submitting newer versions. Here it is clear that the client owns the file and is authorised to make arbitrary changes. However, in some commercial applications there may be a need for authorised clients to be able to read confidential information and only make certain specified changes to it. The enforcement of these conditions will be required to sit on the server side, otherwise an error or malicious behaviour from the client could result in an undesirable system state occurring. However, this then requires one or more servers to have access to the unshared data, meaning that compromise of this server will result in confidentiality being compromised.

An attempt has been made to use secret sharing to protect confidentiality, integrity, and some cases of availability in [65]. This involves generating shares of values to be protected and storing them in memory locations generated from each other, making it hard for an attacker to find the locations of all shares or to reconstruct the values without recovering enough shares. The alteration of critical values by an attacker, including some values used in buffer overflow attacks can also be detected through the use of these shares. However, this scheme will not protect the confidentiality, or integrity, of data if the entire system is compromised. Instead, it gives added protection against certain types of intrusion being successful, and protection against an attacker accessing data without fully compromising the system.

2.11 Proxy Servers

Proxy servers present themselves to clients as the system to be accessed. They receive client requests, pass these requests to the servers that will process the requests, receive the response from the servers and pass this back to the clients as their own response. A malicious intrusion into a proxy server does not compromise confidentiality or integrity as the proxy server does not store system state, it merely passes requests and responses between client and server. Furthermore, a malicious intrusion into a proxy server does not compromise availability, as long as there are still other uncompromised proxy servers available and the client is aware of this.

However, a malicious intrusion into a proxy server may still cause the following undesirable consequences:

1. Servers may be given incorrect client requests by an intruded proxy server.
2. Clients may be given incorrect responses by an intruded proxy server.
3. Intruded proxy servers may be used to launch attacks against servers.

Consequence 1 may be discounted in a well designed system, as servers need to be able to handle and reject incorrect client requests. If this is not the case, then a malicious third-party could simply generate malicious client requests and pass them to a correct proxy server to pass to the server with the same effect.

Consequence 2 can be mitigated by the use of digital signatures to allow the client to check that the response they have received is the response that was sent by the server. This will however necessitate that the proxy servers are not able to mislead the client about the public key of the server.

Consequence 3 is essentially unavoidable. However, if the servers are designed with possibly malicious proxy servers in mind, then this situation is no different from that encountered by a server that directly receives malicious requests from clients. There is also a possibility that proxy servers may be harder to intrude than the servers they pass requests to. This is because proxy servers are likely, in many cases, to be considerably simpler pieces of software than the servers that actually process client requests.

Another possible avenue for a malicious intruder when faced by proxy servers is to attempt to bypass them. That is, to attempt to treat the proxy servers as part of the communication channel that malicious requests are delivered by, and concentrate on attacking the servers themselves. We note that, while the de-randomisation attacks detailed in Sections 2.8.3 and 2.8.4 involve sending a series of malicious requests, and hence individual requests may bypass proxy servers, the whole attacks are less likely to.

Both of these attacks require the attacker to monitor a TCP connection to see if the attack has been successful or not, and neither a successful nor unsuccessful attack will return a valid result. Instead, a successful attack will cause the TCP connection to stay open longer than an unsuccessful attack. When proxy servers are introduced, the immediate termination of a TCP connection by the server, as opposed to a short delay before the connection terminates will not be directly measurable, and the response of proxy servers may well be identical in both cases.

One possible way of modifying the attacks to provide the needed feedback is to have the server send a response directly to the client if the attack has been successful. There are however two practical problems with this. Firstly, system firewalls may be configured in such a way that this is not possible, and the proxy servers may be able to screen for invalid responses that are sent via them. This may however be avoidable if responses are sent via the proxy servers that appear to be valid, but have some property that is meaningful to the malicious intruder in them. Secondly, both attacks make use of the injection of very small and simple pieces of code. The injection of much larger pieces of code to generate a response to the client would not fit the requirement of one or two byte instructions in the attack against instruction set randomisation. Similarly, generating a message that can be successfully sent via a proxy to a particular client may not be possible via a return-to-libc attack, or may require a much higher level of trial and error than just finding the location of the system libraries.

2.12 Additional Hardware

Schemes involving state machine replication or the use of proxy servers may require additional hardware to that used in unreplicated systems without proxies, although proxies are often already present for use in load balancing. This introduces additional cost due to the need to purchase, run and maintain this hardware, and also the need to modify software for schemes like state machine replication. This cost can be split into five key areas:

1. Hardware purchase cost - The cost of purchasing the additional hardware, including any installation and delivery costs. This cost will be an initial expense in setting up an intrusion tolerant or resilient system.

2. Software purchase cost - The cost of purchasing any additional software licences and installing this software. This cost will be an initial expense in setting up an intrusion tolerant or resilient system.

3. Hardware running cost - The cost of energy usage in running the additional hardware. This cost will be an ongoing expense in running an intrusion tolerant or resilient system.

4. Hardware maintenance cost - The cost of physically maintaining and managing the additional hardware. This cost will be an ongoing expense in running an intrusion tolerant or resilient system.

5. Software development cost - The cost of developing custom software or modifying existing software to comply with the requirements intrusion tolerance or resilience schemes. This cost is both an initial expense in setting up an intrusion tolerant or resilient system, and an ongoing expense. The ongoing portion of the expense is caused by the need to make sure that any changes to existing software maintain compliance with the intrusion tolerant or resilient scheme.

Hardware and software purchase costs are generally fairly low compared to the costs of running hardware, and developing or modifying software. Hardware maintenance costs are also relatively low compared to the costs of running hardware and developing or modifying software, especially when large amounts of similar hardware are used, allowing economy of scale to be leveraged. This suggests that, while the costs of additional hardware and software will contribute to the overhead of intrusion tolerance or resilience, the majority of the financial overhead will come from running the additional hardware and making modifications such as removing determinism for state machine replication.

We note that the costs of running additional hardware are far higher when this hardware is active than when it is in a power saving mode. This means that machines actively taking part in an active or passive replication protocol or acting as proxies will contribute far more to the financial overhead than machines that are waiting to be used as system nodes at a future time.

We also note that often proxy servers are required for other purposes, such as load balancing, in large systems, and that these servers may be usable for intrusion-resilience purposes without additional hardware costs.

Chapter 3

The FORTRESS System

This chapter presents the system model for the FORTRESS system.

We begin by defining our system model for a publicly accessible distributed system, followed by the attack model for a publicly accessible distributed system facing the threats that the FORTRESS system is designed to provide intrusion resilience against. The system model is illustrated by a real world example. We then present the system model for the FORTRESS system.

Next, we detail the infrastructure for proactive obfuscation (IPO) a set of supporting mechanisms needed to enable a FORTRESS system to function. Finally, we consider the possibilities that these supporting mechanisms may open up to attackers.

3.1 System Model

We define a publicly accessible distributed system to consist of a *server* with an internal *state*. This server accepts and processes *requests* from *clients*. When a request is processed, the server may make changes to its internal state and/or send a *response* to the client. The nature of the state change or response is dependent on the identity of the client, the nature of the request and the current state of the server.

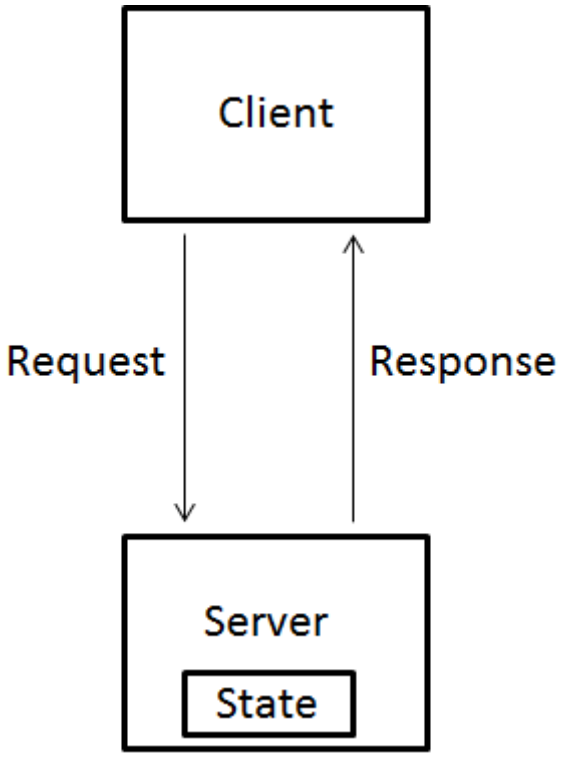
All requests are sent from the clients to the server via unauthenticated channels, so client identity can only be determined by authentication protocols carried out in the requests and responses. This system model is illustrated in Figure 3.1

3.2 Attack Model

We consider an attacker with the following objective:

- To modify the system state in a specific unauthorised manner (an attack against integrity).

Figure 3.1: System Model



The attacker is aware of a vulnerability in the server and can attempt to exploit this vulnerability by sending malicious requests to the server from many malicious clients that he controls. When a malicious request is processed there are two possible outcomes. Either the server will be compromised, allowing the attacker to achieve his objective, or some degradation of performance will occur but the server will not be compromised. The second outcome will occur if the attacker has failed to correctly determine details of the server internals needed to exploit the vulnerability.

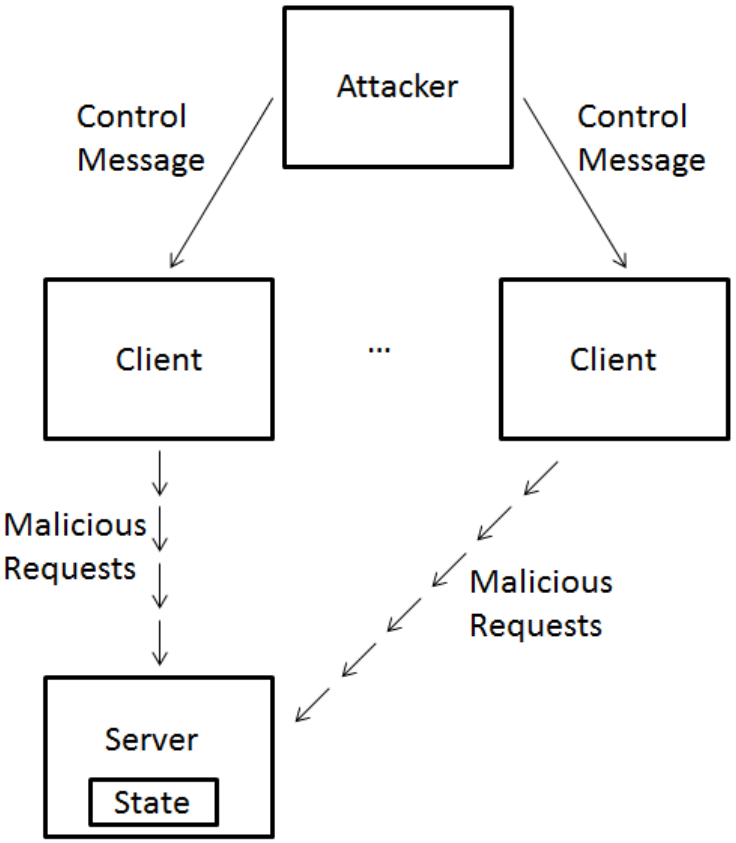
The attacker may make use of many clients to send repeated malicious requests to the server. However, the rate at which these requests are sent will not be arbitrarily fast, as there will be a limit to the rate at which the client can process requests. The attacker will continue to send malicious requests until the server is compromised.

The attack model is illustrated in Figure 3.2.

The attacker's strategy may include compromising the authentication details of a legitimate client through an attack on that client. However, if this results solely in the performance of actions that the attacked client was authorised to perform, this model does not consider this a successful intrusion. Instead, it is seen as an an attack on the client, which is an orthogonal issue. Denial of service attacks are also considered as an orthogonal issue; they serve as an attack against availability rather than integrity.

Non-classic attack models can involve attackers targeting system administrators or users of the same internal network as the server with social engineering or malware

Figure 3.2: Attack Model



attacks (e.g. the attacks detailed in [58]) and then using their machines to launch attacks or gather information to be used in attacks. Attacks against availability, or attacks designed to partially exhaust system resources, can be used to provoke system administrators to fail-over to a less secure system, as discussed in [3], or to deploy mitigation strategies that weaken the intrusion resilience of the system.

This attack model does not consider the use of attacks against availability to weaken intrusion resilience. Social engineering and malware attacks are not explicitly included in the model, but some classes of these attacks are implicitly included. We assume that the attacker knows a vulnerability in the system and has all of the necessary information to exploit it other than some details of the system internals. The origin of this knowledge is not specified and could stem from the use of one of these non-classic attack techniques.

3.3 Real World Example

We illustrate the use of the preceding system model and attack model by mapping them to an online shopping system with a buffer overflow vulnerability.

Here, the server in the system model refers to a physical machine running a web server that executes an online shopping application. Client requests may perform three key

activities; requesting a product web page, making an order or checking the status of an existing order. All of these requests will result in the server returning a response and a request to make an order will result in a change in system state if successful.

System state will include details of all products that are available for purchase and details of all current and previous orders.

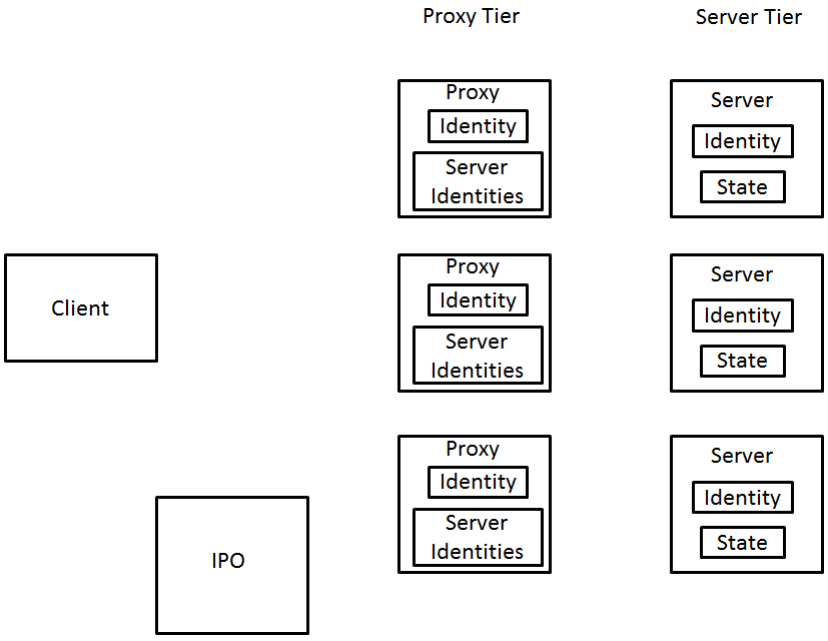
The known vulnerability in the system is an unchecked buffer that will receive the contents of the "additional delivery information" field from an order request. Malicious requests will consist of order requests in which the "additional delivery field" is long enough to overflow the buffer and contains malicious code and return pointer information to attempt to execute this code. Each malicious request will cause the server to give the attacker full access if that request contains a correct memory offset in the return pointer information and will cause the execution of this request to fail if the request contains an incorrect memory offset in the return pointer information.

3.4 Proactive Fortification of a Distributed Application

The FORTRESS concept can be used to proactively fortify a system conforming to the system model presented in Section 3.1. We note that the system model assumes that all state is stored within the server, and hence will be replicated if the server is replicated. In many systems there is likely to be a need to transfer information that is held in persistent storage as well as information about the current application state. There may also, if a transactional database is used, be a need to transfer transaction information. This will be considered when we examine state transfer mechanisms, using a technique similar to that in [41].

Some real-world systems are constructed using the three-tier architecture design pattern. Here the server tier is split into an application-tier and a data-tier. The data-tier handles any persistent storage that is required on the server side. The application tier receives requests from the client, and processes them, making requests to the data-tier to retrieve or update persistent information. When considering a system built using the three-tier architecture design pattern it may be necessary take the application-tier as the basic unit for proactive fortification and replication. This leaves the data-tier unreplicated by our scheme, although we note that it would be possible to use other replication schemes at the server-tier such as that in [32].

Figure 3.3: The FORTRESS System Model



3.5 The FORTRESS System Model

The FORTRESS approach requires introducing a new tier of nodes, which we call the proxy tier, in front of the server/application tier in two/three tier systems. The proxy tier, together with infrastructure for proactive obfuscation (IPO) described in Section 3.6, offer, like ramparts, a defensive platform against attacks on the server system (hence the name FORTRESS). The system model is illustrated in Figure 3.3.

3.5.1 The Proxy Tier

The proxy tier consists of three *proxy nodes*. Each of these proxy nodes has an internal *state* that includes identities of the nodes in the server tier. The proxy nodes receive *requests* from *clients* and forward these requests to the server tier, and receive *responses* from the server tier and forward these to the corresponding client. As proxy nodes are periodically replaced, each node has a unique *identity* that enables clients to send requests to that node. Clients are able to discover the identities of the current proxy nodes via the IPO, as discussed in Section 3.6.

Each proxy node runs identical software, and hence has identical vulnerabilities. However, each proxy node is obfuscated in such a way that exploitation of a vulnerability will require an attacker to determine the unique *obfuscation key* for that proxy node.

The proxy tier are publicly accessible and hence open to malicious requests from any client connected to the internet.

3.5.2 The Server Tier

The server tier consists of *server nodes*. Each of these server nodes is a replicated version of the original server defined in Section 3.1. These server nodes may be configured as a primary-backup system or an active replication system, or the server tier may consist of a single, unreplicated node. The nodes in the server tier receive requests from the proxy tier, process these requests, then send responses to the proxy tier.

The replicated system used for the server tier will determine whether these server nodes are obfuscated with identical or unique obfuscation keys; if active replication is used then the nodes will be have unique keys, if primary-backup replication is used then the nodes have identical keys. This is due to the fact that one intrusion on the primary is all that is needed to compromise a primary-backup system, so there is no benefit in having diversity between the primary and backups.

Nodes in the server tier are not directly accessible by clients. They will only accept requests from nodes in the proxy tier, update messages from other nodes in the server tier where appropriate, and control messages from the IPO as specified in Section 3.6. Each server node has a unique *identity* that enables proxy nodes to send requests to that server node and clients to determine that responses have originated from that server node.

We note one large advantage of the FORTRESS system here. There is no intrinsic need for the server nodes to be implemented as deterministic state machines, unless the server tier is configured as an active replication system. This means that proactive fortification can be used as an intrusion resilience strategy even when it is impractical to remove all sources of non-determinism from legacy code that needs additional intrusion resilience.

3.5.3 Replacement and State Transfer

After a set period of time the current proxy and server tiers will be replaced by new nodes using new diverse executables. This needs to take place in such a way that the following six requirements must be satisfied:

1. State information is passed from the current server tier to the new server tier without loss, corruption or the possibility of an attacker maliciously changing it.

2. All of the new nodes used to replace the server and proxy tiers are free of malicious intrusions at least until they start processing client requests.

3. All of the nodes in the new proxy tier know the identities of all of the nodes in the new server tier.

4. Clients know the identities of all of the nodes in the new proxy tier.

5. Clients know the public keys required to authenticate the digital signatures of the new server tier.

6. The overhead of replacement and state transfer in terms of time in which requests are not being processed is minimised.

3.5.4 Example of Execution

We show, in Figure 3.4, the normal execution of client requests for a FORTRESS system with a server tier consisting of a primary node and 2 backup nodes. Client requests are sent to the proxy nodes, these requests are in turn forwarded to the server nodes. The current primary processes the requests, updates the backups in the standard way for a primary-backup system and returns a response to the proxies (Figure 3.4.3). Each proxy returns the response to the client (Figure 3.4.4).

The client is responsible for verifying the validity of responses and removing duplicates. Verifying the validity of a response is relatively easy to achieve, as valid responses will be digitally signed by the server that produced them. If active replication is used in the server tier then there may also be invalid responses generated by a malicious intruder who is able to digitally sign them. However, if active replication is used then there will be more unique responses, as determined by the digital signatures used, from uncompromised servers than from compromised servers, allowing the client to choose the majority value as correct.

Removing duplicates is similarly easy, as each response will contain the unique ID that was included in the request. Compromised proxies will not be able to change response IDs as they will be digitally signed by the server along with the rest of the response, and proxies do not have access to the private keys needed to produce valid digital signatures.

3.6 Infrastructure for Proactive Obfuscation (IPO)

The following supporting mechanisms and the interactions between them are summarised in Figure 3.5.

3.6.1 Name Server

A name server is required to allow clients to determine the identities of the current proxy nodes and to verify that responses do originate from one of the current server nodes. It must accept the identities of new proxy and server nodes from the controller

Figure 3.4: Example of Execution of a FORTRESS System

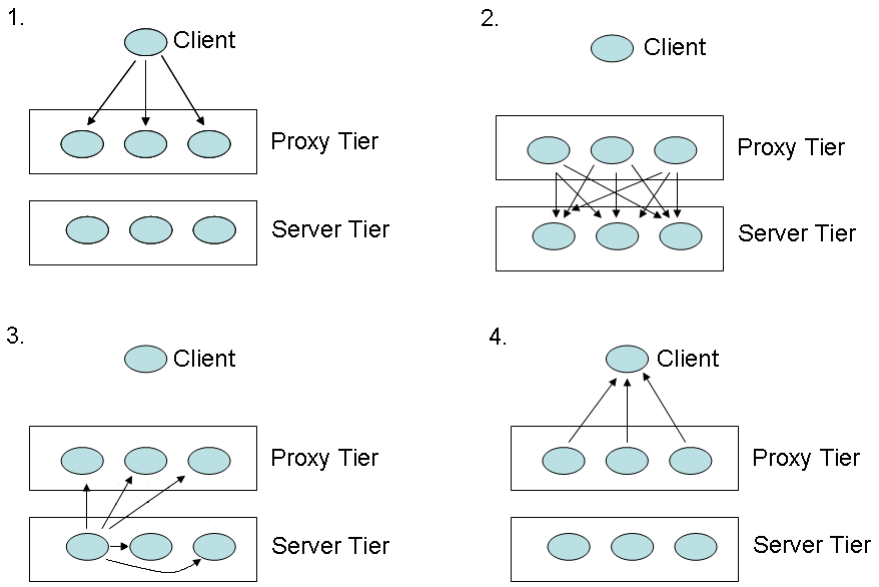
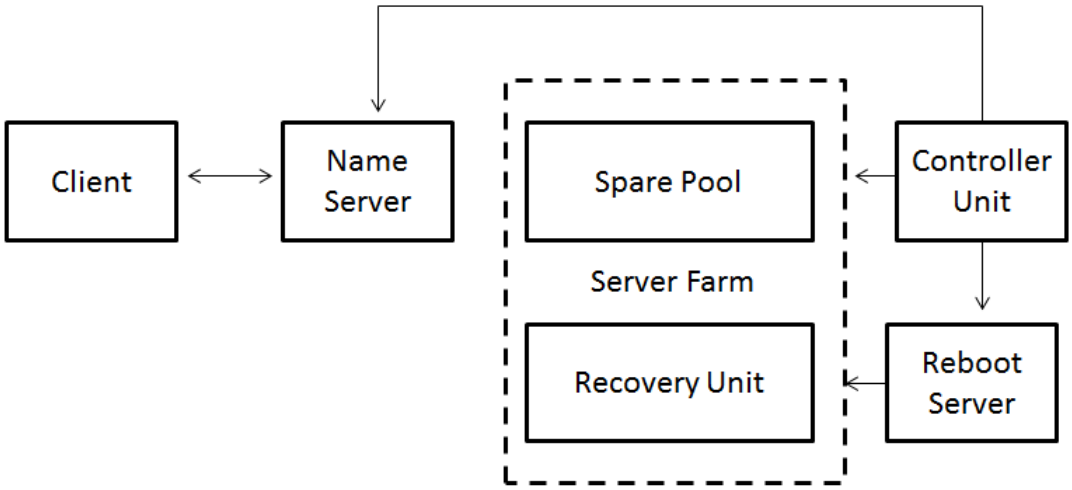


Figure 3.5: Components of The IPO



unit via a private channel. The name server must provide these identities to any client that requests them.

In practice, these identities are likely to consist of the IP addresses of the current proxy nodes, and the public keys of the server nodes for a digital signature scheme.

3.6.2 Reboot Server

The reboot server is a mechanism enabling the physical reboot and refresh of nodes to ensure that compromised nodes do not remain compromised when they are later re-used. The reboot server may also need to ensure that nodes choose new obfuscation keys upon reboot. We assume that the reboot server is capable of instructing a node to reboot in such a way that it will be rebooted even if it is compromised. This may involve the use of a power switch that, on receipt of a message from the reboot server, cuts off and then restores the power to each node after instructing it to shut down. A mechanism of this type is suggested in [45].

3.6.3 Server Farm

The server farm consists of two sub-components, the spare pool and the recovery unit. These sub-components, as well as the nodes that are used in the current proxy and server tiers, are logically rather than physically distinct.

3.6.3.1 Recovery Unit

The recovery unit contains nodes that have been used in a previous server or proxy tier and have not yet finished being rebooted and refreshed.

3.6.3.2 Spare Pool

The spare pool consists of physical machines available to be used as proxy or server nodes. When the reboot server has rebooted and refreshed a node, it is removed from the recovery unit and returned to the spare pool for future use. The controller unit can keep track of which nodes are in which state as detailed in Section 3.6.4.

3.6.4 Controller Unit

The controller unit has four main functions that it performs periodically, based on a timer. Firstly it instructs fresh nodes from the server farm to become proxy or server nodes, and gives the identities of the server nodes to the proxy nodes. Then it instructs the current server nodes to transfer their state to the new server nodes. Next

it provides the name server with the identities of the new proxy and server nodes and instructs the name server to point clients to the new proxy nodes. Finally, it informs the reboot server to reboot the old proxy and server nodes.

The server node identities provided to the name server will not enable the client to send requests to the server tier. Instead, they will allow the client to verify that responses originate from the server tier, using a digital signature scheme.

One key challenge in the design of the controller unit is the need to determine which nodes are in the recovery unit and which nodes are in the spare pool, without risking the controller unit accepting malicious requests from a compromised node. This is especially problematic when handling proxy nodes or server nodes when active replication is used in the server tier, as the system model assumes that these nodes may have been intruded into without the overall system being compromised.

One way in which this challenge can be addressed is as follows:

A hardware module can be installed onto each node. This hardware module has three actions it can perform. Firstly, it can receive ping messages via the secure private network from anyone with access to that network. Secondly, it can respond to ping messages via the secure private network. Thirdly, the hardware module can check if the node it is installed onto is currently powered on and active.

As this module is hardware based and very simply designed, we can assume that is considerably more attack resilient than our proxies or servers, and can hence discount the possibility of an attacker compromising it and using it to send malicious requests to the controller unit. There is a very real possibility of the hardware module being unable to tell the difference between a compromised node and an uncompromised node, however the scheme we propose does not need it to have this capability.

Instead, the controller unit uses the reboot server to send a reboot message that can be guaranteed to cause the machine to reboot. Then, the controller unit simply uses pinging the hardware module to find out whether the machine has finished rebooting.

Another way of addressing the challenge of determining which nodes are in the recovery unit and which nodes are in the spare pool is to have the controller unit ping nodes without the use of a hardware module. This is a riskier proposition, as the controller unit will have to have an open channel of communication to possibly compromised nodes. It may however be possible to have the controller unit refuse to accept any messages from a node that has not yet been physically rebooted by the mechanism discussed in Section 3.6.2, and to make use of properties of the network to ensure that messages from such nodes are not spoofed by other compromised nodes.

3.6.5 Digital Signature Scheme

Each server makes use of digital signatures to allow the client to check that the response was generated by the server. This prevents an attacker from compromising a proxy node and using that proxy to impersonate the server. These digital signatures will have to be unique to each server if an active replication scheme is used on the server tier, as an attacker may intrude nodes and hence determine their private keys without compromising the system. If a passive replication scheme or no replication scheme is used then there will not be a need for unique private keys, as once the attacker has intruded a node the entire system is compromised.

3.6.6 Secure Private Network

A secure private network is required that allows the controller unit to send messages to the name server, reboot server and server farm nodes, and the reboot server to send messages to the server farm nodes. This network must deliver messages within bounded time due to timing assumptions on the speed of reboot.

3.7 Security Concerns Arising from the IPO

3.7.1 Attacks on the Name Server

A fundamental issue with the concept of periodically replacing vulnerable components is that there will always need to be some sort of name server in place to provide clients with the identities of the components that are currently in use. This then gives the attacker the option of targeting the name server rather than the replaceable components.

However, there are two major hindrances to an attacker. Firstly, a compromised name server does not allow the attacker to compromise system state, or breach confidentiality of the whole system, instead only denial of service or impersonation of the nodes in the server tier are possible. Secondly, the name server is likely to be much more resilient to attack than the servers or proxies, as it provides a read-only service to the clients and will be far less complex than most distributed systems, having a very limited range of requests it responds to.

We also note that an attack on the name server is analogous to a man-in-the-middle attack over HTTP [50], or an attack on the DNS server [3]. In each case the client is led to believe that they are receiving responses from the server when in fact they are coming from elsewhere. This means that the danger of an attack on the name server does not open up a new avenue of attack, and instead just reinforces the fact that authentication of the server to the client is required.

The name server is only required to take one pre-set type of request from a client; a request for the current IP addresses and public keys to be returned. The only other request that the name server receives is a request from the controller unit to set the IP addresses and public keys. This request is being sent from a trusted component, and by a trusted network, so can be assumed to be free of malicious code.

3.7.2 Exhaustion of the Spare Pool

A replacement scheme requires that, at the end of each replacement period, there is at least one node available in the spare pool for every node that is used during normal execution of the system. If this number of nodes is not available then either the system will have to stop processing client requests until the full number of nodes are available, or some of the previous nodes will have to continue to be used, giving an attacker a larger time period in which to attack them or make use of previously compromised nodes.

Exhaustion of the spare pool is likely to be avoidable in practice as the relatively cheap cost of hardware, coupled with known replacement periods and reasonably predictable reboot times will enable a sufficiently large server farm to make this risk small. The choice of a system in which processing stops in the event of exhaustion of the spare pool will also make whatever small risk remains a risk of lack of availability, rather than a risk of system compromise.

There is however the possibility that some business situations will require such high availability that a system in which processing stops is a large financial risk. This is mitigated by the likelihood of such situations involving sufficiently large amounts of money to make extra hardware a relatively small expense.

We also note that, as explored in Section 2.12, the majority of costs in software systems come from system development or modification and hardware running costs. Additional nodes in the spare pool will not require any additional software development or modification costs. Nodes will not be required to perform any processing while in the spare pool, so power management strategies can be used to significantly reduce the running costs relative to those of active nodes.

3.7.3 Rebooted Nodes Remaining Compromised

One possible goal for an attacker could be to compromise nodes in such a way that they stay compromised after they have been rebooted. This would allow the attacker to gradually fill up the server farm with compromised nodes, resulting in an increasing likelihood of a compromised node being chosen to be a server node.

The risk of this occurring can be reduced by physically forcing machines to reboot and then reloading the operating system and all software onto each node from read-only

storage during refresh. One example of this is the method presented in [45] where nodes were rebooted from a custom Linux CD which reloaded the operating system and software onto the node. We also note that in some cases an attacker may be able to exploit vulnerabilities to obtain sufficient access to run malicious code on a node, yet still not have the level of access or the level of knowledge of system architecture needed to alter the system in such a way that it remains compromised after reboot.

3.7.4 Attacks on the Controller Unit

The controller unit does not receive messages from any other component other than ping messages generated by secure hardware, or nodes that are guaranteed to have rebooted if one of the schemes described in Section 3.6.4 is used to check if possible nodes have finished rebooting. There is a possibility, if the second scheme is used that a compromised node can send malicious ping messages before it has rebooted. However, the controller unit will reject these messages as coming from an invalid sender, making them highly unlikely to be of use as a possible vector of attack.

The controller unit only sends messages via a secure private network and the software it runs is relatively simple. Hence we discount the possibility of attacks against the controller unit from an external attacker.

3.7.5 Attacks on the Reboot Server

The reboot server only receives messages from the controller unit, and these messages are communicated via a secure private network. The software it runs is relatively simple and all messages it sends are via a private network. Hence, as we are discounting attacks on the controller unit from an external attacker, we also discount attacks on the reboot server from an external attacker.

Chapter 4

Modelling Intrusion Resilience

This chapter presents our strategy for modelling systems and evaluating their intrusion resilience.

We begin by detailing our choice of modelling methodology and how it compares to other possibilities. Then, we consider how to evaluate the effectiveness of attacks on an unreplicated system following the system model presented in Section 3.1 and the attack model presented in Section 3.2. Next, we explore three techniques that can be used to calculate the intrusion resilience from this attack model. We then build on this attack model for an unreplicated system to model attacks against a FORTRESS system, an SMR system and a PB system.

4.1 Choice of Modelling Methodology

When deciding how to evaluate the intrusion resilience of the FORTRESS system, we considered two key possibilities:

1. Formal methods

Here a model is produced of the system expressed in a formal language. System properties and relationships between the properties of different systems can then be proven mathematically, or by model checking. Model checking involves examining every possible state that the system can enter and checking that the desired properties or relationships hold for each of these states.

The main advantage of formal methods is that, when all assumptions made in the model hold, the system properties are proven to hold. However, when it is shown that a particular property does not hold for several systems that are being compared, this does not necessarily give any insight into the relative likelihood of the property being violated for each system. This has led to statistical extensions being made to formal models such as stochastic activity networks (SANs) being developed from Petri-nets.

The main disadvantage is that, even with the optimisations possible with current model checking tools, it is often only possible to perform exhaustive model checking for relatively simple systems as more complex systems can result in a state space explosion that makes them uncheckable with current technology. This can result in system models having to be considerably simplified from the real-world system that they are modelling.

2 Statistical techniques

Statistical techniques involve modelling a system with probabilities attached to events (e.g. the success of a particular malicious request in compromising a server) and then calculating expected values or average times of key system properties. The probabilities used can be experimentally derived from real-world systems, or a range of probabilities can be considered. In either case, the probabilities chosen become an implicit assumption of the system.

Statistical modelling of intrusion resilient systems often involves modelling various stages of an intrusion (e.g. [34, 39]) with probabilities attached to each stage and to actions such as intrusion detection.

Our modelling methodology: We mainly focus on statistical techniques as these techniques allow a quantitative comparison between systems when security properties do not hold with some probability. Our modelling of an intrusion does not involve multiple actions in an intrusion, but instead treats the intrusion into a system as one probabilistic event. We also provide a proof that the FORTRESS system is more intrusion resilient than the SMR system when certain assumptions hold (in Section 5.1), and proofs that the state transfer protocols used in the FORTRESS system satisfy the properties of safety and liveness (in Appendix E).

We note that it may be possible to produce formal proofs of the relationship between the FORTRESS, SMR and PB systems in the cases we have considered statistically. However, this was judged to be a considerably more time consuming process than the statistical analysis performed, with no guarantee of success, and hence the statistical analysis was produced instead.

4.2 Modelling Attacks on a Server

We model the susceptibility of an unreplicated server to attack in the following generic manner:

We define a *unit time-step* with which to measure system lifetime until compromise. When considering systems with proactive recovery or obfuscation, this unit time-step is the period of time in which all servers are recovered.

We define the probability of the server being successfully compromised in unit time-step i as $p(i) = \alpha_i, 0 < \alpha_i \leq 1$.

The expected lifetime until system compromise is then calculated for a range of α_i values.

The rationale behind this modelling technique is a follows:

The attack model presented in Section 3.2 contains an attacker attempting to exploit a known vulnerability in the server by sending malicious requests from malicious clients. Whether a given malicious request succeeds in exploiting the vulnerability depends on whether the attacker has successfully determined the details of the server internals needed to correctly execute malicious code. These details will include the obfuscation key when an obfuscated system is attacked.

One of the main difficulties in modelling intrusion attempts is the lack of information about how an attack will take place. While it is possible to study attacks that have occurred in the past, the attacks that will occur in the future will fall into two categories; exploitation of vulnerabilities of a known type that have not been found in testing, and exploitation of new types of vulnerabilities that have not been encountered before. When an entirely new vulnerability is exploited there is, by definition, no data available about the likelihood of success or the steps involved. Moreover, there is a large variety of known vulnerabilities, with widely differing likelihoods of attacker success and widely differing methodologies of exploitation. This difficulty is further amplified by the fact that the rate at which malicious requests can be processed will be dependent on characteristics of the system being attacked.

Even if we were to obtain reliable data from which to estimate α_i values for past vulnerabilities, we do not know which of these vulnerabilities a given system may have, or whether it contains a new vulnerability with significantly different α_i values. This leads us to consider the system over a wide range of α_i values, in an attempt to cover all likely values.

There is of course the possibility that, for any range chosen, a vulnerability could be discovered that has α_i values outside of this range. We attempt to mitigate this possibility by considering a large enough range of α_i values that values at the higher end give such a small lifetime until intrusion that a system would be compromised immediately and that values at the lower end give an incredibly large lifetime with regard to that expected in real systems. This then allows us to state that any vulnerabilities that fall outside of the range considered for the parameters will either be so serious that no intrusion tolerant or resilient system is likely to withstand them, or so minor that even an unfortified system is likely to withstand them for a long time.

4.3 Obfuscation Schemes

In the preceding section we defined α_i to be the probability of an unreplicated server being compromised in unit time-step i. However, we did not show any relationship between these α values. We now define three obfuscation schemes, and the relationship between α_i values for each scheme.

1. Proactive Obfuscation (PO): Here, the server is replaced at the end of each unit time-step by a new server running code that has been obfuscated with a new obfuscation key.

Here, we define $\alpha_i = \alpha_j, \forall i, j \geq 0$.

This follows from the fact that there is a new obfuscation key chosen for each unit time-step and hence the attacker will not have previously tried any possible keys against this executable.

2 Proactive Recovery (PR): Here, the server is running code that has been initially obfuscated. The server is rebooted at the end of each unit time-step without any new obfuscation taking place.

Here, we define

$$\alpha_i = \begin{cases} \frac{\alpha_0}{1-i\alpha_0} & (1 - i\alpha_0) > \alpha_0 \\ 1 & (1 - i\alpha_0) \leq \alpha_0 \end{cases}$$

This is derived in the following manner:

We assume that the server has been obfuscated with one obfuscation key out of a set of x obfuscation keys, and that the server can process y malicious requests in each unit time-step. Hence, $\alpha_0 = \frac{y}{x}$.

If the server is not compromised during unit time-step 0 then y obfuscation keys have been tried and found to be incorrect. Hence the attacker will be left with $x-y$ possible keys, so $\alpha_1 = \frac{y}{x-y} = \frac{y/x}{1-y/x} = \frac{\alpha_0}{1-\alpha_0}$.

Similarly,

$$\alpha_i = \frac{y}{x - iy} = \frac{y/x}{1 - iy/x} = \frac{\alpha_0}{1 - i\alpha_0}$$

if there were more than y obfuscation keys left to try at the end of unit time-step $i-1$. If there were less than y obfuscation keys left to try at the end of unit time-step $i-1$ then the correct key will be chosen in unit time step i and hence $\alpha_i = 1$.

3 Start-up Only Obfuscation (SO): Here the server is running code that has been initially obfuscated. The server runs continuously without any reboots taking place.

Here α_i is defined the same as when PR is used. We note that this scheme is listed separately from the PR scheme as when we consider replicated systems the PR and SO schemes will not always have the same α_i values.

4.4 Diversity

When we consider the proactive obfuscation case in the preceding section, the server is replaced at the end of each unit time-step by a new server running code that has been obfuscated with a new obfuscation key. As the system makes use of a single server, we can guarantee that, until the entire system is compromised, the attacker does not encounter code that is obfuscated with an obfuscation key that has been discovered in the past.

However, later when we consider systems with more servers it is possible that an obfuscation key may be re-used that has previously been discovered. Whether the attacker can quickly determine that he is facing an obfuscation key he has previously discovered depends on the nature of the obfuscation techniques used, but this may be possible in some cases. Hence we define the *diversity* of the system as being one of the following two cases:

1. When the attacker has no way to determine whether a server is using an obfuscation key he has encountered before we say that the system has *infinite diversity*.

We note that the actual number of obfuscation keys is finite, but as the attacker cannot make use of the fact that one has been re-used, every key chosen can be treated as if it had never been used before, making the pool of new keys effectively infinite.

2 When the attacker can determine whether the server is using an obfuscation key he has encountered before we say that the system has x *diversity* where x is a positive integer.

We note that some attack techniques may not require the exact value of the obfuscation key to be guessed, but instead simply a value near to it. In these cases the amount of diversity may be smaller than the actual key space from which obfuscation keys can be drawn.

4.5 Evaluation Techniques

4.5.1 Markov Chain Techniques

A Markov Chain is defined in the following way (from Chapter 11 of [25]):

We have a set of states $\{S_1, S_2, \ldots, S_n\}$. The Markov Chain starts in one of these states and moves into another state after each step. The probability of moving from state S_i to state S_j is denoted p_{ij} and is independent of any state that the Markov Chain has been in before state S_i.

We note that, as $p_{i,j}$ is independent of any state that the Markov Chain has been in before state S_i, it is also independent of the number of steps that have occurred before reaching state S_i. Hence this technique is only suitable in the cases where we have constant probabilities for each state transition.

A transition matrix is defined as a matrix with n rows and n columns where the entry in position i, j is $p_{i,j}$. This transition matrix is valid for any unit time-step and can be used to generate the expected lifetime using the following absorbing Markov Chain method as detailed in Chapter 11.2 of [25].

1. All of the rows and columns relating to a compromised system compromise are removed from the transition matrix to give a new non-absorbing transition matrix N.

2. We calculate the expected lifetime

$$EL = \sum_{i=1}^{i=n} Q_{1i}$$

where

$$Q = (N - I)^{-1}$$

and I is the identity matrix.

4.5.2 Time Dependent Stochastic Process Techniques

We define a Time Dependent Stochastic Process to differ from a Markov Chain in one key detail: Transition probabilities may depend on previous states, and hence on time.

This allows us to generate a transition matrix for each unit time-step and use the expected value formula $E(X) = \sum_{x=0}^{x=\infty} xp(x)$ where $p(x)$ is the probability that the system will be compromised during unit time-step x, but was not compromised in a

previous unit time-step. This allows us to iterate through the unit time-steps in order, calculating the compromise probability from the transition matrix for each one, and hence the corresponding term of the expected lifetime until system compromise for each one.

The following steps are used to generate the expected lifetime until system compromise.

1. We produce $N0$ the transition matrix for the first unit time-step.
2. We calculate the probability of system compromise during the first unit time step by taking $p(0) = \sum_{j \epsilon C} N0_{1j}$ where C is the set of all states where the system is compromised.
3. We produce $N'0$, the non-absorbing transition matrix for the first unit time step by removing all entries $N0_{ij}, i \epsilon C \cup j \epsilon C$ from $N0$.
4. For unit time-step $y + 1$ we produce the transition matix Ny.
5. We calculate the matrix Q with entries $Q_{ij} = Ny_{ij} \sum_{a=0}^{a=m} N'y - 1_{ai}$ where m is the number of rows in $N'y - 1$.
6. We calculate $p(x) = \sum_{i=0}^{i=n} \sum_{j \epsilon C} Q_{ij}$ where n is the number of rows in Q.
7. We produce $N'y$, the non-absorbing transition matrix for time step $y + 1$ by removing all entries $Ny_{ij}, i \epsilon C \cup j \epsilon C$ from y.
8. We continue this process until y is sufficiently large that $\sum_{x=0}^{x=y} xp(x)$ converges.

4.5.3 Monte Carlo Methods

Monte Carlo methods are a sampling technique that can be used to statistically evaluate the expected lifetime of Markov chains and time dependent stochastic processes (among other models).

The techniques shown in sections 4.5.1 and 4.5.2 can be used to calculate expected lifetimes until system compromise for all of the systems we will consider. However, there are practical considerations that prevent their analytical evaluation in a number of cases.

The use of a transition matrix requires that, when there are n possible states, n^2 transition probabilities are calculated. This can become problematic when the number of states becomes very large.

We note that all of the systems we will consider have a small set of possible configurations with regard to node compromise. However, when systems with finite diversity for node replacement are considered, the state does not just include the configuration

of nodes, it also includes the amount of diversity that has been compromised in the past, and the amount of uncompromised diversity available.

So, when we consider systems with large amounts of diversity, we end up with very large numbers of states. For example, a system with 4 possible configurations and 2^{16} diversity will have approximately 2^{18} possible states and hence approximately 2^{36} transition probabilities.

We also note that, when large but finite amounts of diversity are available, the transition probabilities are time dependent even if α is constant. This occurs because only a small amount of the diversity can be exhausted in each unit time-step, as the attacker can only attack the diverse executables that the nodes are currently using. So, for example, the probability of moving from one node compromised in the first unit time-step to one node compromised in the second unit time-step will be smaller than the probability of moving from one node compromised in the 100th unit time-step to one node compromised in the 101st unit time-step as there are likely to be more previously compromised executables that could be chosen after one hundred unit time-steps than after one unit time-step.

Hence, it is impractical to analytically calculate the expected lifetimes until system compromise of systems when large but finite diversity is assumed, and Monte-Carlo methods are used to evaluate the time-dependent stochastic processes instead. This involves repeatedly simulating an attack on the system and using the results as a random sample to estimate the expected lifetime until system compromise. The procedure is as follows:

1. We define d to be the amount of diversity available and c to be the amount of diversity that has been compromised so far, and α to be the probability of a node being intruded in the current unit time-step.

2. Every node is randomly assigned as being previously compromised or not previously compromised. This is achieved by picking a random integer $x_i \epsilon [0, d]$ for each node i and declaring node i to be compromised if $x_i < c$.

3. For each node i that is previously compromised a random decimal $z_i \epsilon [0, 1]$ is chosen.

4. If $z_i < \alpha$ then node i becomes compromised and c is incremented.

5. The configuration of nodes compromised and uncompromised at this point is checked against the system compromise conditions.

6. If the configuration is a compromise configuration then the number of unit time-steps the system survived is recorded.

7. If the configuration is not a compromise configuration then the procedure is repeated for the next unit time-step.

4.6 Modelling Attacks on the FORTRESS System

We begin by extending the attack model in Section 3.2 to address the FORTRESS system as defined in Section 3.5. This is followed by a derivation of the probabilities of compromising the server in the SO and PR cases. Finally, we present the transition matrices for the FORTRESS system under the obfuscation schemes and levels of diversity considered, and the derivation of these transition matrices.

4.6.1 Attack Model

The attacker is aware of a vulnerability that is present in all of the proxy nodes, and a vulnerability that is present in all of the server nodes. This may be a common vulnerability to the proxy nodes and server nodes or a distinct vulnerability in each. As the goal of the attacker is to compromise system integrity, the attacker wishes to compromise one or more server nodes. This can be attempted in two ways:

1. Indirect attack - The attacker uses malicious clients to send malicious requests to the proxy nodes. These requests are not designed to target a vulnerability on the proxy nodes, but instead to be forwarded to the server and target a server vulnerability.

2. Direct attack - The attacker uses malicious clients to send malicious requests to the proxy nodes. These requests are designed to target a vulnerability on the proxy nodes. Once one or more proxy nodes are compromised, the attacker can use these proxy nodes to send malicious requests to the server.

We note that in many cases indirect attacks may not be possible, or may take longer to execute then direct attacks. This is discussed in detail in Section 5.3 and motivates the following probability definitions:

- α_i is the probability of a proxy being compromised in unit time-step i.

This definition follows the same rationale as for the single server case in Section 4.2.

- α_i is the probability of a server being compromised in unit time-step i, if at least one proxy was compromised at the start of unit time-step i.

This definition follows the same rationale as for the single server case in Section 4.2, as the server is being directly sent malicious requests by one or more compromised proxy nodes.

- $\kappa\alpha_i$ is the probability of a server being compromised in unit time-step i, if no proxy nodes were compromised at the start of unit time-step i.

Here we define $0 \leq \kappa \leq 1$ to be the *indirect attack coefficient*. The indirect attack coefficient is a measure of how much less likely an indirect attack is to succeed then a direct attack.

We assume that an attacker will simultaneously launch direct attacks against the proxy nodes and indirect attacks against the server nodes until either the server is compromised or one or more proxy nodes is compromised. Once a proxy node is compromised the attacker will then launch direct attacks against the server.

We also expand the compromise conditions to include the case where an attacker has managed to compromise every proxy node. While this does not allow the attacker to compromise the integrity of the system state, it does allow the attacker to continue to directly attack the server while having complete control of any path between a legitimate client and the server. This condition is sufficiently undesirable that we consider the FORTRESS system to have been compromised if it is reached.

4.6.2 Server Compromise Probability in the SO and PR Cases

When considering the FORTRESS system with start-up only obfuscation we will also consider the fact that the attacker can only start to exhaust possible keys for the server when one or more proxies have been compromised. Hence we will define α_{is} as the probability of the adversary successfully intruding the primary server in unit time step i where:

$\alpha_{0s} = \alpha_0$

and

$$\alpha_{is} = \begin{cases} p(0,i-1)\alpha_0 + (p(1,i-1) + p(2,i-1) + p(3,i-1))\frac{\alpha_{i-1s}(1-(i-1)\alpha_0)}{1-i\alpha_0} & (1-i\alpha_0) > \alpha_0 \\ p(0,i-1)\alpha_0 + p(1,i-1) + p(2,i-1) + p(3,i-1) & (1-i\alpha_0) \leq \alpha_0 \end{cases}$$

where $p(c,i)$ is the probability that c proxy nodes have been compromised and the primary server has not been compromised at the start of unit time-step i.

We derive these probabilities as follows:

When we reach time-step i there are two possibilities. Firstly, no proxy has been compromised by the start of unit time-step $i-1$. In this case we start unit-time step i with no keys exhausted for the server. The first term, $p(0,i-1)\alpha_0$, is simply the probability of no proxy nodes having been intruded by the start of unit time-step $i-1$ multiplied by the probability of successful intrusion if no keys had been exhausted for the server by the start of unit time-step i.

The second possibility is that one or more proxy nodes had been successfully intruded by the start of unit time-step i. In this case we know that some keys will have been

exhausted for the server due to malicious requests launched in unit time-step $i-1$. The second term $(p(1,i-1)+p(2,i-1)+p(3,i-1))\frac{\alpha_{i-1s}(1-(i-1)\alpha_0)}{(1-i\alpha_0)}$ is the probability of one or more proxy nodes having been successfully intruded by the start of the unit time-step $i-1$ multiplied by the server intrusion probability in unit time-step $i-1$ and the change in intrusion probability caused by keys being exhausted in unit time-step $i-1$.

The change in intrusion probability caused by keys being exhausted in until time step $i-1$ is calculated as follows:

The probability of intrusion during unit time-step $i-1$ into a node that had been receiving malicious requests from the beginning of unit time-step 0 is $\frac{\alpha_0}{1-(i-1)\alpha_0}$. The probability of intrusion during unit time-step i into a node that had been receiving malicious requests from the beginning of unit time-step 0 is $\frac{\alpha_0}{1-i\alpha_0}$. Hence, the change in probability of intrusion from unit time-step $i-1$ to unit time-step i is $\frac{\alpha_0}{1-i\alpha_0}/\frac{\alpha_0}{1-(i-1)\alpha_0} = \frac{(1-(i-1)\alpha_0)}{(1-i\alpha_0)}$ when $1-i\alpha_0 > \alpha_0$. When $1-i\alpha_0 \leq \alpha_0$ the number of remaining keys is less than or equal to the number of keys that can be exhausted in one unit time-step and hence the probability of an attack successfully intruding into the server is 1.

We note that this change in probability assumes that the server has been receiving malicious requests from the start of unit time-step 0 and hence may result in a small underestimation of the expected lifetime of the system. Hence we will be calculating a lower bound for the expected lifetime of the FORTRESS system with the SO or PR obfuscation scheme.

4.6.3 Transition Matrices

The SO and PR cases. These two cases are treated identically for the FORTRESS system.

We define the following states:

0. No proxy nodes are compromised and the server node is uncompromised.
1. One proxy node is compromised and the server node is uncompromised.
2. Two proxy nodes are compromised and the server node is uncompromised.
3. Three proxy nodes are compromised and the server node is uncompromised.
4. No proxy nodes are compromised and the server node is compromised.
5. One proxy node is compromised and the server node is compromised.
6. Two proxy nodes are compromised and the server node is compromised.
7. Three proxy nodes are compromised and the server node is compromised.

We note that states 4-8 are the states in which the system is compromised.

The probabilities for the FORTRESS system with SO or PR are time dependent, giving a time-dependent stochastic process with transition matrix

$$N = \begin{bmatrix} a^3c & 3\alpha_i a^2 c & 3\alpha_i^2 ac & \alpha_i^3 c & a^3\kappa\alpha_{is} & 3\alpha_i a^2\kappa\alpha_{is} & 3\alpha_i^2 a\kappa\alpha_{is} & \alpha_i^3\kappa\alpha_{is} \\ 0 & a^2 b & 2\alpha_i ab & \alpha_i^2 b & 0 & a^2\alpha_{is} & 2\alpha_i a\alpha_{is} & \alpha_i^2\alpha_{is} \\ 0 & 0 & ab & \alpha_i b & 0 & 0 & a\alpha_{is} & \alpha_i\alpha_{is} \\ 0 & 0 & 0 & b & 0 & 0 & 0 & \alpha_i\alpha_{is} \\ 0 & 0 & 0 & 0 & a^3 & 3\alpha_i a^2 & 3\alpha_i^2 a & \alpha_i^3 \\ 0 & 0 & 0 & 0 & 0 & a^2 & 2\alpha_i a & \alpha_i^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & a & \alpha_i \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$a = 1 - \alpha_i$$

$$b = 1 - \alpha_{is}$$

$$c = 1 - \kappa\alpha_{is}$$

$$\alpha_i = \begin{cases} \frac{\alpha_0}{1-i\alpha_0} & (1 - i\alpha_0) > \alpha_0 \\ 1 & (1 - i\alpha_0) \leq \alpha_0 \end{cases}$$

$$\alpha_{0s} = \alpha_0$$

$$\alpha_{is} = \begin{cases} p(0, i-1)\alpha_0 + (p(1, i-1) + p(2, i-1) + p(3, i-1))\frac{\alpha_{i-1s}(1-(i-1)\alpha_0)}{1-i\alpha_0} & (1 - i\alpha_0) > \alpha_0 \\ p(0, i-1)\alpha_0 + p(1, i-1) + p(2, i-1) + p(3, i-1) & (1 - i\alpha_0) \leq \alpha_0 \end{cases}$$

where $p(m, i-1)$ is the probability that m proxy nodes were compromised by unit time-step i.

The FORTRESS system with the PO obfuscation scheme and infinite diversity.

Here we define two states

0. The system is not compromised and hence all proxy and server nodes are replaced by uncompromised nodes.

1. The system is compromised.

The probabilities for the FORTRESS system with PO and infinite diversity are time independent, giving a Markov Chain with transition matrix

$$N = \begin{bmatrix} (1-\alpha_i)^3 + (1-\kappa\alpha_i) & 1 - ((1-\alpha_i)^3 + (1-\kappa\alpha_i)) \\ 0 & 1 \end{bmatrix}$$

$$\alpha_i = \alpha_j = \alpha_{is} = \alpha_{js}, \forall i, j \geq 0$$

The FORTRESS system with the PO obfuscation scheme and finite diversity.

Here we define d to be the total amount of diversity available and c to be the amount of diversity already compromised. We can then define a transition matrix that is dependent on c rather than time and then define the probability of each possible change in c for each starting state. This allows us to evaluate the expected lifetime in the same way as a time-dependent stochastic process, except that c is tracked rather than time.

The states are identical to those used in the SO and PR cases.

The transition matrix is

$$N = \begin{bmatrix}
\frac{bD_1D_2D_3}{D_0} & \frac{3bD_1D_2c}{D_0} & \frac{3bD_1cc_1}{D_0} & \frac{bcc_1c_2}{D_0} & \frac{\kappa\alpha_{is}D_1D_2D_3}{D_0} & \frac{3\kappa\alpha_{is}cD_1D_2}{D_0} & \frac{3\kappa\alpha_{is}cc_1D_1}{D_0} & \frac{\kappa\alpha_{is}cc_1c_2}{D_0} \\
\frac{aD_1D_2D_3}{D_0} & \frac{3aD_1D_2c}{D_0} & \frac{3aDcc_1}{D_0} & \frac{acc_1c_2}{D_0} & \frac{\alpha_{is}D_1D_2D_3}{D_0} & \frac{3\alpha_{is}cD_1D_2}{D_0} & \frac{3\alpha_{is}cc_1D_1}{D_0} & \frac{\alpha_{is}cc_1c_2}{D_0} \\
\frac{aD_1D_2D_3}{D_0} & \frac{3aD_1D_2c}{D_0} & \frac{3aDcc_1}{D_0} & \frac{acc_1c_2}{D_0} & \frac{\alpha_{is}D_1D_2D_3}{D_0} & \frac{3\alpha_{is}cD_1D_2}{D_0} & \frac{3\alpha_{is}cc_1D_1}{D_0} & \frac{\alpha_{is}cc_1c_2}{D_0} \\
\frac{aD_1D_2D_3}{D_0} & \frac{3aD_1D_2c}{D_0} & \frac{3aDcc_1}{D_0} & \frac{acc_1c_2}{D_0} & \frac{\alpha_{is}D_1D_2D_3}{D_0} & \frac{3\alpha_{is}cD_1D_2}{D_0} & \frac{3\alpha_{is}cc_1D_1}{D_0} & \frac{\alpha_{is}cc_1c_2}{D_0} \\
0 & 0 & 0 & 0 & \frac{D_1D_2D_3}{D_0} & \frac{3cD_1D_2}{D_0} & \frac{3cc_1D_1}{D_0} & \frac{cc_1c_2}{D_0} \\
0 & 0 & 0 & 0 & \frac{D_1D_2D_3}{D_0} & \frac{3cD_1D_2}{D_0} & \frac{3cc_1D_1}{D_0} & \frac{cc_1c_2}{D_0} \\
0 & 0 & 0 & 0 & \frac{D_1D_2D_3}{D_0} & \frac{3cD_1D_2}{D_0} & \frac{3cc_1D_1}{D_0} & \frac{cc_1c_2}{D_0} \\
0 & 0 & 0 & 0 & \frac{D_1D_2D_3}{D_0} & \frac{3cD_1D_2}{D_0} & \frac{3cc_1D_1}{D_0} & \frac{cc_1c_2}{D_0}
\end{bmatrix}$$

$$D_0 = d(d-1)(d-2)$$

$$D_1 = (d-c)$$

$$D_2 = (d-c-1)$$

$$D_3 = (d-c-2)$$

$$a = 1 - \alpha_{is}$$

$$b = 1 - \kappa\alpha_{is}$$

$$c_1 = c - 1$$

$$c_2 = c - 2$$

The probabilities of each change in c for each starting state are shown in the following matrix:

$$C = \begin{array}{c} \\ c = c \\ c = c+1 \\ c = c+2 \\ c = c+3 \end{array} \begin{bmatrix} a^3 & a^2 & a & 1 & a^3 & a^2 & a & 1 \\ 3\alpha_i a^2 & 2\alpha_i a & \alpha_i & 0 & 3\alpha_i a^2 & 2\alpha_i a & \alpha_i & 0 \\ 3\alpha_i^2 a & \alpha_i^2 & 0 & 0 & 3\alpha_i^2 a & \alpha_i^2 & 0 & 0 \\ \alpha_i & 0 & 0 & 0 & \alpha_i & 0 & 0 & 0 \end{bmatrix}$$

$$a = 1 - \alpha_i$$

We note that system compromise occurs if either the system reaches any of the compromise states in the transition matrix, or a c change occurs such that the sum of the change in c and the number of nodes compromised at the start of the unit time-step is equal to 3, as this is the case when all three proxy servers are compromised.

4.7 Modelling Attacks on the PB System

The PB system consists of one primary server that processes requests and propagates the results to backup servers that can take over processing if the primary server crashes. The fact that only one server is processing requests at any given time means that an attacker can only send malicious requests to that server. Similarly, if that server is compromised then the attacker can compromise the system state that will be propagated to the backups, and hence compromise the entire system.

This allows us to model the system as identical to that in Section 4.2.

We define two states

0. The server is not compromised.

1. The server is compromised.

The probabilities for the PB system with PO are time independent, giving a Markov Chain with transition matrix

$$N = \begin{bmatrix} 1-\alpha_i & \alpha_i \\ 0 & 1 \end{bmatrix}$$

$$\alpha_i = \alpha_j, \forall i, j \geq 0$$

This transition matrix can be evaluated analytically to give the expected lifetime until system compromise.

The probabilities for the PB system with SO or PR are time dependent, giving a time-dependent stochastic process with transition matrix

$$N = \begin{bmatrix} 1-\alpha_i & \alpha_i \\ 0 & 1 \end{bmatrix}$$

$$\alpha_i = \begin{cases} \frac{\alpha_0}{1-i\alpha_0} & (1-i\alpha_0) > \alpha_0 \\ 1 & (1-i\alpha_0) \leq \alpha_0 \end{cases}$$

The psuedo-code for the evaluation of this time-dependent stochastic process is presented in Appendix A.

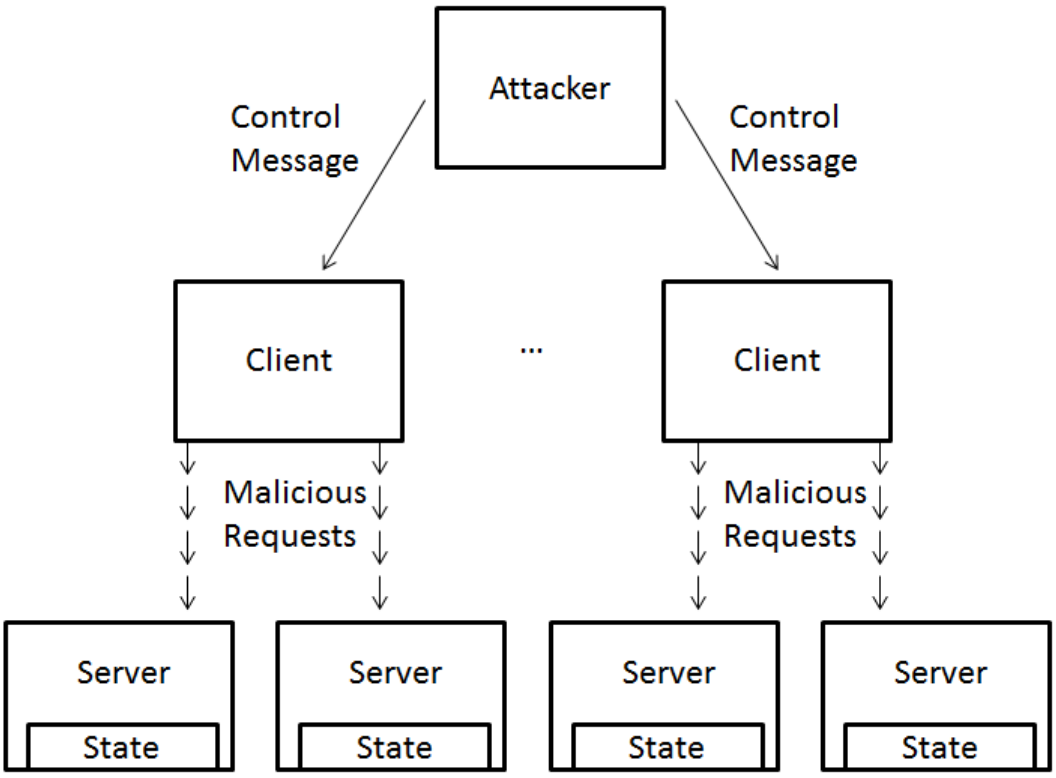
4.8 Modelling Attacks on the SMR System

The SMR system consists of 4 server nodes that process client requests. These server nodes use an active replication scheme that ensures that system integrity is maintained when 1 of those server nodes is compromised. Hence, the system integrity is compromised when 2 or more server nodes are compromised at the same time. All 4 server nodes are publicly accessible and involved in processing, allowing an attacker to send malicious requests to all four.

The SMR system is illustrated in Figure 4.1.

When we consider an SMR system using the PR or PO obfuscation schemes we note that there are several patterns of server reboot or replacement that may be used. The standard pattern that we consider is when the system replaces rebooting servers with spares that take their system state from the all nodes, including the exiting one, as in [66]. This results in a situation analogous to that found in the PB and FORTRESS cases, which we name *en-masse replacement*. However, many schemes exist that involve the reboot of servers during the unit time-step with the recovering server receiving its state only from the non-rebooting servers.

Figure 4.1: The SMR System



There are generally two ways in which the non-rebooting servers in an system can provide the service state to a rebooted server:

1. CP1: A past checkpoint from before the server rebooted and a sequence of client requests which includes those that the rebooted node could not receive while being rebooted.

2. CP2: A checkpoint generated by the other servers that includes client requests that the rebooted replica missed while rebooting and the first request to resume processing with.

The key difference between these two checkpointing methods is that CP1 ensures that every server processes every request, while CP2 allows some requests to be processed by 3 of the 4 servers, while the other server is rebooting.

We note that, for modelling purposes, an SMR system using the SO obfuscation scheme and an SMR system using the PR obfuscation scheme with en-masse replacement are treated identically. This is due to the fact that an attacker who has once compromised a node will be able to compromise it again at any point in the future.

Furthermore, an SMR system using the PR obfuscation scheme and checkpointing method *CP1* will be treated identically to an SMR system using the SO obfuscation scheme (and hence the SMR system using the PR obfuscation scheme and en-masse replacement). This is due to two characteristics of the SMR system using the PR

obfuscation scheme and checkpointing method *CP1.* Firstly, all requests (and hence all malicious requests) are processed by every node. Secondly, the use of the PR obfuscation scheme results in an attack that would be successful in one unit time-step being successful in any unit time-step. Hence, if an attack that would have been successful against a node is launched while that node is rebooting, then the attack will still eventually be successful, and the attacker can then re-use it at any point in the future.

Checkpointing method *CP2* does not guarantee that every request will be processed by every node, so the SMR system using the PR obfuscation scheme and checkpointing method *CP2* is modelled separately.

When considering the SMR system using the PO obfuscation scheme, en-masse replacement, checkpointing method CP1 and checkpointing method CP2 are considered separately. We model en-masse replacement and a special case of CP2, and then show that the expected lifetimes of all other cases of CP1 and CP2 fall between these values.

The transition matrices for the SMR system are as follows:

The SMR system with the SO obfuscation scheme (or the PR obfuscation scheme with en-masse replacement or checkpointing method CP1):

Here we define states 0 to 4 to consist of the corresponding number of server nodes currently compromised.

The probabilities are time-dependent giving a time-dependent stochastic process with transition matrix

$$N = \begin{bmatrix} (1-\alpha_i)^4 & 4(1-\alpha_i)^3\alpha_i & 6(1-\alpha_i)^2\alpha_i^2 & 4(1-\alpha_i)\alpha_i^3 & \alpha_i^4 \\ 0 & (1-\alpha_i)^3 & 3(1-\alpha_i)^2\alpha_i & 3(1-\alpha_i)\alpha_i^2 & \alpha_i^3 \\ 0 & 0 & (1-\alpha_i)^2 & 2(1-\alpha_i)\alpha_i & \alpha_i^4 \\ 0 & 0 & 0 & (1-\alpha_i) & \alpha_i \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

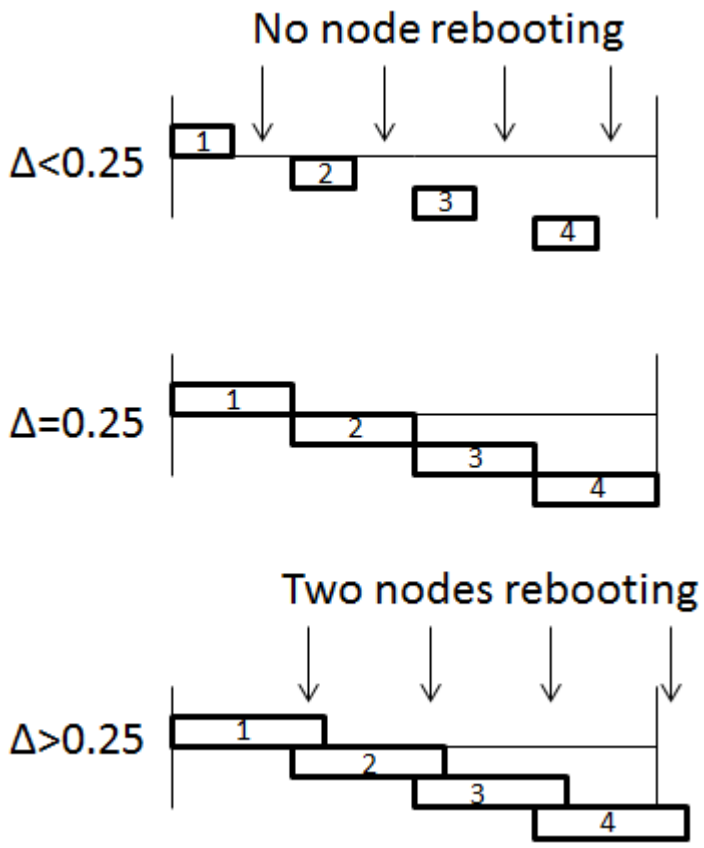
$$\alpha_i = \begin{cases} \frac{\alpha_0}{1-i\alpha_0} & (1-i\alpha_0) > \alpha_0 \\ 1 & (1-i\alpha_0) \leq \alpha_0 \end{cases}$$

The SMR system with the PR obfuscation scheme with checkpointing method CP2:

The transition matrix is identical to the transition matrix for the SMR system with the SO obfuscation scheme, however α_i is defined as:

$$\alpha_i = \begin{cases} \frac{\alpha'_0}{1-i\alpha'_0} & (1-i\alpha'_0) > \alpha'_0 \\ 1 & (1-i\alpha'_0) \leq \alpha'_0 \end{cases}$$

Figure 4.2: Reboot Intervals as $\triangle$ Varies



$$\alpha'_0 = 1 - (1 - \alpha_0)^{1-\triangle}$$

The α_i values are derived in the following manner:

When the checkpointing method *CP2* is applied, some requests will be not be processed by every node. This reduction in the number of malicious requests processed by each node will reduce the probability of the system being compromised in each unit time-step. If we take $\triangle$ to be the proportion of the unit time-step that each node spends on rebooting then the probability of system compromise will be $\alpha' = 1 - (1 - \alpha)^{1-\triangle}$ where α is the probability of compromising a node if *CP2* was not used. We note that $\triangle \leq 0.25$ as larger $\triangle$ values will result in two or more nodes having overlapping reboot periods, invalidating the intrusion tolerance assumptions of the system. This is illustrated in Figure 4.2.

The SMR system with the PO obfuscation scheme with en-masse replacement and infinite diversity:

The states are

0. The system is not compromised and hence all servers are replaced by uncompromised servers.

1. The system is compromised.

The transition matrix is

$$N = \begin{bmatrix} (1-\alpha_i)^4 + 4\alpha_i(1-\alpha_i)^3 & 1 - ((1-\alpha_i)^4 + 4\alpha_i(1-\alpha_i)^3) \\ 0 & 1 \end{bmatrix}$$

The SMR system with the PO obfuscation scheme with en-masse replacement and finite diversity:

Here we define d to be the total amount of diversity available and c to be the amount of diversity already compromised. We can then define a transition matrix that is dependent on c rather than time and then define the probability of each possible change in c for each starting state. This allows us to evaluate the expected lifetime in the same way as a time-dependent stochastic process, except that c is tracked rather than time.

We define the states as

0. No nodes compromised

1. One node compromised.

2. System compromised.

The transition matrix is

$$N = \begin{bmatrix} \frac{(a^4+4\alpha_i a^3)D_1D_2D_3D_4}{D_0} & \frac{4(a^4+4\alpha_i a^3)cD_1D_2D_3}{D_0} & 1-S_1 \\ \frac{(1-\alpha_i)^3D_1D_2D_3D_4}{D_0} & \frac{4(1-\alpha_i)^3cD_1D_2D_3}{D_0} & 1-S_2 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\alpha_i = \alpha_j = \alpha_{is} = \alpha_{js}, \forall i, j \geq 0$$

$$D_0 = d(d-1)(d-2)(d-3)$$

$$D_1 = (d-c)$$

$$D_2 = (d-c-1)$$

$$D_3 = (d-c-2)$$

$$D_4 = (d - c - 3)$$

$$S_1 = (a^4 + 4\alpha_i a^3)D_1 D_2 D_3 D_4 / D_0 + 4(a^4 + 4\alpha_i a^3)cD_1 D_2 D_3 / D_0$$

$$S_2 = (1 - \alpha_i)^3 D_1 D_2 D_3 (D_4 + 4)/D_0$$

$$a = 1 - \alpha_i$$

The probabilities of each change in c for each transition between states in which the overall system is not compromised are shown in the following matrix:

$$C = \begin{array}{c} \\ c = c \\ c = c + 1 \end{array} \begin{array}{c} \begin{array}{cc} c = c & \end{array} \\ \begin{bmatrix} (1 - \alpha_i)^4 & (1 - \alpha_i)^3 \\ 4\alpha_i(1 - \alpha_i)^3 & - \end{bmatrix} \end{array}$$

The SMR system with the PO obfuscation scheme with checkpointing method CP2 and infinite diversity:

As a different node is rebooting in each of 4 subsections of the unit time-step, the transition matrix is defined for the transition between these subsections. A state vector (a, b, c, d, e) is defined where a is the number of the current subsection, and b, c, d, e correspond to the nodes 1,2,3,4, each holding the value 0 if the corresponding node is not compromised and 1 if the corresponding node is compromised. This results in a transition matrix of size 64. Hence, for ease of presentation we explain how this transition matrix is calculated rather than present it.

1. The probability of node x being successfully compromised in subsection y is calculated as $1 - (1 - \alpha)^{0.25}$ if $x \neq y$ and 0 if $x = y$.

2. Transition probabilities are calculated between all states, using the probabilities from step 1 and the fact that each node moves to the uncompromised state in the corresponding numbered subsection and begins the next subsection in that state.

3. System compromise states are identified as those in which two or more nodes are compromised.

The psuedo-code for the Monte-Carlo simulation used to evaluate this scheme is presented in Appendix A.5.

The SMR system with the PO obfuscation scheme with checkpointing method CP2 and finite diversity:

This system uses the same model as the preceding case, except that each node is not guaranteed to be uncompromised after the unit time-step in which it reboots. Instead, each rebooting node has the probability $(d - c + e)/d$ of being uncompromised after reboot, where d is the available diversity, c is the total number of servers compromised so far, and e is the number of compromised servers currently in use.

Chapter 5

Comparison of Intrusion Resilience with Other Systems

This chapter compares the intrusion resilience of the FORTRESS, SMR and PB systems, using the models defined in Chapter 4. We begin by presenting a general result about the relationship between the performance of the SMR and FORTRESS systems using the SO or PR obfuscation schemes when indirect attacks are not possible against the FORTRESS system. This is followed by a numerical comparison of the expected lifetimes of the three systems when indirect attacks are not possible against the FORTRESS system.

Next, we discuss the conditions that make the possibility of indirect attacks likely and the conditions that make the possibility of indirect attacks unlikely. This followed by a comparison of the expected lifetimes of the three systems when indirect attacks are possible against the FORTRESS system.

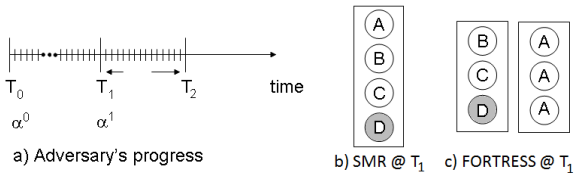
We note that a summary of some of the results presented in this Section appeared in [16].

5.1 General Result

Here we analytically compare the intrusion-resilience of the SMR and FORTRESS systems using the SO or PR obfuscation schemes when indirect attacks are not possible against the FORTRESS system. This comparison follows the structure and arguments of that presented in [21], where we considered the FORTRESS system with different attack assumptions.

The unit time-steps are assumed to be of the same duration for the two systems and nodes are assumed to be rebooted instantly (if the PR obfuscation scheme is used). This assumption makes the analysis independent of whether nodes are rebooted or replaced with new nodes using the same obfuscation key. Moreover, for *both* systems,

Figure 5.1: Lifetime comparison.



we assume that nothing occurs to remove vulnerabilities (such as system patches being applied in response to a successful intrusion) during the period considered.

Since exploiting a node will require the attacker to determine the obfuscation key, and nodes are not re-obfuscated, a node that has been exploited and then rebooted can be exploited again rapidly. Hence, we will consider the exploitation of a previously exploited and then rebooted node to be instantaneous. This means that once a node of type X is exploited, it remains compromised subsequently despite the fact that it is being rebooted. This allows us to treat the SO and PR obfuscation schemes identically.

Finally, once an adversary identifies an exploitable vulnerability, he is assumed to exploit it instantaneously.

Both systems start at time T_0 when the adversary is assumed to know nothing of the vulnerabilities that might be present in nodes. Of interest are two instances (see Figure 5.1(a)): T_1 when the adversary compromises the first node and T_2 when the adversary compromises a second node. For S_0, $T_2 - T_0$ is the time to fatal intrusion. For simplicity, we focus on the time left until system compromise after T_1 and the mean lifetime after T_1 will be denoted as ℓ.

We assume that four obfuscation keys have been chosen and used to create four obfuscated versions of the server for the SMR case and three obfuscated versions of the proxy and one obfuscated version of the server for the FORTRESS case. We label the nodes using these obfuscation keys as A, B, C, D.

We model the adversary by defining α_X as the probability that he successfully determines the obfuscation key needed to exploit a vulnerability in node X during a unit time-step, where $X \epsilon \{A, B, C, D\}$, given that the attacker can directly send requests to node X (see Section 4.6.1 for an explanation of direct and indirect attacks).

Thus, α_X is the adversary's success rate for nodes of type X if he has direct access. Values of α_X for later periods are at least as large as earlier ones. When node X is accessible from T_0, we denote the values of α_X at T_0 and T_1 as α_X^0 and α_X^1 respectively. Note that $\alpha_X^0 \leq \alpha_X^1$, and when nodes have no obvious deterministic bugs, $(T_1 - T_0)$ is large and $\alpha_X^0 < \alpha_X^1$.

5.1.1 SMR

With no loss of generality, we let node D be the most vulnerable (relative to the scheme the attacker is using to determine obfuscation keys) and hence it is compromised at T_1 (Figure 5.1(b)). The adversary can now do the following from T_1 onwards: attack the server nodes both as a client (as he was doing before T_1) *and* also as a controller of the compromised server replica of type D. However, we have made the general assumption that, by the use of distributed attacks, the adversary has been able to launch attacks as fast as the nodes can process them (see Section 2.7). Hence, attacks originating from a server do not increase the speed with which the adversary can attack.

There may however be some advantage in launching attacks from a server replica, but we will assume, optimistically from the *SMR* system's perspective, that the presence of the compromised replica does not increase the success rate. We note that this optimistic assumption can only lead to an over-estimation of ℓ_0 - the metric measuring the intrusion-resilience of the *SMR* system.

For analytical simplicity, we will assume for both the systems that the adversary's success rate is constant at α_X^1 for all periods in $[T_1,\ T_2]$ and for all $X \in \{A, B, C\}$. In reality, however, his success rate will increase over time, and the effect of this simplification can be ignored as we are only comparing the system lifetimes.

The probability that the attacker compromises at least one correct node in the SMR system during any given period after T_1 is:

$$\gamma_0 = 1 - (1 - \alpha_A^1)(1 - \alpha_B^1)(1 - \alpha_C^1). \tag{5.1}$$

This shows us that the number of rounds in the interval $(T_2 - T_1)$ is geometrically distributed with parameter γ_0 and hence the probability that $(T_2 - T_1)$ has i, $i \geq 0$, periods is $\gamma_0 \times (1 - \gamma_0)^i$. Thus,

$$\ell_0 = E(T_2 - T_1) = \frac{1 - \gamma_0}{\gamma_0}. \tag{5.2}$$

5.1.2 FORTRESS

With no direct access to server nodes, the adversary attacks the proxies until T_1. Consider the case where the most vulnerable node D is *not* selected to be used in the proxy tier which therefore will contain nodes A, B and C. By the time the adversary compromises one of these less vulnerable proxies, he could have compromised both this node and node D in the SMR system. That is, the SMR system is compromised at the same time as T_1 for the FORTRESS system. So we will only analyse the alternative case as shown in Figure 5.1(c).

At T_1, the adversary's use of a compromised proxy is modelled as follows. The compromised proxy attacks the primary server node at the same rate as the proxy was originally attacked. This has no effect on the continued attacks on the other proxies; the attacker and the compromised proxy attack in parallel. The original adversary, as before, attacks the proxies and has no direct access to servers; the compromised proxy attacks server nodes but not the other proxy nodes.

As with the SMR system, we will retain the same simplification that the success rate for the A-type server nodes remain constant for all periods in $[T_1, T_2]$ at α_A.

System compromise occurs at the earliest occurrence of either (i) the adversary successfully exploiting vulnerabilities in all non-compromised proxies or (ii) the compromised proxy exploiting the primary server node. If (i) were to occur earlier, the FORTRESS system is obviously more resilient than the SMR system: between the compromise of the second and third proxies, the FORTRESS system works correctly while the SMR system would be fatally intruded. So, we will let (ii) be the earlier event. Note that only one server node can be attacked at any given time, as only one server node will be processing client requests. The probability that the adversary replica compromises the server during any given period after T_1 is:

$$\gamma_2 = 1 - [(1 - \alpha_A)], \tag{5.3}$$

As in (5.2),

$$\ell_2 = E(T_2 - T_1) = \frac{1 - \gamma_2}{\gamma_2}. \tag{5.4}$$

Claim. When using the SO or PR obfuscation schemes , the FORTRESS system is more resilient than the SMR system, provided that an adversary cannot intrude a server without having compromised at least one proxy.

Proof. By (5.1) and (5.3), $\gamma_2 < \gamma_0$; by (5.4) and (5.2), $\ell_2 > \ell_0$.

Corollary. If the SMR system is intrusion resilient for a set of adversaries, then the FORTRESS system is always more intrusion-resilient against those adversaries, provided that the claim above holds.

5.2 Comparison Without Indirect Attacks

Here we consider the SMR, PR and FORTRESS systems first with the PR and SO obfuscation schemes and then with the PO obfuscation scheme. When the SMR system with the PO obfuscation scheme is considered we calculate expected lifetimes for each checkpoint scheme. The first, $SMRPO$, is the case in which en-masse replacement of nodes occurs at the end of each unit time-step, or another mechanism

requiring checkpoints from exiting node(s) is used such as that in [66]. The second, $SMRPO-CP2$, is the case in which nodes are rebooted individually using the *CP2* checkpointing method and each node takes one quarter of a unit time-step to reboot and construct its state. We note that these two figures give lower and upper bounds on the expected lifetime in the case when the *CP1* checkpointing mechanism is used, or the *CP2* checkpointing mechanism is used with a shorter reboot time. A full discussion of the relationship between these expected lifetimes can be found in Appendix B.

The cases with proactive obfuscation use the following amounts of diversity; 2^3, 2^4, 2^8, 2^{16}, 2^{32}, 2^{40} and infinite diversity (as defined in Section 4.4). The infinite diversity case assumes that the adversary has no way of knowing that a diverse executable it has encountered is the same diverse executable that it previously compromised. Hence, in this case, as far as the adversary is concerned, it only encounters executables that it has not seen before in previous unit time-steps.

The other cases fall into two categories. The first of these is motivated by the amounts of diversity commonly available through ASLR (one of the randomisation techniques that can be used as part of proactive obfuscation); 2^{16} in 32 bit systems and 2^{32} or 2^{40} in 64 bit systems, as detailed in [51]. The second is motivated by considering the situation when the reboot and refresh process does not manage to successfully remove all of the malicious code from a previously compromised node. Then, we have the situation where previously compromised nodes stay compromised, and may be used in future time-steps. However, variability in reboot and message delivery times results both in randomness of the order in which nodes are chosen to become active and the need for a spare pool containing more machines than are needed for one time-step. This leads us to consider systems with diversities of 2^3,2^4 and 2^8, literally systems with 2^3,2^4 or 2^8 machines in the server farm, all of which stay compromised after reboot and refresh.

5.2.1 Systems with SO or PR Obfuscation Schemes

A surprising result here is that, when the SO or PR obfuscation schemes are used, the PB system outperforms the SMR system, except when checkpointing method CP2 is used with a relatively large reboot time. This is surprising as an active replication system with proactive recovery was, up until [53], considered a fairly satisfactory method for providing intrusion tolerance, and a PB system aims only to provide crash tolerance.

We also observe that the FORTRESS system significantly outperforms the SMR system regardless of checkpointing method used.

These results suggest that, when faced with the possibility of attacks conforming to the attack models presented in Sections 4.6.1,4.7,4.8, implementing an SMR system

without proactive obfuscation is not likely to provide a significant improvement in intrusion resilience, and may in some cases cause a reduction. The reasons behind these results are discussed in more detail in Section 5.5.

The expected lifetimes for the SMR, PB and FORTRESS systems using the PR or SO obfuscation schemes are shown in Table 5.1, as α_0, the probability of node compromise in the first unit time-step varies (and hence the probability of node compromise in each unit time-step varies). The table shows that, for each value of α_0, the system providing the highest expected lifetime is the FORTRESS system, followed by the SMR system using CP2, the PB system and finally the SMR system using en-masse replacement or CP1.

The abbreviation $SMR - CP2$ is used to denote the SMR system using the PR obfuscation scheme and checkpointing method *CP2* with $\triangle = 0.25$, which corresponds to each node taking $1/4$ of the unit time-step to reboot . We note that, for $\triangle \leq 0.2$ the PB system outperforms the SMR system using checkpointing method *CP2*.

We also note that, during the time when a node is rebooting for the SMR system using checkpointing method CP2, an attacker who has intruded one node other than the one that is rebooting can prevent the ordering, and hence processing of client requests. This suggests that the expected lifetime of the SMR system using the PR obfuscation scheme and checkpointing method CP2 is not directly comparable to that of the other systems here, as in all other cases we are assuming that the attacker can only adversely affect the system by achieving the compromise conditions, and here it is possible for an attacker to prevent processing during a proportion of the unit time-step equal to $3\triangle$ simply by compromising one node.

We also note that the compromise conditions for the FORTRESS system explicitly include the situation where all three proxy nodes have been compromised, and hence the availability of the system has been completely blocked, although the integrity may not yet have been compromised.

5.2.2 Systems using the PO Obfuscation Scheme

When the PO obfuscation scheme is used, the relationship between the intrusion tolerance of the SMR, PB and FORTRESS systems depends on the amount of diversity available (see Section for the definition of diversity). When 2^4 or less diversity is available, the expected lifetimes for the PB and FORTRESS system are very close, with the PB system outperforming the FORTRESS system for most values of α_0 (the probability of an attacker successfully compromising a node in a unit time-step). However, when 2^8 or more diversity is available, the FORTRESS system significantly outperforms the PB system.

When 2^8 or less diversity is available the PB system significantly outperforms the

Table 5.1: Expected Lifetimes of Systems with SO and PR Obfuscation Schemes

α_0	SMR	SMR-CP2	PB	FORTRESS
0.00001	39997.2	53330.5	50000.5	65087
0.00002	19997.2	26663.8	25000.5	32542.8
0.00003	13330.5	17774.9	16667.2	21694.8
0.00004	9997.2	13330.5	12500.5	16270.7
0.00005	7997.2	10663.8	10000.5	13016.3
0.00006	6663.8	8886.1	8333.83	10846.7
0.00007	5711.45	7616.2	7143.36	9297
0.00008	4997.18	6663.8	6250.5	8134.6
0.00009	4441.61	5923.1	5556.06	7230.7
0.0001	3997.17	5330.5	5000.5	6507.5
0.0002	1997.17	2663.8	2500.5	3253
0.0003	1330.50	1774.9	1667.17	2168
0.0004	997.17	1330.5	1250.5	1626
0.0005	797.17	1063.83	1000.5	1300.3
0.0006	663.84	886.1	833.84	1083.4
0.0007	568.61	759.1	714.79	928.5
0.0008	497.18	663.8	625.5	812.23
0.0009	441.62	589.8	556.06	721.83
0.001	397.18	530.5	500.5	649.5093
0.002	197.20	263.9	250.5	324.07
0.003	130.54	175	167.17	215.59
0.004	97.224	130.54	125.5	161.36
0.005	77.238	103.9	100.5	128.82
0.006	63.919	86.1	83.834	107.13
0.007	54.409	73.4	71.929	91.64
0.008	47.280	63.9	63	80.016
0.009	41.74	56.5	56.056	70.982
0.01	37.31	50.61	49.995	63.754

SMR system. However, when 2^{16} or more diversity is available, the SMR system significantly outperforms the PB system.

The FORTRESS system outperforms the SMR system for all amounts of diversity.

This leads us to conclude that, when the PO obfuscation system is used, the PB system is the best choice if 2^4 or less diversity is available, and the FORTRESS system is the best choice if 2^8 or more diversity is available. We note that this conclusion assumes that indirect attacks are not possible against the FORTRESS system. We will discuss where this assumption is likely to hold in Section 5.3, and present results for when it does not in Section 5.4.

We illustrate the results for the cases with $2^3, 2^4$ and 2^8 diversity by presenting the expected lifetimes for the SMR, PB and FORTRESS systems for $\alpha_0 = 0.00001$ to $\alpha_0 = 0.001$ in Table 5.2.

The results for the cases with $2^{16}, 2^{32}, 2^{40}$ and infinite diversity are shown in Table 5.3. Both of these tables show the relationships between expected lifetimes for the three systems that we have discussed.

We note that the values given here for the SMR system assume that en-masse replacement is being used rather than a checkpointing scheme. A comparison between values for the SMR system with en-masse replacement and the SMR system using checkpointing method $CP2$ is presented in Appendix C. In general however we note that the values for SMR using checkpointing method CP2 are larger than those for SMR with en-masse replacement while still maintaining the same relationship with those for the PB and SMR systems.

Table 5.2: Expected Lifetimes of Systems with Proactive Obfuscation

α_i	System	$EL(2^3)$	$EL(2^4)$	$EL(2^8)$
0.00001	SMR	50832	52423	54438
	PB	100000	100000	100000
	FORTRESS	98667	99466	242829
0.00002	SMR	25851	25970	29289
	PB	50000	50000	50000
	FORTRESS	49421	49980	139395
0.00003	SMR	17179	17205	20465
	PB	33333	33333	33333
	FORTRESS	32195	33229	99595
0.00004	SMR	12802	12991	16235
	PB	25000	25000	25000
	FORTRESS	24286	25315	78162

α_i	System	$EL(2^3)$	$EL(2^4)$	$EL(2^8)$
0.00005	SMR	10365	10574	13436
	PB	20000	20000	20000
	FORTRESS	19817	20436	64965
0.00006	SMR	8547	8734	11612
	PB	16667	16667	16667
	FORTRESS	16317	17018	57020
0.00007	SMR	7371	7423	10099
	PB	14286	14286	14286
	FORTRESS	14048	14715	49894
0.00008	SMR	6438	6536	9258
	PB	12500	12500	12500
	FORTRESS	12236	12938	44633
0.00009	SMR	5695	5725	8399
	PB	11111	11111	11111
	FORTRESS	11025	11599	40530
0.0001	SMR	5146	5244	7726
	PB	10000	10000	10000
	FORTRESS	9912	10535	37193
0.0002	SMR	2578	2622	4464
	PB	5000	5000	5000
	FORTRESS	4972	5494	20989
0.0003	SMR	1714	1746	3270
	PB	3333	3333	3333
	FORTRESS	3355	3816	14949
0.0004	SMR	1315	1338	2633
	PB	2500	2500	2500
	FORTRESS	2562	2929	11709
0.0005	SMR	1041	1058	2190
	PB	2000	2000	2000
	FORTRESS	2055	2368	9741
0.0006	SMR	878	879	1935
	PB	1667	1667	1667
	FORTRESS	1717	2010	8349
0.0007	SMR	754	757	1716
	PB	1429	1429	1429
	FORTRESS	1497	1762	7223
0.0008	SMR	663	664	1555
	PB	1250	1250	1250
	FORTRESS	1310	1565	6472

α_i	System	$EL(2^3)$	$EL(2^4)$	$EL(2^8)$
0.0009	SMR	587	595	1429
	PB	1111	1111	1111
	FORTRESS	1179	1410	5805
0.001	SMR	534	535	1310
	PB	1000	1000	1000
	FORTRESS	1073	1287	5342

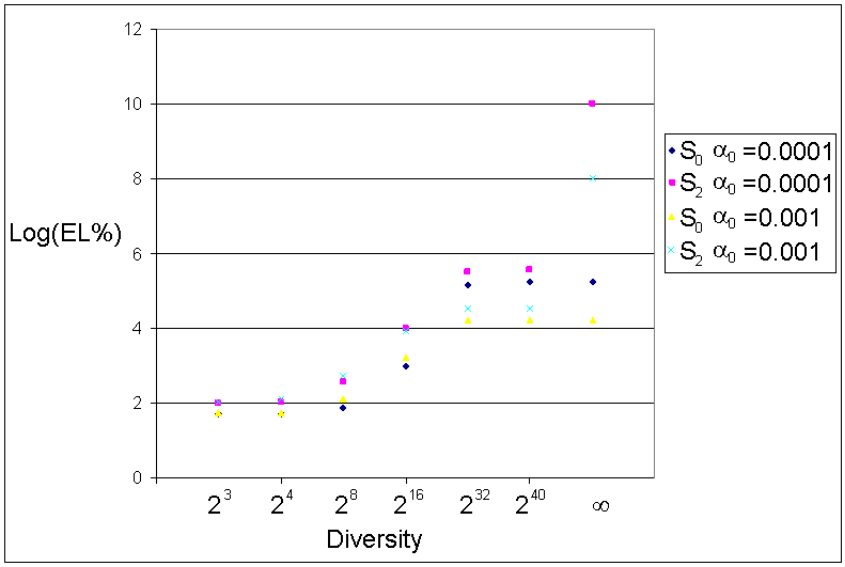
Table 5.3: Expected Lifetimes of Systems with Proactive Obfuscation

α_i	System	$EL(2^{16})$	$EL(2^{32})$	$EL(2^{40})$	$EL(\infty)$
0.00001	SMR	493946	3.0×10^8	1.6×10^9	1.7×10^9
	PB	100000	100000	100000	100000
	FORTRESS	9.0×10^6	1.4×10^9	3.4×10^9	1.0×10^{15}
0.00002	SMR	306317	1.4×10^8	4.3×10^8	4.2×10^8
	PB	50000	50000	50000	50000
	FORTRESS	4.8×10^6	5.2×10^8	8.5×10^8	1.3×10^{14}
0.00003	SMR	233115	8.8×10^7	1.9×10^8	1.9×10^8
	PB	33333	33333	33333	33333
	FORTRESS	3.2×10^6	2.8×10^8	3.7×10^8	3.7×10^{13}
0.00004	SMR	190370	5.7×10^7	1.1×10^8	1.0×10^8
	PB	25000	25000	25000	25000
	FORTRESS	2.5×10^6	1.7×10^8	2.2×10^8	1.6×10^{13}
0.00005	SMR	162610	4.2×10^7	6.7×10^7	6.7×10^7
	PB	20000	20000	20000	20000
	FORTRESS	2.0×10^6	1.1×10^8	1.3×10^8	8×10^{12}
0.00006	SMR	142772	3.2×10^7	4.7×10^7	4.6×10^7
	PB	16667	16667	16667	16667
	FORTRESS	1.6×10^6	8.5×10^7	9.4×10^7	4.6×10^{12}
0.00007	SMR	128109	2.5×10^7	3.4×10^7	3.4×10^7
	PB	14286	14286	14286	14286
	FORTRESS	1.4×10^6	6.3×10^7	6.8×10^7	2.9×10^{12}
0.00008	SMR	116551	2.0×10^7	2.6×10^7	2.6×10^7
	PB	12500	12500	12500	12500
	FORTRESS	1.3×10^6	4.9×10^7	5.3×10^7	2.0×10^{12}
0.00009	SMR	105816	1.7×10^7	2.1×10^7	2.1×10^7
	PB	11111	11111	11111	11111
	FORTRESS	1.1×10^6	4.0×10^7	4.1×10^7	1.4×10^{12}

α_i	System	$EL(2^{16})$	$EL(2^{32})$	$EL(2^{40})$	$EL(\infty)$
0.0001	SMR	99073	1.4×10^7	1.7×10^7	1.7×10^7
	PB	10000	10000	10000	10000
	FORTRESS	1.0×10^6	3.2×10^7	3.5×10^7	1.0×10^{12}
0.0002	SMR	59599	4.0×10^6	4.1×10^6	4.2×10^6
	PB	5000	5000	5000	5000
	FORTRESS	509185	8.1×10^6	8.3×10^6	1.3×10^{11}
0.0003	SMR	43864	1.8×10^6	1.8×10^6	1.9×10^6
	PB	3333	3333	3333	3333
	FORTRESS	334515	3.7×10^6	3.8×10^6	3.7×10^{10}
0.0004	SMR	35163	1.0×10^6	1.0×10^6	1.0×10^6
	PB	2500	2500	2500	2500
	FORTRESS	244124	2.1×10^6	2.1×10^6	1.6×10^{10}
0.0005	SMR	29529	652002	666486	667111
	PB	2000	2000	2000	2000
	FORTRESS	192574	1.4×10^6	1.4×10^6	8×10^9
0.0006	SMR	25235	453463	456939	463333
	PB	1667	1667	1667	1667
	FORTRESS	158368	922068	922957	4.6×10^9
0.0007	SMR	22339	339199	340000	340454
	PB	1429	1429	1429	1429
	FORTRESS	133905	679182	689444	2.9×10^9
0.0008	SMR	19831	264487	263622	260695
	PB	1250	1250	1250	1250
	FORTRESS	114270	529604	529393	2.0×10^9
0.0009	SMR	17949	207162	206891	206008
	PB	1111	1111	1111	1111
	FORTRESS	99213	415022	417156	1.4×10^9
0.001	SMR	16165	166331	164607	166889
	PB	1000	1000	1000	1000
	FORTRESS	87731	339522	340623	1.0×10^9

The effect of diversity on the systems is illustrated in Figure 5.2 where the expected lifetimes of the FORTRESS and SMR systems are presented as percentages of the expected lifetimes of the PB system as diversity varies. The percentages are presented on a logarithmic scale to improve readability. We note that, as base 10 logarithms are used, any value less than 2 indicates a smaller expected lifetime than that of the S_1 system and any value greater than 2 indicates a larger expected lifetime than that of the S_1 system.

Figure 5.2: Relative Expected Lifetimes of S_0, S_2 as Diversity Varies



5.3 On the Feasibility of Successful Indirect Attacks

Here we consider three groups of real-world situations in which indirect attacks may not be possible or successful, and then comment on the conditions under which these situations are likely to occur.

5.3.1 Systems where Proxies and Servers have the Same Vulnerabilities

When the proxies and servers have the same vulnerabilities an attacker will be unable to target the servers with indirect attacks, as any malicious request intended as an indirect attack on the servers, will instead function as a direct attack on one or more proxies. This will mean that either the attack will, with some small positive probability, compromise the proxy server, or, with some large probability will cause the process handling it to crash and be replaced by a new process without passing it to the server tier.

An example of this would be when both the servers and proxies make use of the Apache Tomcat application server with the mod_jk connector installed. A previous version of mod_jk was found to have a vulnerability where an HTTP request with an overly long URL would result in a buffer overflow. Hence an attacker would be able to craft HTTP requests that caused a stack overflow and contained malicious code such as detailed in Section 2.8.1.1-2.8.6.

However, if these malicious requests were sent from a client, then when a proxy received a request it would cause a buffer overflow, preventing the proxy from passing

the malicious request to the server tier. This would make indirect attacks impossible, and instead require the attacker to first compromise a proxy, from which they could then directly attack the server tier

5.3.2 Filtering Out Malicious Requests

Here an attacker will be unable to target the servers with indirect attacks as any malicious request intended as an indirect attack on the servers will either be rejected by the proxies or modified in such a way that it fails to work. This will result in either the attack failing to reach the servers, or reaching the servers as an invalid request incapable of compromising a server.

An example of this would be when the servers make use of a version of the Apache Tomcat application server with the mod_jk connector installed, with the URL length vulnerability discussed in Section 5.3.1, and the proxies make use of a later version of the Apache Tomcat application server that no longer has this vulnerability. In this case an attacker could produce malicious HTTP requests with an overly long URL to attempt to attack the server tier.

If these requests were sent directly to the server tier from a compromised proxy then they would cause a buffer overflow. However, if they were sent from a malicious client, this would result in the proxies that receive them rejecting them as having overly long URLs, rather than passing them to the server tier.

We note that this example may seem contrived, as it involves the servers running a version of Apache Tomcat with a known vulnerability and the proxies running a later version with that vulnerability fixed. However, the general principle illustrated is that a correctly written piece of defensive programming in software running on the proxy layer, designed to prevent vulnerabilities in the proxy, may disrupt indirect attacks targeted at the server tier. We also note that some legacy software may only be able to be run using particular versions of software such as web servers, potentially resulting in a situation where the server tier does run an older version of common software than the proxy tier.

5.3.3 Filtering out Attack Feedback

We have also noted in Section 2.11 that proxies may make it harder for an attacker to monitor the effect of a malicious request, in effect filtering out the necessary feedback from a malicious request that makes it possible for an attacker to use it. While this does not stop the malicious request from reaching the server, it does prevent indirect attacks being launched at the same speed as direct attacks, and may, in some cases, slow them down so much that they effectively become impossible to use.

5.3.4 Likelihood of Indirect Attack Impossibility

Producing software free of vulnerabilities is difficult and costly, and in practice vulnerabilities are regularly found in commercial software. This has a very negative effect on security as attackers only need one vulnerability to compromise a piece of software, whereas developers need to remove every possible vulnerability to make the software completely attack-proof. However, the prevention of indirect attacks in the ways highlighted here is a much more likely proposition. Indirect attacks are prevented either when the proxy has the same vulnerability as the server, or when the proxy handles a potential vulnerability to the point of not allowing it to propagate to the server.

The case in Section 5.3.1 is likely to happen when the proxy uses one or more pieces of software, such as web servers, that are also used on the server. The case in Section 5.3.2, assuming that the software on the proxy is written with defensive programming in mind, does not require either the proxy or the server to be free of vulnerabilities, instead it just requires the proxy and server to have different vulnerabilities. This means that being able to launch indirect attacks is a considerably different proposition from being able to launch direct attacks; while every large piece of software is likely to have some vulnerability, and hence be open to some sort of direct attack, the chance of two separate pieces of software having the same or sufficiently similar vulnerabilities is likely to be a lot smaller.

Finally the case in Section 5.3.3 does not require any of the conditions for the other two cases. Instead, it is likely to occur when an attacker uses a carefully crafted attack such as those in [51, 57].

5.4 Intrusion Resilience when Indirect Attacks are Possible

We now look at expected lifetimes for the FORTRESS system using the PO obfuscation scheme when indirect attacks are possible. We note that indirect attacks do not occur in the SMR and PB systems, as all nodes are accessible by direct attacks, making the concept of an indirect attack meaningless for these systems.

We compare these expected lifetimes to those of the SMR and PB systems using the PO obfuscation scheme that were presented in Section 4.3 to determine how the possibility of indirect attacks affects the choice of intrusion resilience method.

5.4.1 Expected Lifetimes of the FORTRESS System with PO and Indirect Attacks

The relationship between the intrusion tolerance of the SMR, PB and FORTRESS systems depends on the amount of diversity available, as was the case when indirect attacks were not possible.

When 2^4 or less diversity is available, the FORTRESS system outperforms the SMR system, but is outperformed by the PB system for all values of κ. Hence our recommendation of the PB system when 2^4 or less diversity is available is unchanged by the possibility of indirect attacks.

When 2^8 diversity is available, the FORTRESS system outperforms the SMR system, but is outperformed by the PB system for high values of κ. The κ values for which the PB system outperforms the FORTRESS system vary with the intrusion probability, in the region $0.8 < \kappa < 1$.

This leads us to modify our recommendations when 2^8 diversity is available as follows: When $\kappa \leq 0.8$ the FORTRESS system is the best choice and when $\kappa > 0.8$ the PB system may be a better choice.

When 2^{16} or greater diversity is available, the FORTRESS system outperforms the PB system except when κ is close to 1, and the SMR system outperforms the PB system except when κ is close to 0.

Hence, when 2^{16} or greater diversity is available we recommend that the SMR system is the best choice, unless it can be determined that indirect attacks are impossible, as discussed in Section 5.5.

We illustrate the difference between the FORTRESS, SMR and PB systems when 2^4 or less diversity is available by comparing the intrusion resilience of the three systems with 2^3 diversity in Figure 5.3 and 2^4 diversity in Figure 5.4. In both cases, we show the FORTRESS system with indirect attack coefficient values $\kappa = 0.1$ and $\kappa = 0.9$. These figures demonstrate that the PB system outperforms both the SMR system and the FORTRESS system for these amounts of diversity.

The FORTRESS system with indirect attack coefficients $\kappa = 0.7$ and $\kappa = 1$ are compared with the PB system when 2^8 diversity is available in Figure 5.5, illustrating that the PB system generally outperforms the FORTRESS system when $\kappa > 0.8$ while the FORTRESS system outperforms the PB system when $\kappa < 0.8$.

The FORTRESS system with indirect attack coefficients $\kappa = 0$ and $k = 0.3$ are compared with the SMR system when 2^{16} diversity is available in Figure 5.6, illustrating that the FORTRESS system outperforms the SMR system when $\kappa = 0$ whereas the SMR system outperforms the FORTRESS system for higher κ values. A similar comparison is provided when 2^{32} diversity is present in Figure 5.7 and when 2^{40} diversity is present in Figure 5.8.

Figure 5.3: EL with 2^3 Diversity

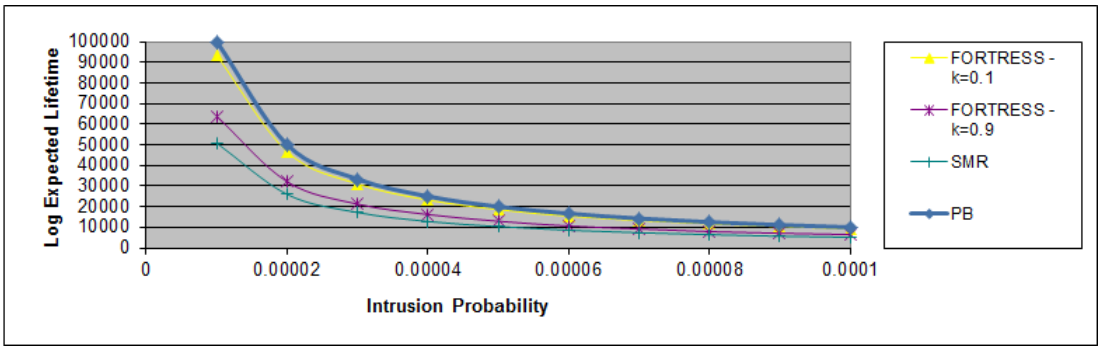


Figure 5.4: EL with 2^4 Diversity

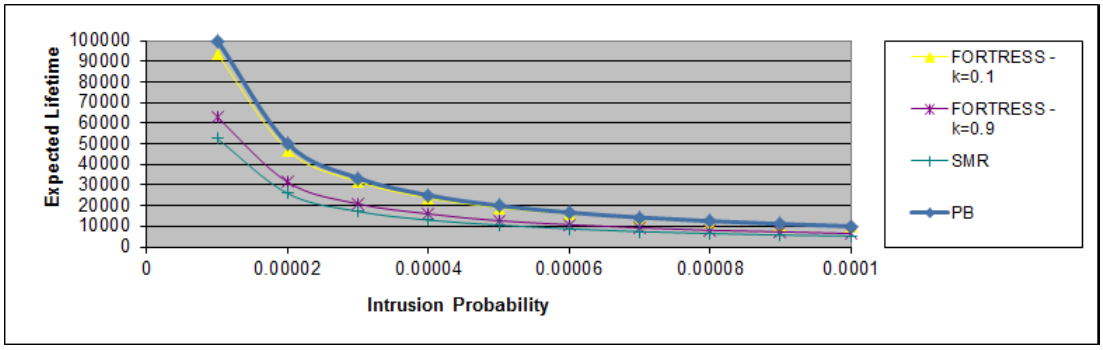


Figure 5.5: EL with 2^8 Diversity

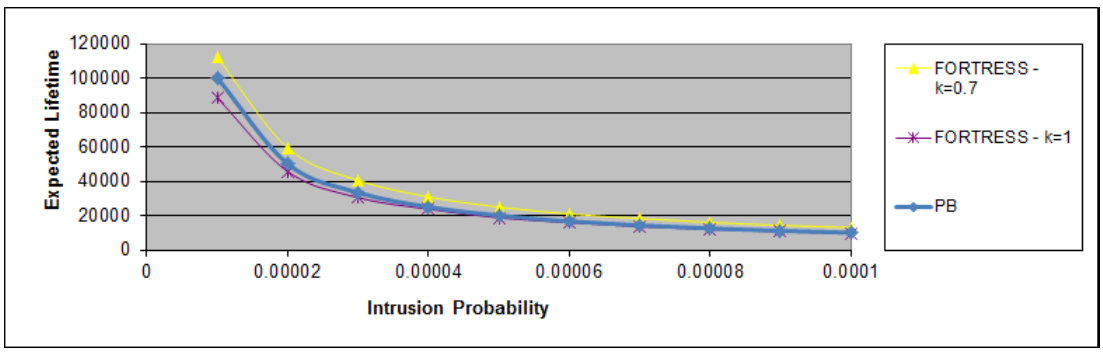


Figure 5.6: EL with 2^{16} Diversity

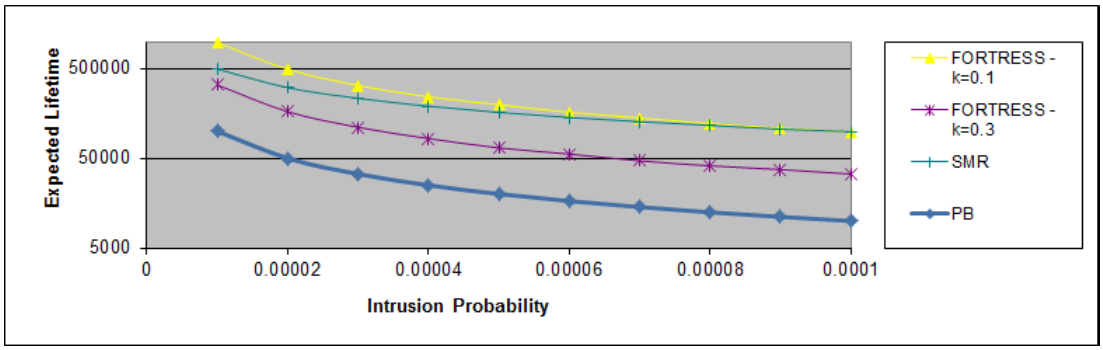


Figure 5.7: EL with 2^{32} Diversity

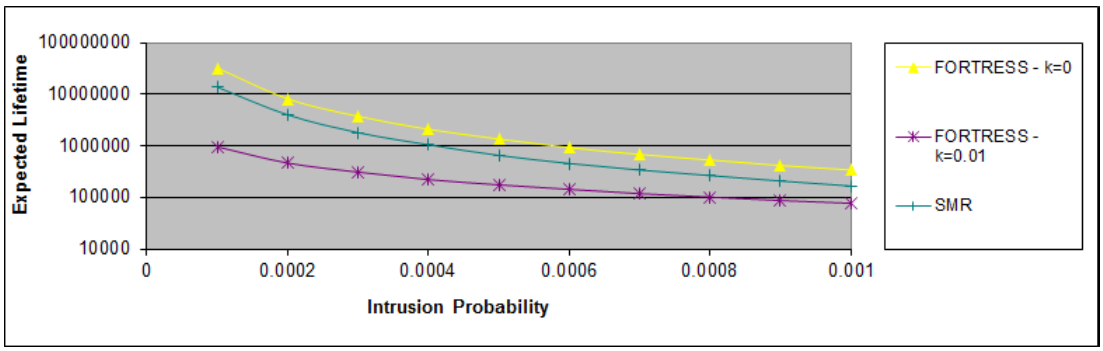


Figure 5.8: EL with 2^{40} Diversity

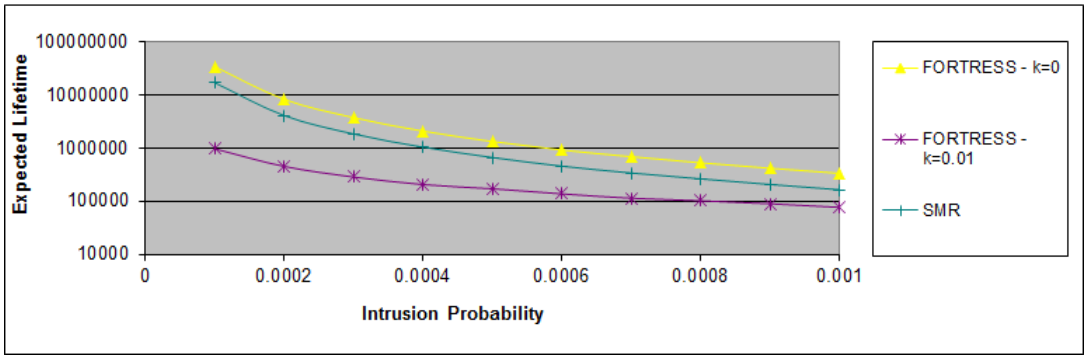


Figure 5.9: EL with Infinite Diversity

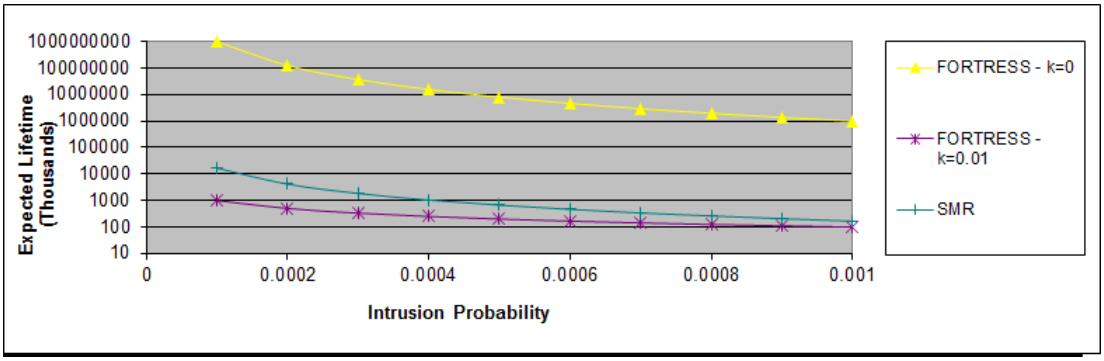


Table 5.4: EL of FORTRESS System with 2^3 Diversity and $\alpha = 0.00001$ as κ Varies

κ	Expected Lifetime	EL as % EL of SMR
0	98667.175	194.1061497%
0.01	96385.452	189.6173573%
0.1	93304.3701	183.5560005%
0.5	75655.1112	148.8349326%
0.9	63796.945	125.5065766%
1	62022.218	122.0151882%

Table 5.5: EL of FORTRESS System with 2^4 Diversity and $\alpha = 0.00001$ as κ Varies

κ	Expected Lifetime	EL as % EL of SMR
0	99465.5084	189.735746%
0.01	98454.6064	187.8073966 %
0.1	93504.1717	178.3641792 %
0.5	76072.3366	145.1120269 %
0.9	62730.1518	119.6611 %
1	59275.4777	113.0711254 %

We also present the expected system lifetimes for $\alpha = 0.00001$ to give an illustration of the actual differences between the systems with the seven levels of diversity. The expected lifetimes are presented both as actual values, and as a percentage of the expected lifetime of the SMR system using the PO obfuscation scheme. The comparisons are shown in Table 5.4 for 2^3 diversity, Table 5.5 for 2^4 diversity, Table 5.6 for 2^8 diversity, Table 5.7 for 2^{16} diversity, Table 5.8 for 2^{32} diversity, Table 5.9 for 2^{40} diversity and Table 5.10 for infinite diversity.

Table 5.6: EL of FORTRESS System with 2^8 Diversity and $\alpha = 0.00001$ as κ Varies

κ	Expected Lifetime	EL as % EL of SMR
0	242829.1864	446.064225
0.01	237436.1586	436.1575214
0.1	212953.5655	391.1843076
0.5	134371.8495	246.8338992
0.9	94679.9261	173.9219592
1	88353.3262	162.3003337

Table 5.7: EL of FORTRESS System with 2^{16} Diversity and $\alpha = 0.00001$ as κ Varies

κ	Expected Lifetime	EL as % EL of SMR
0	8977638.682	1817.53%
0.01	5229388.777	1058,696%
0.1	978791.0867	198.1574%
0.5	198847.1175	40.256%
0.9	112898.9021	22.856%
1	101364.491	20.52136

Table 5.8: EL of FORTRESS System with 2^{32} Diversity and $\alpha = 0.00001$ as κ Varies

κ	Expected Lifetime	EL as % EL of SMR
0	1400422229	465.9829282 %
0.01	8299669.933	2.761670315%
0.1	985562.4444	0.327940577%
0.5	198383.2668	0.06601096 %
0.9	111977.8188	0.037260014 %
1	99995.12875	0.033272839%

Table 5.9: EL of FORTRESS System with 2^{40} Diversity and $\alpha = 0.00001$ as κ Varies

κ	Expected Lifetime	EL as % EL of SMR
0	3355837206	211.5782516%
0.01	8192575.169	0.516524081%
0.1	942884.1564	0.059446799%
0.5	199968.3893	0.012607573%
0.9	113760.8021	0.007172372%
1	97772.80357	0.006164363%

Table 5.10: EL of Fortress System with Infinite Diversity and $\alpha = 0.00001$ as κ Varies

κ	Expected Lifetime	EL as % EL of SMR
0	1E+15	59999172.3 %
0.01	9999999.9	0.599991717 %
0.1	999999.999	0.059999172 %
0.5	200000	0.011999834 %
0.9	111111.1111	0.006666575 %
1	99999.99999	0.005999917 %

5.5 Discussion of Results

The key results we have seen are:

- **When the SO or PR obfuscation schemes are used, both the FORTRESS system and the PB system outperform the SMR system, unless the SMR system uses checkpointing method CP2.**

This can be seen to result from a combination of the properties of distributed attacks, the nature of defences against code injection and the nature of active replication schemes.

Distributed attacks allow an attacker to provide as many malicious requests as the publicly accessible servers can handle. Hence an increase in the number of servers does not result in a decrease in the rate at which each server is attacked. So, each new server added is just as vulnerable to attack as the existing servers.

We have seen in Section 2.8.7 that, in general, defences against code injection attacks do not entirely stop these attacks. New vulnerabilities, commonly termed zero day exploits, are constantly being found and used by malicious intruders. Instead, what defences against code injection attacks do achieve is to complicate the methods used to intrude into systems when vulnerabilities have been found, and to introduce uncertainty into the success of each individual malicious request. However, once an attack has succeeded then, unless the machine intruded into is re-obfuscated, most or all of this uncertainty is removed. The attacker is left with knowledge of the system structure and any randomisation keys that have been used to alter the structure or instruction set. This means that in practice, without re-obfuscation, a previously intruded machine is very easily intruded into again.

Active replication schemes require four replicas to tolerate one intruded replica, and an additional three replicas for every additional intrusion that they can tolerate. This means that, as the number of servers available increases by three, the number of extra intrusions that can be tolerated only increases by one. When this is combined with the fact that these three servers can all be attacked at the same rate as the existing servers, and that once intruded into they can be easily re-intruded into, we can explain why the PB system can outperform the SMR system. Adding active replication increases the number of replicas an attacker can attempt to compromise from one to four while only requiring the attacker to compromise one more node than in the primary-backup case.

We note that the idea of a replicated system being out-performed by an unreplicated system has previously been presented in the context of n-modular redundancy in [36]. Here, a system employing n-modular redundancy was shown to be less reliable than a system without redundancy (excluding the possibility of component replacement)

for sufficiently large λT, where T is the period of time considered and λ is the failure rate.

- **When the PR obfuscation scheme is used, the FORTRESS system outperforms the SMR system using checkpointing method CP2 which in turn outperforms the PB system.**

The use of the CP2 checkpointing method allows each server to avoid processing some part of the requests sent to the SMR system. This reduces the probability of each server being compromised in a given unit time-step sufficiently to allow the SMR system to outperform the PB system.

- **Using the PO obfuscation scheme results in a substantial increase in intrusion resilience for any system relative to using the SO or PR obfuscation schemes.**

When faced with the PO obfuscation scheme, an attacker will start each unit time-step knowing that the randomisation key for each node could hold any possible value. Thus the attacker will try as many of these values as possible and hope that one of the values tried will be correct for enough nodes to compromise the system. However, when faced with the SO or PR obfuscation schemes, the attacker will have a more effective strategy available. The randomisation keys are known not to change from unit time-step to unit time-step, so the attacker does not need to consider any randomisation keys that have been tried in a previous unit time-step and found to be incorrect. Hence, the attacker can follow a systematic strategy to eliminate all of the possible randomisation keys until the correct ones are found. This not only increases the probability of success with time, but also results in some finite upper-bound on the amount of time that will be required to compromise the system.

- **When the PO obfuscation scheme is used and diversity of 2^4 or less is available, the PB system outperforms the FORTRESS system and the SMR system.**

The use of the PO obfuscation schemes causes a large increase in the intrusion tolerance of each node due to the nodes being replaced at the end of each unit time-step. However, when the amount of diversity available is relatively small, this is more of an advantage for the PB system than it is for the SMR or FORTRESS systems.

The FORTRESS and SMR systems both have a larger number of nodes that can be attacked at once, and rely on being able to survive the intrusion of some of these nodes. When a relatively small amount of diversity is available, these intrusions will soon result in a large proportion of that diversity having been compromised. This will produce a high probability of the system beginning a unit time-step in the condition

that an attacker can easily compromise enough nodes to compromise the entire system through re-using previous attacks.

When 2^4 or less diversity is available, an attacker is able to exhaust the uncompromised diversity available more rapidly than the PB system can be compromised.

- **When the PO obfuscation scheme is used and 2^8 diversity is available, the FORTRESS system outperforms the SMR system, but is outperformed by the PB system for $\kappa > 0.8$**

When diversity is increased to 2^8, the fact that the attacker can more rapidly exhaust the uncompromised diversity available for the SMR and FORTRESS systems than for the PB system has less of an effect than with smaller amounts of diversity. The exact threshold value of κ at which the PB system outperforms the FORTRESS system is dependent on α_i, but is in the range $0.8 < \kappa < 1$.

- **When the PO obfuscation scheme is used and 2^{16} or greater diversity is available, the FORTRESS system outperforms the PB system except when κ is close to 1, and the SMR system outperforms the FORTRESS system except when κ is close to 0.**

When diversity is increased to 2^{16} or higher, exhaustion of diversity no longer has a significant effect on the relationship between the intrusion resilience of the systems.

The actual threshold value of κ at which the SMR system starts to outperform the FORTRESS system is dependent on both the value of α_i and the amount of diversity, but it is generally in the region of 0.2-0.21 for 2^{16} diversity and 0-0.01 for higher diversity. This suggests that the FORTRESS system is only really superior from an intrusion resilience perspective when indirect attacks are either impossible, or result in the number of malicious requests that can be launched in a unit time-step to be reduced by at least a factor of 100 relative to the number of malicious requests that could be launched in a unit time-step as a direct attack. The exception to this observation is when there is only 2^{16} diversity available and an attacker can easily recognise a previously compromised executable. Then, the number of malicious requests that can be launched in a unit time-step during an indirect attack needs to only be reduced by at least a factor of 5 relative to the number of malicious requests that could be launched in a unit time-step as a direct attack.

- **The possibility of indirect attacks has a large effect on the intrusion resilience of the FORTRESS system.**

The expected lifetime of the FORTRESS system with diversity of 2^{16} or greater when $\kappa = 0$ ranges from 211.58% of that of the SMR system to 59999172.3 % of that of the

SMR system, depending on the amount of diversity available. The expected lifetime of the FORTRESS system with diversity of 2^{16} or greater when $\kappa = 1$ ranges from 0.006% of that of the SMR system to 20.52% of that of the SMR system, depending on the amount of diversity available. This shows both a very large difference between the system lifetimes, and a very large difference between the situations when $\kappa = 0$ and $\kappa = 1$. This suggests that, when purely concerned with expected system lifetime, assessing whether indirect attacks are likely to be possible is very important.

Chapter 6

State Transfer Support System

Previously, in Section 3.5 we have defined the components of a FORTRESS system. Some of these components, such as the server farm and its sub-components, the recovery unit and spare pool, are made up of off-the-shelf hardware. Others, such as the reboot server and name server can be implemented in hardware or software using pre-existing techniques such as the reboot server technique in [45] and standard name server technologies.

This leaves us with three key components that are needed; the controller unit, the software for producing proxy nodes and the additional software that is needed to turn machines running legacy code into proactively fortified systems. We also require state transfer algorithms to allow state to be transferred from the current servers to a new set at the end of each unit time-step.

We begin by defining a set of possible state transfer algorithms in Section 6.1. We then present the conceptual design of a state transfer support system to provide the needed components for a FORTRESS system in Section 6.2. Finally, we present a concrete architecture implemented in Java EE for this state transfer support system in Section 6.3.

6.1 State Transfer Mechanisms

The FORTRESS system requires that system state can be transferred from the servers used in one unit time-step to those in the next, as detailed in Section 3.5.3. This results in a need to be able to produce *checkpoints*, blocks of data that encapsulate the system state at a given instant in time. These checkpoints can then be used to transfer system state from one server to another.

This differs from the state transfer requirements of a primary backup system [37] and from the state transfer requirements of an SMR system [14] in the following ways:

Primary backup systems require that every update is propagated to the backups before any further processing takes place. This is not required between the servers for one unit time-step and the next in the FORTRESS system.

SMR systems transfer state to a set of rebooting servers from a set of non-rebooting servers, some of which may be compromised. Hence there is a requirement for Byzantine agreement between the non-rebooting servers. This requirement is not present in a FORTRESS system.

We begin by considering the checkpointing requirements of a proactively fortified system. We then continue to define four state transfer mechanisms and analyse the advantages and disadvantages of each. These state transfer mechanisms assume that primary-backup replication is used in the server tier. The possibility of extending them for use with state machine replication is considered in appendix D.

Finally, we consider the supporting mechanisms that are required for correctness and liveness to hold for these state transfer mechanism. The correctness and liveness of these state transfer mechanisms are fully analysed in Appendix E.

6.1.1 Checkpointing and Information to be Transferred

There are two possible kinds of system we can encounter; those in which computations are short-lived, and those in which computations are sufficiently long-lived that it would be necessary to allow partially completed computations to be part of the transferred state.

6.1.1.1 Short-Lived Computations

Here it is likely that no computation will be in progress at the time of state transfer, and also that restarting a computation involves very little overhead. This allows us to simply transfer the system state prior to any currently running computations, alongside a list of computations to be performed which includes any that are in progress. In many cases this will allow us to simply serialise the values of all state variables and all computations that are waiting to be performed, then transfer them to a new node with minimal overhead.

Further, if, as in many systems, the results of completed computations are stored in a database, then there may be no need to transfer any information other than the contents of the database and the list of computations that are waiting to be performed. If these values are not entirely stored in a database then it may be possible to extract them using the user interface. Otherwise, there will either be a requirement for the developers providing the proactive fortification to make significant modifications to the legacy code, or to use the methods detailed in Section 6.1.1.2 instead.

6.1.1.2 Long-Lived Computations

Here we are unable to stop and restart computations at state transfer without incurring significant overhead. This makes it beneficial to be able to take a checkpoint of the entire system state, including everything that has been achieved in currently running computations.

There are two ways in which this can be achieved. Either the contents of the stack, heap and registers for the application can be serialised and transferred (as detailed in [46]), or the current state of each computation can be marshalled into a set of variables, serialised, transferred, and then unmarshalled.

1. Stack, Heap and Register Transfer

The stack, heap and registers are serialised and transferred to the replacement node. This has a fairly low overhead, but is not compatible with unique randomisation of the nodes transferred to and from. We note that the stack is highly likely to contain instructions that are yet to be executed.

If address space layout randomisation, described in Section 2.8.3, is used then the stack will contain return addresses and function addresses that would be incorrect for the new node. Similarly, if instruction set randomisation, described in Section 2.8.4, is used then any instructions transferred as part of the stack will be incorrectly randomised for the new node. So, if we wish to use this method then some sort of translation will need to be performed to make the transferred stack contents, instructions, or other randomised values usable on the new node.

2 Marshalling

Here the current state of the computation must be converted into an intermediate form that is independent of the randomisation applied to the system. This intermediate form is transferred to the new node, which then constructs a new state from the intermediate state. This has a considerable overhead at both the marshalling and unmarshalling stages. This method may also require significant modification of the legacy code in some cases, to allow relevant values to be marshalled. This then in turn requires the developers providing the proactive fortification to have an in-depth knowledge of how the legacy code works, something which may be impossible or prohibitively expensive in practice.

6.1.2 State Transfer Mechanisms

The preceding issues lead to us considering four state transfer mechanisms.

Figure 6.1: Single Transfer: Processing Phase

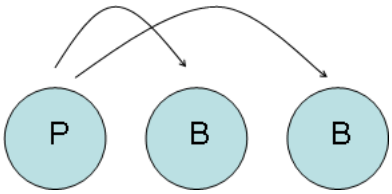
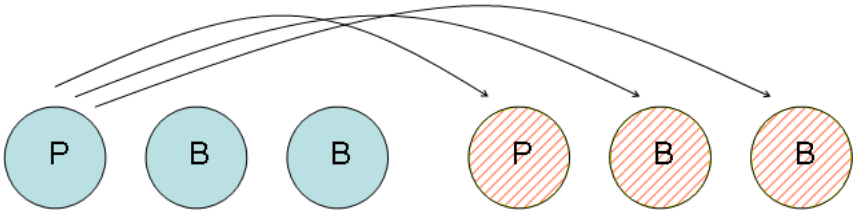


Figure 6.2: Single Transfer: Transfer Phase



6.1.2.1 Single Transfer

Each unit time-step is partitioned into two phases, the processing phase and the transfer phase. A new randomisation key is chosen during the processing phase and a new set of nodes are initialised with that randomisation key. The processing phase is shown in Figure 6.1.

As soon as the processing phase ends, all processing ceases and client requests are queued until the start of the next unit time-step. The primary node generates a checkpoint and sends this to every server node in the new system and all of the backups. If the first backup node does not receive a checkpoint within a pre-set time from the start of the transfer phase then it sends a checkpoint of its own to every server node in the new system, and all lower ordered backups. Each lower ordered backup behaves similarly if it does not receive a checkpoint from the primary or one of the backups that is higher ordered than itself. This transfer phase is illustrated in Figure 6.2.

6.1.2.2 Progressive Transfer

The system starts with two subsets of server nodes. Each subset is identically randomised with the other members of the subset, but differently randomised from the members of the other subset. The primary sends updates to the backups as normal, and also marshals updates to send to the second subset of server nodes. The second subset of server nodes perform exactly like backups with regard to how they handle updates sent to them. The second subset of server nodes do not expect heartbeat messages, and as a result are not able to become the new primary during the unit time-step that they are the second subset of server nodes. The sending of update messages during progressive transfer is illustrated in Figure 6.3.

During the unit time-step a new randomisation key is chosen and a third subset of

Figure 6.3: Progressive Transfer

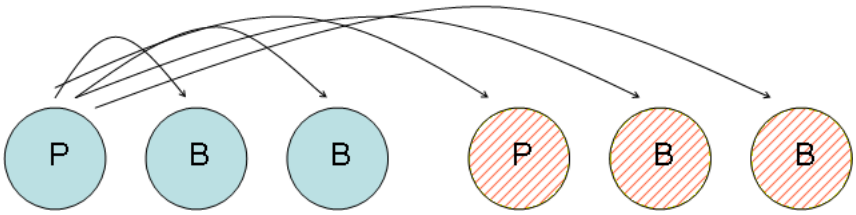
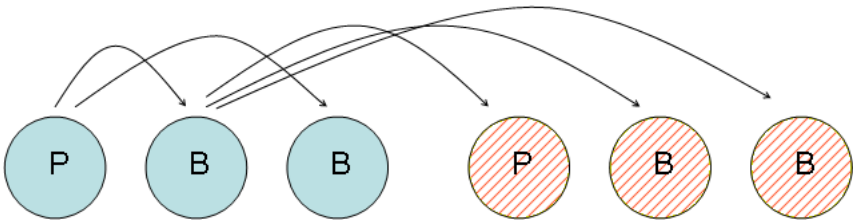


Figure 6.4: Progressive Transfer with Load Balancing



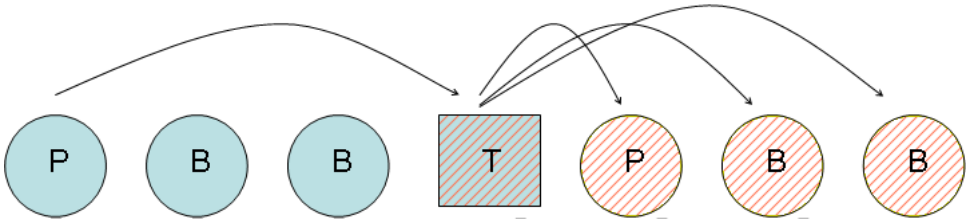
servers is initialised with this key immediately before the end of the unit time-step. When the unit time-step ends, the second subset of servers become the new primary and backups, and the new, incoming subset of servers become the new second subset of servers. The new primary generates a checkpoint, marshals it and sends it to the new second subset of servers. Processing then continues as in the previous unit time-step.

6.1.2.3 Progressive Transfer with Primary Load Reduction

The system starts with two subsets of server nodes. Each subset is identically randomised with the other members of the subset, but differently randomised from the members of the other subset. The primary sends updates to the backups as normal. While there is at least one non-crashed backup, the first backup marshals these updates to send to the second subset of server nodes. When all backups are crashed, the primary performs this marshalling. The second subset of server nodes perform exactly like backups with regard to how they handle updates sent to them, but do not expect heartbeat messages or become the new primary during the unit time-step that they are the second subset of server nodes. The sending of state update messages during progressive transfer with primary load reduction is illustrated in Figure 6.4.

During the unit time-step a new randomisation key is chosen and a third subset of servers is initialised with this key immediately before the end of the unit time-step. When the unit time-step ends, the second subset of servers become the new primary and backups, and the new, incoming subset of servers become the new second subset of servers. The first of the new backups generates a checkpoint, marshals it and sends it to the new second subset of servers. Processing then continues as in the previous unit time-step.

Figure 6.5: Single Transfer with Trusted Components



6.1.2.4 Transfer with Trusted Components

A trusted server is used to convert system state from a format suitable for servers with one set of randomisation keys to a format suitable for servers with a different set of randomisation keys. This server never performs any processing, other than a transformation between system state with an old randomisation key and system state with a new randomisation key. This means that any malicious code from an overflowed buffer will not be executed by the trusted server, making it invulnerable to any of the attacks that may compromise the nodes that perform processing.

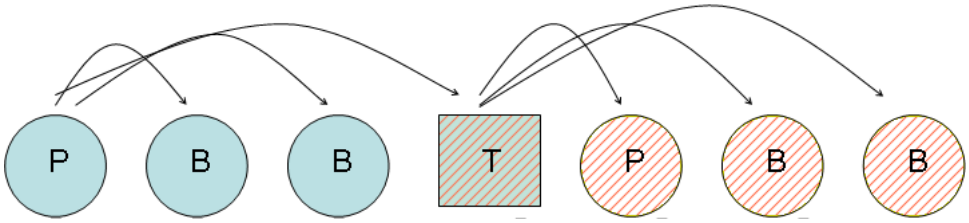
During processing, updates are sent to this trusted server, as though it were another backup. The server uses them to construct the system state, which is stored, and periodically transformed into a system state with the randomisation key for the next time-step. This can be performed as one operation as long as we are considering cases where randomisation involves a linear offset, such as ASLR or ISR. The old key and new key can be used by the trusted server to generate a linear offset that will transform the system state between the two randomisations. If a more complicated randomisation scheme is used then the trusted component will need to perform additional processing, increasing the time to pass on updates. However, if a more complicated randomisation scheme is used then the potential for the transformation of the system state to slow down the primary is greater, making the use of trusted components more beneficial. The sending of state update messages during single transfer with primary load reduction is illustrated in Figure 6.5 and the sending of state update messages during progressive transfer with primary load reduction is illustrated in Figure 6.6.

This trusted component can be used either with single transfer to provide a faster transfer between sets of nodes if randomisation uses a linear offset scheme, as only one transformation will be required rather than at least two if no trusted server was used, or with progressive transfer as an alternative to primary load reduction.

6.1.3 Supporting Mechanism Requirements

The attack model presented in Section 3.2 assumes that an attacker can only compromise the FORTRESS system by sending malicious requests to attempt to compromise proxy or server nodes. Hence, we require that the supporting mechanisms

Figure 6.6: Progressive Transfer with Trusted Components



covered in Section 3.6 are not a legitimate source of attack, and hence will not be compromised during the lifetime of the system. This requirement does not have to mean that we require these mechanisms are completely resilient to attempts at compromising them, just that they are sufficiently more resilient than the proxies and servers to make them uncompromisable in the time frame that an attacker may be able to compromise a server in.

The following six requirements are also needed to fulfil the assumptions of the attack model to ensure the safety and liveness of the state transfer mechanisms (as proven in Appendix E):

1. Approximately Synchronised Clocks

We assume that all server replicas have approximately synchronised clocks. This allows us to set an upper bound on the time at which a correct server replica sends a state message to the server replicas for the next unit time-step.

2 Timely Links

We assume that all messages are either received within a bounded period of time or lost. We also assume that there is a known bound on the number of times a given message can be lost, when transmitted repeatedly.

This, coupled with assumption 1, means that there exists an upper bound on the time taken for a state transfer message that is sent by a correct replica to be received by all new replicas, assuming that messages are re-sent if an acknowledgement is not received.

We will also make the assumption in practice that this bound can be made small enough to make practical systems possible. This is generally a reasonable assumption for the systems that we will analyse, although it could provide practical problems if an attempt was made to implement a proactively fortified system with an incredibly small interval between migrations, or over a very slow network.

3 Authenticated Channels

We assume that all messages received from a given replica have been sent by that replica. This can be achieved in practice by the use of digital signatures or encryption methods with similar properties.

4 Reboot at Fixed Intervals

We assume that nodes are rebooted at fixed times. Each node is rebooted after each time it is used in the system as a server or proxy. No node is re-used as a server or proxy until it has been rebooted.

Whether a node can be a proxy in some unit time-steps and a server in others is left as a design decision for individual implementations of the FORTRESS system. We do however assume that if this is allowed then there will still be a reboot between these two uses of the node.

Reboot at fixed intervals can be achieved in practice by the use of hardware that physically causes reboot, and a mechanism to allow the controller unit responsible for them becoming servers or proxies to become aware of when they have finished rebooting, such as those discussed in Section 3.6.4.

5 Availability of a Name Server

We assume there is a name server available that can receive the addresses of the current proxies from the controller unit and make it available to clients. We do not specify how this name server is implemented in practice, so it may be a hardware switch on a fixed IP address that directs requests to the correct proxies, a software based name server, or even a digitally signed multicast of the addresses to clients in systems that have a small client base that is known to the system administrator.

We also assume that this name server will receive the current public keys for the servers from the controller and make them available to clients, ensuring that authenticated channels exist between servers and clients.

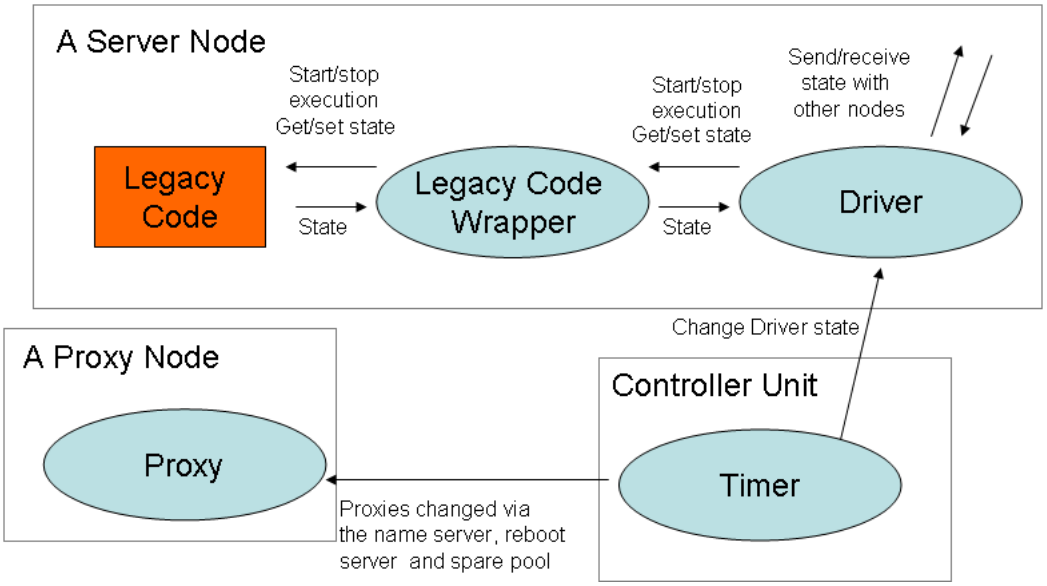
6 Re-Sending of Client Requests

We assume that, if a client does not receive a response to a given request within a given period of time then the client will re-send the request. We also assume that each client will provide an identifier for each request that is unique from the identifier of any other request made by that client. This ensures that, assuming that both the client and the FORTRESS system remain available, every request will eventually be processed exactly once.

6.2 Design

We now present the conceptual design of a state transfer support system for FORTRESS systems.

Figure 6.7: System Components - Conceptual Level



6.2.1 Requirements

We assume that a legacy system is available which may be vulnerable to buffer overflow attacks and attacks of a similar nature. This system is sufficiently complex that an attempt to identify and fix vulnerabilities may not be successful.

We require that this system either runs on hardware capable of also supporting additional software for proactive fortification, or that information flows to and from the legacy system can be channelled through additional hardware running this proactive fortification software. In the second case we assume that it is possible for this additional hardware to copy and replace the memory space in which the legacy code executes and to start and stop its execution. We do not assume that the architects of the proactive fortification system have the necessary understanding of the legacy code to modify it, other than by copying the current contents of the memory space of one execution and writing it to another execution.

The system is built using the progressive transfer method outlined in Figure 6.3. Extensions are possible to allow other transfer methods to be used, and trusted components may be needed in some cases.

6.2.2 Components

The system software is divided into 5 components at the conceptual level as shown in Figure 6.7, we differentiate the Legacy Code from the other components as this is the initial software system to be replicated, rather than being part of the FORTRESS framework. A higher level view of the server node and proxy nodes is shown in Section 3.5 and a system level view of the controller unit is shown in Section 3.6

We note here that the timer process will be situated in the controller unit, while the driver, legacy code wrapper and legacy code processes will all be situated in every server node, and the proxy server will be situated on every proxy node.

6.2.3 Legacy Code

The legacy code is treated as a black box, with four important exceptions; the legacy code wrapper defined in Section 6.2.4 can perform the following actions:

1. Start the legacy code executing.
2. Freeze the legacy code at the current state of execution.
3. While code is frozen: Copy the contents of the memory space in which the legacy code runs.
4. While code is frozen: Replace the contents of the memory space in which the legacy code runs.

Usually, the legacy code will make use of some form of permanent storage, such as a a database. The updates sent to this permanent storage are assumed to be readable by an authorised process.

6.2.4 Legacy Code Wrapper

The legacy code wrapper is a process authorised to perform all of the actions listed in 6.2.3. It contains additional functionality which allows it to perform the following actions when instructed to by an authorised process:

1. Start the execution of the legacy code.
2. Freeze the execution of the legacy code.
3. While code is frozen: Copy the contents of the memory space of the legacy code and forward it to one or more other replicas.
4. While code is frozen: Replace the contents of the memory space in which the legacy code runs with a given copy of the memory space from another replica.
5. Forward all database updates and transactions made by the legacy code in a given time period to one or more other replicas.

As the legacy code wrapper is responsible for forwarding and receiving the contents of the memory space and the database updates and transactions, it is also responsible for any operations such as encryption or producing digital signatures that are necessary as part of ensuring the assumption of authenticated channels that is made in Section 6.1.3.

The last action requires that the legacy code wrapper can intercept all database updates and transactional information generated by the legacy. This may at first appear to violate our assumption that the architects of the proactive fortification system do not need detailed knowledge of the workings of the legacy code. However, this assumption can be seen to hold when we consider the following two solutions.

6.2.4.1 Intercepting Database Updates

Database updates can be intercepted using proxy database drivers as in [32]. These drivers will forward the update information to the other replicas, as well as sending it to the standard database drivers. The only knowledge required by administrators is the public API of the database drivers, code which is not specific to the legacy system, and will, for any major database, be both simple and well documented.

Alternatively, if the legacy system is implemented in J2EE, annotations can be added to entity beans to capture database updates. This does require some modification of the legacy code, however this is limited to adding an annotation to each entity bean class which points to a standardised method that takes the entity bean, wraps it appropriately and passes it to the method for forwarding. The classes that are entity beans can be recognised by their annotations without any need to understand the structure of the legacy code. Similarly, the wrapping and forwarding code requires only knowledge of the name of the entity bean.

6.2.4.2 Intercepting Transactions

The methods available for intercepting transactions depend on whether a common middleware transaction manager is used, or whether the legacy code handles its own transactions.

If a common middleware transaction manager is used then we can simply use a proxy or a modified version of this transaction manager (possibly using a technique like interceptors in J2EE) to intercept transactional information and propagate it before continuing. This technique will be compatible with both systems where every replica has its own database(s) (such as that in [41]) and systems where replicas access an independently replicated database layer (such as that in [32]).

Alternatively, if the system handles its own transactions then transactional information will be propagated along with other system state information when the legacy

code memory space is updated.

6.2.5 Driver

A driver is associated with each replica of the legacy code. The latter can be in one of three modes depending on the role of the host node: Primary, Backup or Standby. A driver takes the status of the node hosting the legacy code it is associated with.

Each driver process is initialised by a message from the timer process into one of three states Primary, Backup or Standby. A driver process in the Primary or Backup state can be stopped by a message from the timer process. A driver process in the Standby state can be changed to the Primary or Backup state by a message from the timer process. With this set-up, drivers, prompted by the timer process, will perform the basic functionalities of the primary-backup replication. Details are given next.

1. Primary

When the driver process transitions into the Primary state it will instruct the legacy code wrapper to start the legacy code executing. It will then start periodically sending heartbeat messages to the backup nodes. After each client request is processed it will request the contents of the legacy code's memory space and any database or transaction updates generated to be forwarded to the backup nodes. Periodically, it will request the contents of the legacy code's memory space and all database updates since the last send to be forwarded to the replicas that will be used in the next unit time-step. This will continue until the timer process declares that the unit time-step is over. At this point any remaining transfer actions will take place, and the driver process will close down.

2 Backup

When the process is in the Backup state it will leave the legacy code in the frozen state. When an update is received from the primary it will instruct the legacy code wrapper to replace the contents of the legacy code's memory space with this update, and apply any database or transaction updates. If an update is received from a higher ordered backup then the process will take this backup to have become the new primary.

If the process does not receive a heartbeat message from the primary within a pre-set period of time then it will wait another pre-set period of time based on the number of higher ordered backups between itself and the primary. If no response is received then it will change state to become the primary. Similarly, if the node receives a heartbeat message from a higher ordered backup then it will recognise this node as having become the primary.

If the backup process does not become the primary then it will continue until the timer process declares that the unit time-step is over. At this point any remaining transfer actions will take place, and the driver process will close down.

3 Standby

When the process is in Standby state it will leave the legacy code in the frozen state and receive updates from the primary like a backup. When a request is received from the timer process, the process will change to the backup or primary state. The standby will not expect heartbeat messages, and will be prepared to receive updates from any of the nodes that are currently designated primary or backups.

6.2.6 Timer

The timer process fulfils the role of the controller unit as detailed in Section 3.6. The timer process waits until the end of each unit time step, and sends a transition message to the primary and backups. It then waits a sufficient length of time to ensure that transition will have occurred and then sends messages to the standby nodes to inform them to become the new primary and backups. The timer also sends messages to initialise new proxies and update the name server to ensure clients are accessing these new proxies.

6.2.7 Proxy

The proxy can be initialised into an active state by a message from the timer process. This message will inform it of the identities of the servers it will forward requests to.

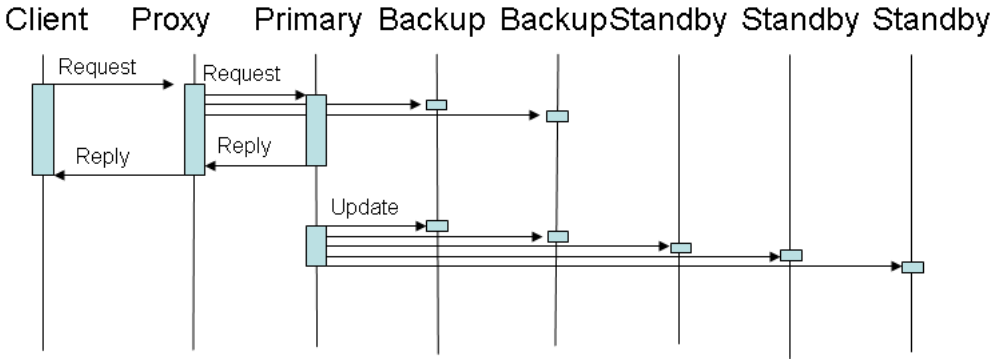
When the proxy server is in an active state it will accept client requests and forward them to the primary and backup nodes. When it receives a response from any of these server nodes it will forward it to the client that it originated from.

6.2.8 Execution

We present an example of the normal execution of a proactively fortified system assuming progressive transfer.

On system start-up one replica is initialised as the primary, a set of $n-1$ replicas are initialised as backups and a set of n replicas are initialised as standbys. Clients send requests via the proxies to the primary and backups. The primary processes these requests and responds to the client via the proxies. The backups ignore the client requests they receive. The primary sends an update to every backup after each request is processed. Periodically, the primary sends updates to the standbys. The

Figure 6.8: Sequence Diagram Showing Normal Execution of Requests and Updates



backups and standbys will replace the memory space of their legacy code with each update they receive.

The normal sequence of execution is shown in Figure 6.8. Only one proxy server is shown in this diagram for clarity. Normally the system would use three proxy servers, all forwarding requests to the primary and backups. Unique request identification numbers allow the primary to ignore multiple copies of the same request and provide exactly-once execution.

If a client sends a request and does not receive a response within a time-out period it will re-send the request. If the primary has crashed then the delay before re-send will enable a backup to have become the new primary.

At the end of the unit time-step the timer will send a message to the primary and all backups to start transition. The primary will then send a final update to all backups and standbys. If a backup does not receive an update from the primary within a time-out period then it will wait for a further time-out period for each higher order backup. If none of the higher ordered backups sends an update then it will become the primary for the transition and send the update message.

After a sufficient time period to allow this transition to occur, the timer sends a message to the standbys to become the new primary and backups. The controller unit also sends a message to the name server with the identities of the new proxies, and a message to the new proxies to become active and direct client requests to the replicas that have just become the primary and backups (see Section 3.6).

6.3 Architecture

Now that the state transfer support system has been defined on a conceptual level we can define it on a concrete level; that is, the actual objects that will be required to make up the components defined in Section 6.2. We begin by defining the implementation environment we will use, before specifying the individual objects to be constructed. This is followed by a discussion of how the components map to the

processes defined in Section 6.2 and the normal execution of the fortified system is presented.

6.3.1 Implementation Environment

This framework has been implemented using an EJB3.1 architecture and is designed to run on JBoss application server 6.0.0. This is a technology and middleware platform that is widely used for producing distributed applications. Heartbeat, state transfer and timer messages are sent and received via JMS.

A physical reboot mechanism has not been implemented, although the system will easily work with such a mechanism, as there is an option to add a wrapper class conforming to a simple reboot handler interface that will be called with the IP address of every node to be rebooted when a reboot request is sent.

6.3.2 Objects

Figure 6.9 illustrates the architecture of the proactive fortification framework that sits on each server node. The three message bean classes have been omitted for clarity, and in each case would handle messages from the appropriate incoming message queue and call the appropriate handler class.

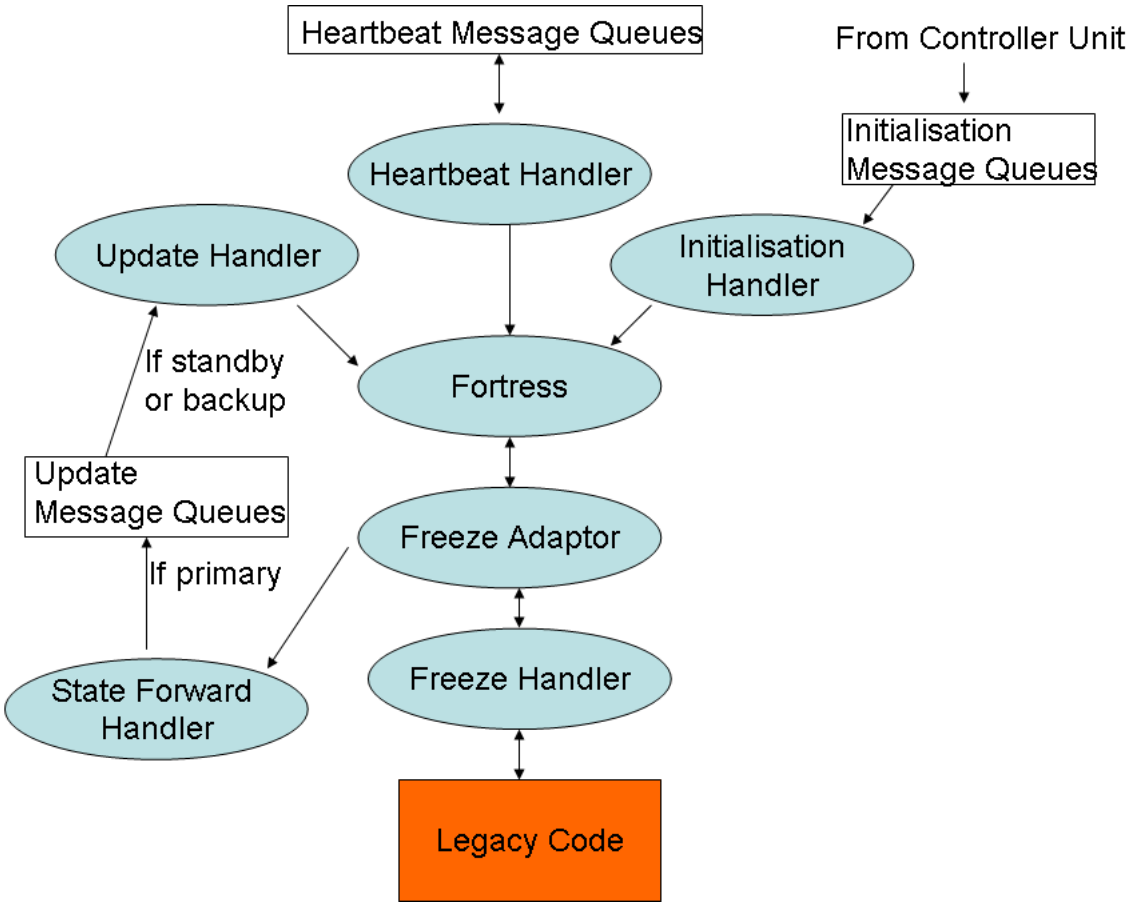
1. Fortress Timer

The Fortress Timer is a trusted process, situated in the controller unit, that informs each node when to change state. It starts with a list of candidate nodes, and allocates the first one to be the primary, the next two to be backups and the next three to be standbys. After a pre-set time period has elapsed the Fortress Timer instructs the primary and backups to refresh themselves, the standbys to become the new primary and backups, and three new nodes to become standbys.

Nodes can register themselves with the Fortress Timer at any point as being available. This does introduce the danger of a compromised machine re-registering itself without refreshing. However, in a production system it would be possible to use a hardware device to force each node to physically reboot after a stop message was received from the Fortress Timer. This possibility is catered for by providing a method that allows a Reboot Handler to be registered with the Fortress Timer. If a Reboot Handler is registered then every time a stop message is called the reboot node method of the Reboot Handler will be called with that node's IP address as a parameter.

2 Reboot Handler

Figure 6.9: Architecture of a Server Node



The Reboot Handler is an interface with one method allowing a node to be rebooted. This method takes the IP address of the node to be rebooted. A concrete implementation of the reboot handler can be provided for any production system to allow a physical reboot mechanism, situated in the Reboot Server (see Section 3.6), to be triggered, preventing intruded nodes from refusing to reboot.

The method of having the controller unit ping nodes that are known to have rebooted and re-register them when a ping response is received, as described in Section 3.6.4, can be implemented by having a separate process that is started by the reboot handler, performs the pinging, and then performs the registering with the Fortress Timer on behalf of the node.

3 Initialisation Message Bean

An Initialisation Message Bean is created every time a message is received from the Fortress Timer. It extracts the message's details and passes these details to the corresponding methods on the Initialisation Handler.

4 Initialisation Handler

The Initialisation Handler receives messages from the Fortress Timer via the Initialisation Message bean. These messages can be set-up messages or stop messages.

Set-up messages are subdivided into three types, Primary set-up, Backup set-up and Standby set-up. Each of these causes the Initialisation Handler to set up the Fortress, Heartbeat Handler and Update Handler in the correct manner for the message received.

Stop messages result in any final updates being sent, and then the node refreshing itself. Stop messages are expected for primary and backup nodes, but not for standbys, unless the whole system is being shut down. Instead a standby node would expect a Primary set-up or Backup set-up message to be the next thing it receives from the Fortress Timer.

The Initialisation Handler is implemented as a singleton bean.

5 Fortress

The Fortress contains state information about whether this node is the primary, a backup, or a standby, and lists of the other nodes. It also holds a reference to an appropriate Freeze Adaptor for the node. It has methods that enable the status of the node (primary, secondary, backup or stopped) to be changed, as well as methods to cause updates to be sent to other nodes.

The Fortress is implemented as a POJO, a single instance of it is registered with the Initialisation Handler, Heartbeat Handler and Update Handler. The existence of one Fortress is ensured by the fact that a single instance is registered with all three handlers, and every handler is a Singleton bean. This is reinforced by the Fortress class making use of the Singleton Factory design pattern.

6 Legacy Code State

The Legacy Code State is a wrapper class for two values; a byte array designed to contain any serialised object representing the state of a legacy system, and a Boolean to indicate whether this is the final update from a node. It is implemented as a POJO and designed to be used in Update messages.

7 Freeze Adaptor

The Freeze Adaptor holds references to an appropriate Freeze Handler and State Forward Handler.

It is able to set a given state in the legacy code by passing the state to the Freeze Handler or forward a state from the legacy code by calling the Freeze Handler and passing the information to the State Forward Handler. When an update is forwarded it can be declared to be either a standard update or the final update of this unit time step. The Freeze Adaptor also has methods that enable it to stop or start the legacy code by calling the Freeze Handler.

The Freeze Adaptor is implemented as a POJO, and a single instance of it is registered with the Fortress. The existence of one Freeze Adaptor is ensured by the fact that a single instance is registered with the Fortress of which there is only a single instance. This is reinforced by the Freeze Adaptor class making use of the Singleton Factory design pattern.

All code states handled by the Freeze Adaptor are encapsulated as instances of the Legacy Code State class.

8 Freeze Handler

The Freeze Handler can start and stop the legacy code, set the code state and return the code state, either flagged as a final update or not. It is implemented as an interface and specific Freeze Handlers (conforming to the Freeze Handler interface) will need to be produced for particular types of legacy code or hardware platform. The existence of a single instance of the Freeze Handler on a particular node is ensured by the fact that a single instance is registered with the Freeze Adaptor, of which there is only a single instance. This is reinforced by the Freeze Handler class making use of the Singleton Factory design pattern.

All code states handled by the Freeze Handler are encapsulated as instances of the Legacy Code State class.

9 State Forward Handler

The State Forward Handler has a method that takes a Legacy Code State and forwards it to all nodes listed as backups in the Fortress. It has a second method that takes a Legacy Code State and forwards it to all nodes listed as standbys in the Fortress. It is implemented as an interface enabling specific State Forward Handlers to be produced that make use of different messaging technology. It is expected that in a production system the State Forward Handler will be implemented to encrypt the contents of the messages that it sends. This will prevent an attacker that has succeeded in intruding into another part of the network from compromising confidentiality by reading state transfer messages, or compromising integrity by changing state transfer messages in an undetectable way. The use of encryption in the State Forward Handler is one way of satisfying the assumption of authenticated channels that was made in Section 6.1.3.

The existence of a single instance of the State Forward Handler on a particular node is ensured by the fact that a single instance is registered with the Freeze Adaptor, of which there is only a single instance. This is reinforced by the State Forward Handler class making use of the Singleton Factory design pattern.

All code states handled by the State Forward Handler are encapsulated as instances of the Legacy Code State class.

10 Heartbeat Message Bean

An instance of this Message Driven Bean is created when a heartbeat message is received from the primary. It extracts the contents of the message and passes them to the Heartbeat Handler.

11 Heartbeat Handler

The Heartbeat Handler can perform one of two roles when started. If the node is the primary then heartbeat messages are periodically sent to all backups.

If the node is a backup then a timer is set and every time a heartbeat message is received from the Heartbeat Message Bean it is re-set. If a heartbeat is received that originated from a node other than the primary then a method is called on the Fortress to change the identity of the current primary. If no heartbeat is received then the node waits for a pre-set period of time multiplied by the number of nodes senior to it. If no heartbeat is received in this time then it calls a method on the Fortress to become the primary and starts sending heartbeat messages to all nodes junior to it.

The Heartbeat Handler is implemented as a Singleton bean.

12 Update Message Bean

An instance of this Message Driven Bean is created when an update message is received from the primary. It extracts the contents of the message and passes them to the Update Handler.

13 Update Handler

The Update Handler can perform one of two roles when started. If the node is the primary then it periodically prompts the Fortress to send update messages. If the node is a backup or standby then it receives update messages from the Update Message Bean and sends them to the Fortress to update the system state.

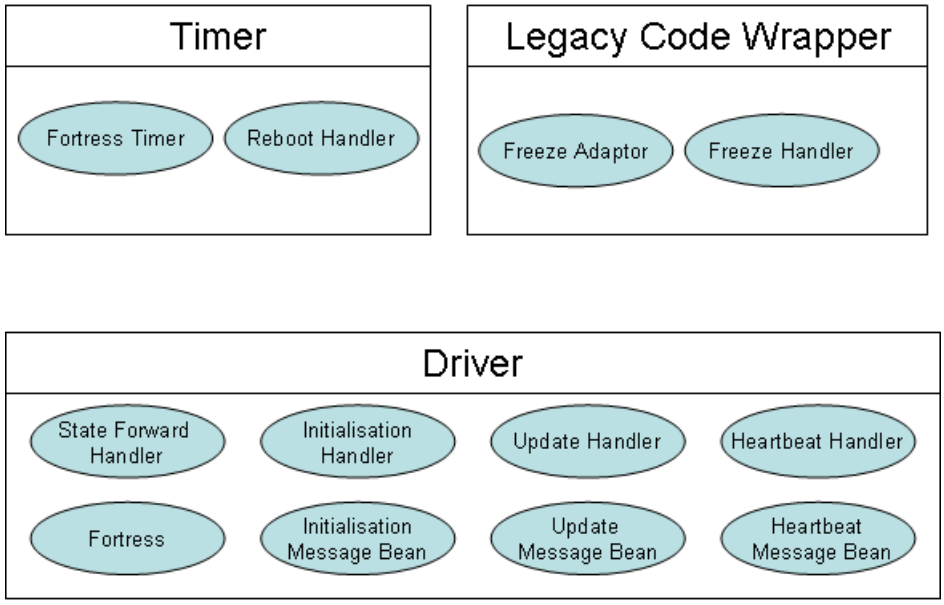
The Update Handler is implemented as a Singleton Bean.

14 Proxy Bean

The proxy bean is a stateless session bean built specifically for each given payload system. It provides a remote interface offering all of the operations offered by the payload system. It forwards requests it receives to the payload system and returns responses from the payload system to the client.

The proxy bean receives initialisation messages from the Fortress Timer. A start message informs it to begin forwarding client requests and sets the identities of the

Figure 6.10: Mapping Between Design and System Architecture Models



servers to forward requests to and receive responses from. A stop message informs it to stop forwarding client requests and clear the identities of the servers to forward requests to and receive responses from. These messages are handled by the Proxy Initialisation Message Bean and the Proxy Entity Bean.

15 Proxy Initialisation Message Bean

An instance of this Message Driven Bean is created when an initialisation message is received from the Fortress Timer. It creates a Proxy Entity Bean from the information contained in the message and commits it to persistent storage. As the data in the Proxy Entity Bean refers to the current state of the proxy, a previous Proxy Entity Bean is deleted if it exists.

16 Proxy Entity Bean

This Entity bean stores the current state of the proxy server it is located on. It contains two variables; a Boolean flag to indicate whether the proxy is currently active, and a list of IP addresses of the current servers. A start message from the Fortress Timer results in the Boolean flag being set to true and the IP addresses being set to those contained in the message. A stop message from the Fortress Timer results in the Boolean flag being set to false and the list of IP addresses being deleted.

6.3.3 Mapping Between Design and System Architecture Models

Here we consider each component of the design and show the pieces of the system architecture that it includes. This relationship is illustrated in Figure 6.10.

1. Timer

The Fortress Timer class implements all of the functionality of the timer component, with the Reboot Handler class performing any triggering of hardware based reboots.

2. Legacy Code

The legacy code is not implemented as part of the Fortress framework.

3. Legacy Code Wrapper

The legacy code wrapper is implemented by the Freeze Handler and Freeze Adaptor classes.

4. Driver

The driver consists of eight classes. Most of its functionality is contained in the Fortress class, with the ability to receive messages from the timer being implemented by the Initialisation Handler and Initialisation Message Bean classes. The ability to send and receive updates is provided by the State Forward Handler, Update Handler and Update Message Bean classes. The ability to decide when it is necessary to switch from backup to primary status is provided by the Heartbeat Handler and Heartbeat Message Bean classes.

5. Proxy

The Proxy Bean class implements the main functionality of the proxy component, forwarding client requests and server responses. The Proxy Message Bean class handles proxy initialisation, and stores the necessary data in the Proxy Entity Bean class.

6.3.4 Execution

As in Section 6.2.8, we present an example of the normal execution of a proactively fortified system assuming progressive transfer. The execution of the Fortress Timer is shown in Figure 6.11, the execution of the primary server is shown in Figure 6.12 and the execution of each backup server is shown in Figure 6.13.

Figure 6.11: Activity Diagram of the Execution of the Fortress Timer

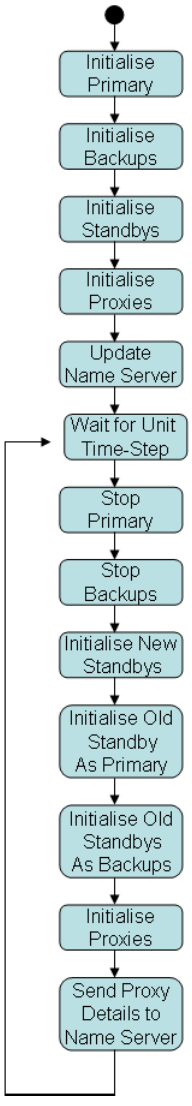
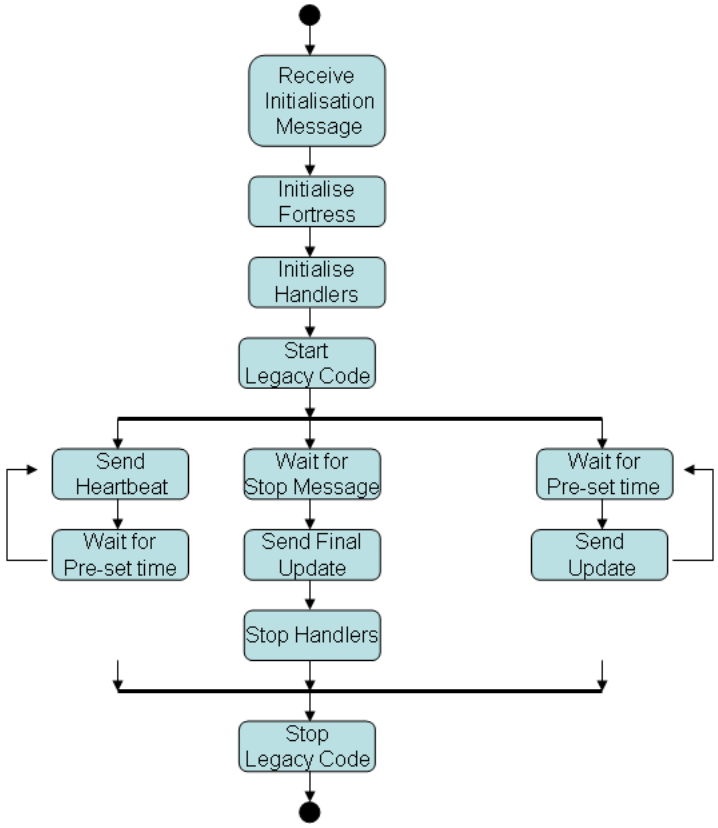


Figure 6.12: Activity Diagram Showing Execution of the Primary



The Fortress Timer starts with a list of available servers and available proxies. It sends requests to the first server to become a primary, the next two servers to become backups and the next three servers to become standbys. This is followed by requests to the first three proxies to become proxies. It then sets a timer for the inter-migration period.

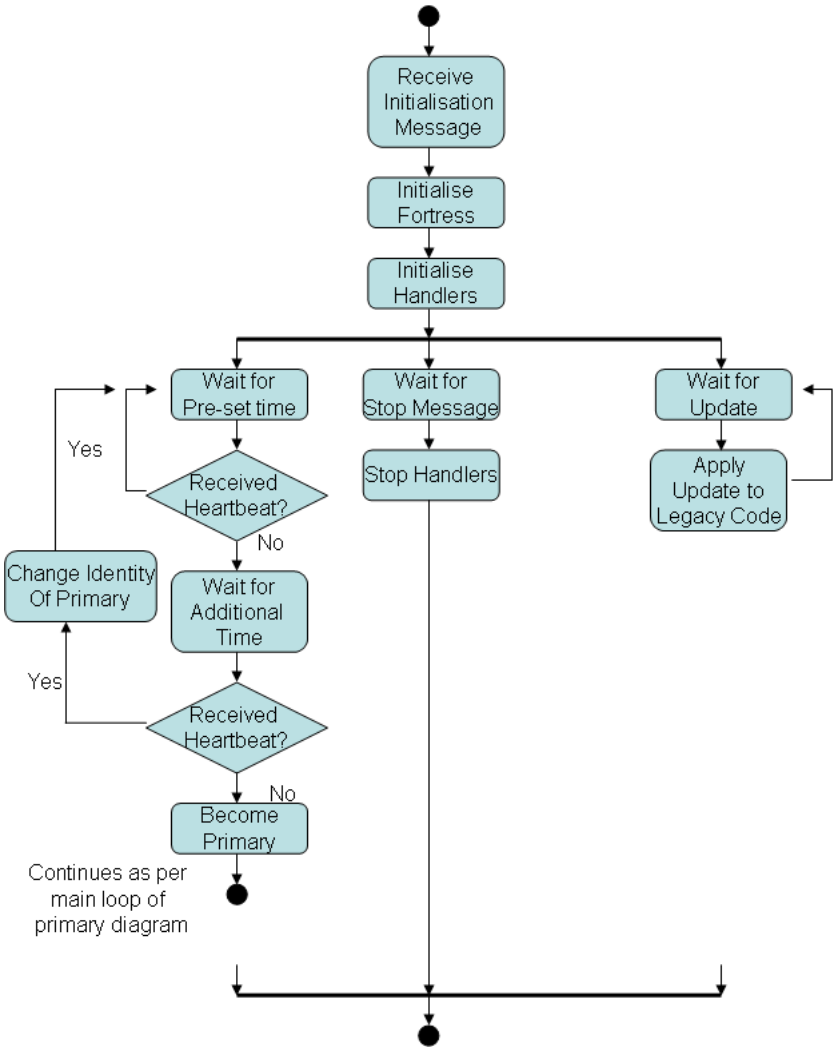
The Initialisation Handler for the primary receives a request from the Fortress Timer. It initialises the Fortress with the lists of backups and standbys, the correct Freeze Handler (itself initialised with the correct Freeze Adaptor and State Forward Handler) and registers the Fortress with itself, the Heartbeat Handler and the Update Handler. It then starts the Heartbeat Handler and Update Handler.

The Heartbeat Handler sends a Heartbeat message to the backups then sets a timer to instruct itself to send the next Heartbeat message. The Update Handler sets a timer to instruct itself when to send the first Update message to the standbys.

The Initialisation Handler for each backup receives a request from the Fortress Timer. It initialises the Fortress with the lists of primary, backups and standbys, the correct Freeze Handler (itself initialised with the correct Freeze Adaptor and State Forward Handler) and registers the Fortress with itself, the Heartbeat Handler and the Update Handler. It then starts the Heartbeat Handler.

The Heartbeat Handler sets a timer after which it will need to identify the new primary, and resets this timer as soon as it receives a Heartbeat message from the

Figure 6.13: Activity Diagram of Execution of Backup



primary. The Update Handler processes any update message that it receives.

The Initialisation Handler for each standby receives a request from the Fortress Timer. It initialises the Fortress with the lists of primary and backups, the correct Freeze Handler (itself initialised with the correct Freeze Adaptor and State Forward Handler) and registers the Fortress with itself and the Update Handler.

The Update Handler processes any Update messages that it receives.

Each proxy server receives a request to become a current proxy. This request includes the addresses of the current primary and backups. The proxies will now forward any requests they receive to the primary and backups, and will forward any responses they receive from the primaries and backups to the client.

When the inter-migration period has elapsed the Fortress Timer sends a stop message to the primary, each of the backups and each of the proxies. If a Reboot Handler is registered then this will also be triggered, causing messages to be sent to the reboot server instructing it to reboot all of the servers and proxies after a sufficient time interval to allow migration. The primary will send update messages to the standbys, with the next backup taking over this role if it does not receive a message within a time-out period and can hence assume that the primary has crashed.

The Fortress Timer will send a request to the first standby to become the primary, the other two standbys to become backups and three new servers to become standbys. Requests will be sent to the next three proxies to become the current proxies, and they will be given the addresses of the new primary and backups. The Fortress Timer will then set a new time-out for the inter-migration period.

Chapter 7

State Transfer Overhead: A Small-Scale Application

This chapter presents an experimental evaluation of the performance overhead caused by the use of the FORTRESS system relative to that of a primary-backup system.

We begin by defining the components that will be needed to produce a server application suitable for measuring the overhead of the proactive fortification framework. We then define the components that will be needed for a suitable client system for using in overhead measurement. This is followed by an outline of our measurement strategy and the physical systems used.

Finally, we present the results of latency and throughput tests as three key intervals are varied; the time between periodic updates being sent to the new set of servers under the progressive transfer scheme, the time between heartbeat messages being sent by the current primary to the current backups and the time between node replacement.

These results show that configuring the time between node replacements can have a significant effect on performance overhead, resulting in a trade-off needing to be made between the increase in intrusion resilience from shortening the time and the increase in efficiency from lengthening it. The results also show that update interval and heartbeat interval can have an effect on performance overhead and will need to be configured correctly for maximum efficiency.

7.1 Server Application Architecture: Simplifying Assumptions

We have made several simplifications to the server application used to measure the overhead of our proactive fortification framework. These simplifications have been made for ease of measurement and implementation. The proactively fortified system is compared to the same server application without proactive fortification, i.e. a

simple primary-backup replicated server system subject to neither randomisation nor periodic re-randomisation.

1. Database access is handled by EJBs

The EJB framework is used for all persistence. This means that in our test application the values to be stored are modelled as entity beans, and the persistence framework is used to store them.

2 Every Node has its own Persistent Store and Updates are Propagated to Backups Whenever an Entity is updated

This means that we are considering a client-server architecture rather than a three-tier architecture. Rather than using re-written database drivers to prompt an update every time an entity bean is modified, we instead use an interceptor to capture the change. This may not be possible in a production system, as many languages do not provide the interceptor functionality that Java EE provides. There is also an issue of understanding which entity beans need interceptors if considering a large system. We have noted in Section 6.2.1 that proactive fortification may need to be undertaken by system architects who do not have a perfect understanding of the internals of the legacy system.

However, the use of interceptors is fundamentally equivalent to the technique of re-writing database drivers discussed in Section 6.2.4.1 and in [32], suggesting that it is appropriate to use in an evaluation system.

3 Application State is Transferred through the Transfer of the Contents of the Relevant Entity Beans

Our evaluation application is sufficiently simple that all information about current processing is held within the set of entity beans. Transferring the information from these entity beans removes the need for us to produce an efficient application for serialising program execution state for an EJB based application. We note that the marshalling and unmarshalling performed in this transfer of entity beans will not be as efficient as the schemes we have discussed for transferring and transforming program state between nodes. Hence this simplification has the potential to reduce the performance of our system relative to what could be achieved with a more efficient transfer scheme.

4 Re-randomisation of Servers is not Performed during the Evaluation Procedure

Our scheme makes use of differently randomised servers being available every time system migration takes place. This means that, re-boot and re-randomisation can

occur while machines are not in use (using a similar scheme to ensure re-boot as that in [55]). Thus, while the periodic re-boot and re-randomisation of servers is vital to the security of the scheme, it will make no difference to the state transfer overhead of the scheme, as long as there are sufficiently many servers available to allow re-boot and re-randomisation to occur while machines are not in use. Hence this overhead evaluation does not require servers to be re-randomised.

5 Co-Location of Proactive Fortification Framework

All of the proactive fortification framework for a given server runs from the same instance of JBoss, on the same physical machine as the legacy code. A production system may prevent timing attacks by requiring that the framework be located in a hardware module that guarantees time bounds on message delivery, and retrieving and setting system state on correct systems. Such specialist hardware would be system specific, and would also be likely to improve the efficiency of the proactive fortification framework, so the absence of this hardware in our overhead evaluation would not invalidate results showing that the proactive fortification framework has sufficiently low overhead to be practical.

7.2 Server Application System Architecture Components

The server application consists of the following components:

1. Server Application Bean

This provides a simple service allowing clients to request a job, which contains a value supplied by the client to represent some job data, or request the information about an existing job. Requests are given with a client provided request number. This request number will be stored with the job data in a Data Entity Bean.

The unique request number is used to differentiate between new jobs, and jobs that are received multiple times through several proxies. We assume in our evaluations that this will always be unique, but in practice the proxy could append a unique client identifier to the start of each request number so that a client could not accidentally use the same request number as another client. Even with this precaution it is possible for a malicious client to duplicate job numbers by submitting several jobs with the same number, knowing that this is likely to result in identical job numbers as they will have the same client identifier appended to them. However, all this would enable the malicious client to do is have a duplicate job refused, which will not compromise system integrity or confidentiality and will only deny availability to that malicious client for that duplicate job.

2 Data Entity Bean

A Data Entity Bean is used to represent each job in the system. It contains the unique job identifier, the job data and a time-stamp for when the job was created. An interceptor is used to capture the action of each data entity bean being created or modified. This interceptor checks to see if this node is currently the primary, and if it is then it a creates a copy of the data entity bean it is called on in and wraps it in a Legacy Code State, then calls the appropriate method on the State Forward Handler to pass this Legacy Code State to the backup nodes.

3 Forward Bean

The Forward Bean is a stateless session bean that allows the current Data Entity Beans, or a subset of them from a particular time period, to be retrieved or replaced, and hence the system state retrieved or altered. It has a get data method that can return a Server Application Data Wrapper containing a list of Data Entity Beans, and a set data method that takes a Server Application Data Wrapper and adds its contents to the stored Data Entity Beans, replacing any existing beans that also occur in the Server Application Data Wrapper. The get data method can be given a parameter to specify the earliest time and date of modification from which to include Data Entity Beans in the Server Application Data Wrapper.

4 Payload Data Wrapper

The Server Application Data Wrapper is a wrapper class that contains a list of Data Entity Beans. It is serialisable and intended to be wrapped by a Legacy Code State.

7.3 Server Application Specific Fortress Components

The following implementations of Fortress framework interfaces were created for use with the server application.

1. Server Application State Forward Handler

The Server Application State Forward Handler implements the State Forward Handler Interface defined in Section 6.3.2. It implements the method to forward state by retrieving the lists of backups junior to this node and standbys, and multi-casting the Legacy Code State to them, using JMS.

2 Server Application Freeze Handler

The Server Application Freeze Handler implements the Freeze Handler Interface defined in Section 6.3.2. It implements the method to retrieve the state by calling the get data method of a Forward Bean and wrapping the Server Application Data Wrapper that is returned in a Legacy Code State.

The method to set the state is implemented by retrieving the Server Application Data Wrapper from the Legacy Code State that it receives in an update message and passing it to the set data method of a Forward Bean.

7.4 Client Components

The following component was created to simulate the action of a client accessing the application:

1. Client Bean

The client bean is a stateless session bean offering a local interface to start the overhead measurement procedure. It performs a series of requests on the payload system, alternating between set and get requests and records the start and finish times for every request. The tests can be started by a jsp page which sends a start request to the local interface. This jsp page plays no part in the actual measurement procedure, as the client bean performs all of its own timing, so there will be no performance issues due to the overhead of using a web interface.

7.5 Measurement Strategy

Our measurement methodology involved measuring three metrics; latency, throughput and correctness of responses and state. Here we describe the rationale behind measuring each of these factors, along with the methods used. We then describe the physical set-up of our measurement system.

Other possible metrics that were considered were hardware cost to handle a fixed rate of requests and hardware cost to provide a fixed percentage of responses within a fixed time. These metrics were rejected as, for a given system, they can be calculated from the throughput and latency measurements.

7.5.1 Latency

Latency is the time taken for a response to be returned from a system. That is, the total round trip time from the request being sent to the response being received. Measuring latency allows us to examine how the use of proactive fortification affects

the actual time that a user will have to wait to receive a response to their request, and what effect this is likely to have in practice.

Latency can be measured by logging the time at which each request is sent and matching this up with the time at which the corresponding response is received. The latency of requests that are lost during reboot and have to be re-sent by the client are measured as being sent when they were first sent, and received when the response is received after re-sending.

7.5.2 Throughput

Throughput is the number of requests that can be handled in a given period of time. Measuring throughput enables us to examine how the maximum number of requests that can handled at once is affected by proactive fortification. This can have direct financial implications as, if a particular throughput level is needed and this is reduced by proactive fortification, the solution is likely to involve the provision and running of extra or faster hardware to make up the shortfall in throughput.

Throughput can be measured by generating requests from a number of clients at the same time, and logging the time at which each request is sent and the time at which the corresponding response is received. Collating these request/response pairs allows us to calculate how many requests were handled on average in a given period of time. Requests that are lost during reboot and have to be re-sent by the client are measured as being sent when they were first sent, and received when the response is received after re-sending.

7.5.3 Correctness of Responses and State

The correctness of responses and state is a variable that is either true or false. Either the responses and the system state at the end of a period of processing are correct, or they are incorrect. Measuring the correctness enables us to ascertain whether the system is behaving correctly, or whether proactive fortification has introduced errors that were not present in the legacy system.

Correctness can be sampled by examining the logs of requests and responses in the latency and throughput tests and comparing the responses received to the values that we would expect them to be, based on the requests that have been sent. The final state of the system can also be compared to the values we would expect them to be by examining the contents of the database.

7.5.4 Overhead Measurement System

Figure 7.1: Measurement - First Unit Time-Step

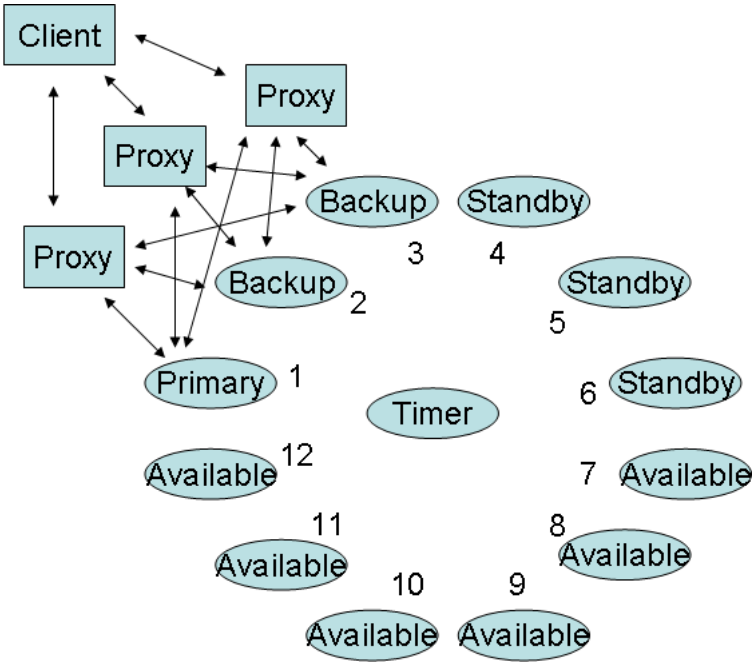


Figure 7.2: Measurement - Second Unit Time-Step

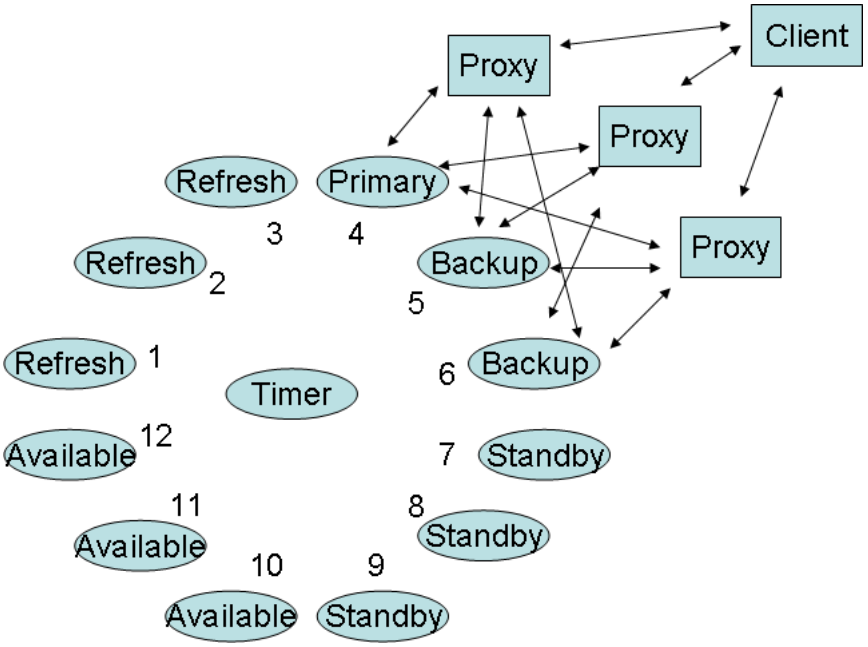
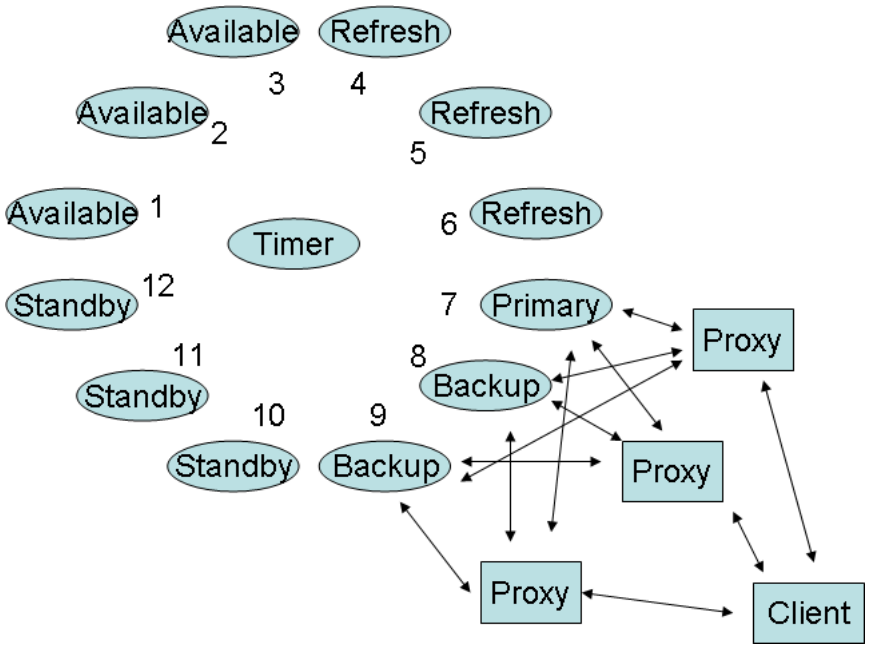


Figure 7.3: Measurement - Third Unit Time-Step



A system of 12 servers was set up, on separate machines. This ensures that there are always sufficient machines to have 3 servers in use as the primary and backups, 3 servers in use as the standbys, and 3 servers available for the next migration, even if it takes longer than one migration period for a server to be refreshed. We assume that it does not take longer than 2 migration periods for a server to be refreshed and this has always been the case in practice.

Six proxies were then set up on separate machines, and a client on a further separate machine. Five instances of the client code were started simultaneously. Each instance of the client made a series of 100000 requests, alternating between increment requests and get value requests. The sending and receiving of requests were de-coupled, to ensure that the latency did not reduce the speed at which the client could send requests.

In all cases, sending this number of requests takes a large enough period of time that the processing time included at least one migration. The average latency and throughput were calculated from the start and end times of the requests sent. The contents of the requests and responses were stored, along with the contents of the database on the primary and backups after the last request was received. The system configuration is demonstrated for the first three unit time-steps in Figures 7.1, 7.2 and 7.3. These figures display all server nodes, available or being refreshed, but only display one client and the current proxy nodes for the unit time-step being considered.

Then, three servers were set up in a primary-backup configuration, and five instances of the client on a separate machine was used to send the same number of requests. The average latency and throughput were again calculated from the start and end times of the requests sent, and the contents of the requests, responses and databases were again stored.

This allowed a comparison of average latencies and throughputs to be made between the same payload system with and without Proactive Fortification.

The system is illustrated through 3 unit time-steps in Figures 7.1, 7.2 and 7.3.

7.5.5 Experimental Methods

Three key system parameters were identified:

- the time between replacement of nodes (migration interval)
- the time between updates being sent to the standby nodes (update interval)
- the time between heartbeat messages being sent from the primary to the backups (heartbeat interval).

The last two intervals are common to both fortified and unfortified systems, and the first interval is unique to the fortified system.

A set of experiments was run for each interval, where it was varied while the other two were held constant. A further set of experiments were run for the case where a primary-backup system with no migration was used, to give a baseline to compare the other values to.

We also made use of the stored requests, responses and the contents of the databases used by the primary and backups after the end of each experiment to check correctness. In every case each response was as expected, and the databases all contained the correct data entity beans. This allowed us to conclude that state transfer is not introducing any errors into the system state, either in the primary-backup system or the proactively fortified system, and that even when migration is taking place, all requests are eventually being processed.

The experiments performed for each system parameter are as follows:

7.5.5.1 Update Interval

The migration interval was held constant at 150 seconds, and the heartbeat interval was held constant at 4 seconds. The update interval was varied between 500ms and 150 seconds, with one experiment being performed for each value. We note that an

update interval equal to the migration interval is essentially identical to using single transfer.

Each experiment consisted of each of five client instances making a series of 100000 requests, alternating between increment requests and get value requests.

7.5.5.2 Heartbeat Interval

The migration interval was held constant at 150 seconds, and the update interval was held constant at 40 seconds. The heartbeat interval was varied between 500ms and 6 seconds, in steps of 500ms. A second set of experiments was performed without system migration, and hence without any updates being sent to standby nodes, to provide comparison values for an unfortified system. The heartbeat interval was varied between 500ms and 6 seconds, in steps of 500ms.

Each experiment consisted of each of five client instances making a series of 100000 requests, alternating between increment requests and get value requests.

The experiment using an unfortified system with a heartbeat interval of 4 seconds also provided a comparison value to use in the update interval and migration interval experiments.

7.5.5.3 Migration Interval

The update interval was held constant at 10 seconds, and the heartbeat interval was held constant at 4 seconds. The migration interval was varied from 150 seconds to 20 seconds in 10 second steps.

Each experiment consisted of each of five client instances making a series of 100000 requests, alternating between increment requests and get value requests.

7.6 Results

7.6.1 Update Interval

The latency and throughput were found to be strongly correlated to the interval between the last update and the time of system migration, with the optimum values occurring when this interval was 4 seconds.

The latencies measured, expressed as percentage increases over the latency measured for a system without migration, are shown in Figure 7.4. The absolute values, along with 95% confidence intervals for each value are shown in Appendix F.1.

The percentage increase in latency has two relatively low areas, sandwiched between

Figure 7.4: Percentage Increase in Latency as Update Interval Varies

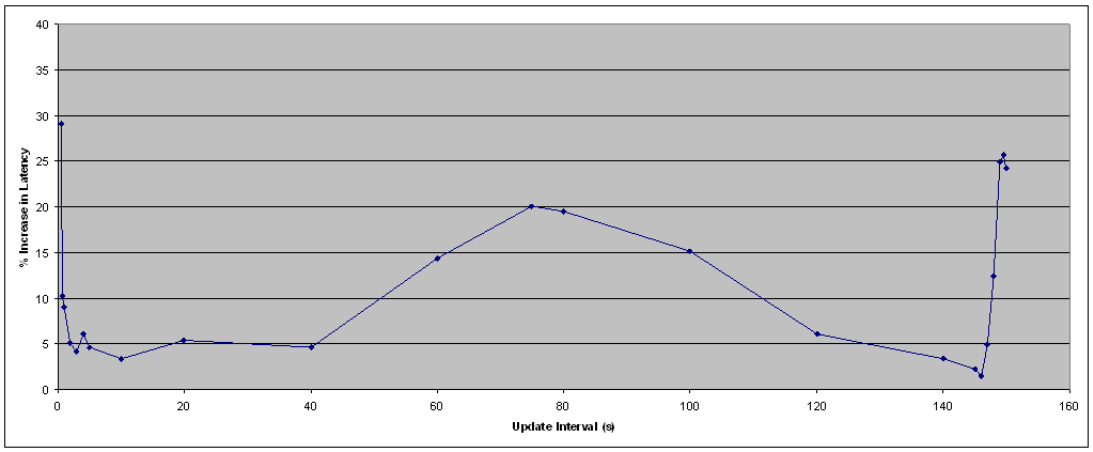
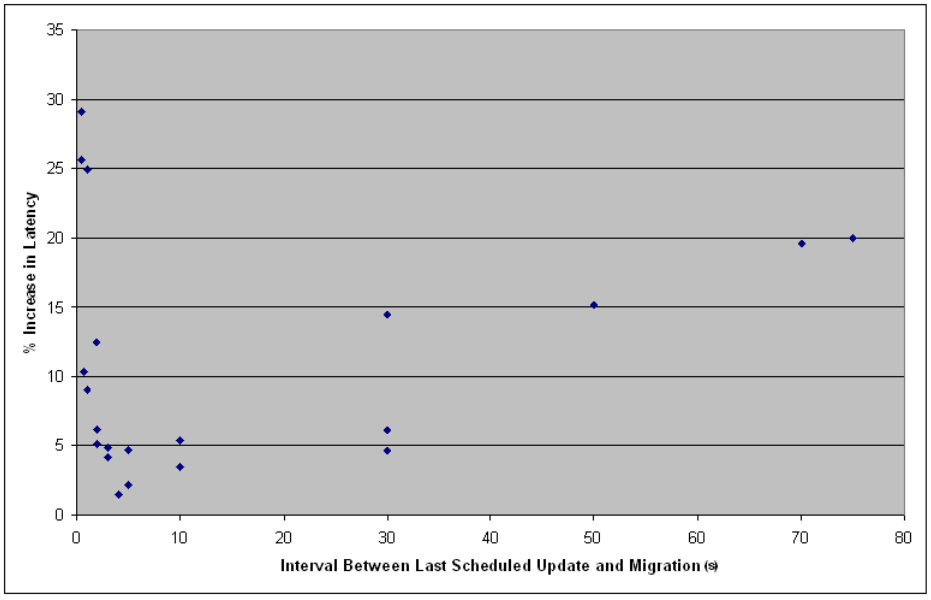


Figure 7.5: Percentage Increase in Latency as Interval Between Last Scheduled Update and Migration Varies



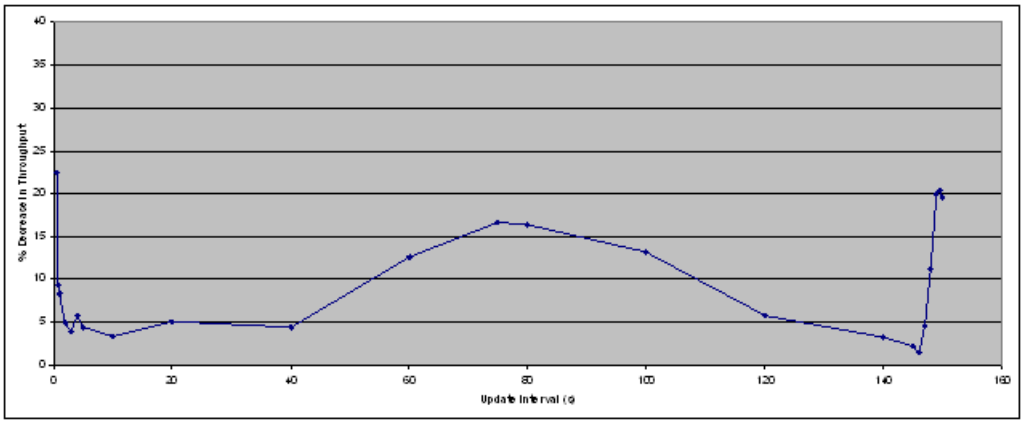
very high end points and a higher central Section. The complex way in which latency varies with update interval suggests that latency may not be linearly affected by update interval, but instead by some property that emerges from the update interval. One property that varies non-linearly with update interval is the interval between the last scheduled update and the final update sent at migration. A manipulation of the data to give the percentage increase in latencies relative to the interval between the last scheduled update and final update sent at migration is shown in Figure 7.5.

Figure 7.5 shows a linear positive relationship between the interval and the percentage increase in latency, for intervals larger then 4 seconds, and also a linear negative relationship between the interval and the percentage increase in latency for intervals smaller than 4 seconds. This suggests that smaller intervals between the last scheduled update and the final update for migration improve efficiency, up until a cut-off point where they become too close, after which smaller intervals decrease efficiency.

Table 7.1: Correlation Coefficients for Comparison between Latency and Interval between Last Update and Migration

Interval Between Last Scheduled Update and Migration	Correlation Coefficient
≥ 4	0.9346
< 4	-0.7980

Figure 7.6: Percentage Decrease in Throughput as Update Interval Varies



This intuition can be reinforced by calculating correlation coefficients for these two sections of the data. These correlation coefficients are shown in Table 7.1 and demonstrate a strong positive correlation between the interval and the percentage increase in latency when the interval is 4 seconds or larger, and a moderate negative correlation between the interval and the percentage increase in latency when the interval is less than 4 seconds.

The throughputs measured, expressed as the percentage decreases compared to the throughput measured for a system without proactive fortification are shown in Figure 7.6. This graph has a strong resemblance to Figure 7.4, so we again consider the interval between the last update and migration, as we did with the latency. A manipulation of the data to give the percentage decrease in throughput relative to the interval between the last scheduled update and final update sent at migration is shown in Figure 7.7.

Here we see the percentage decrease in throughput becoming larger as the interval increases, for interval values in the range 4s to 80s and the percentage decrease in throughput becoming larger as the interval decreases for interval values in the range 0s to 4s. Correlation coefficients for these intervals are shown in Table 7.2 and suggest a strong positive linear correlation between interval and decrease in throughput for intervals between 4s and 80s and a strong negative linear correlation between interval and decrease in throughput for intervals between 0s and 4s.

Figure 7.7: Percentage Decrease in Throughput as Interval Between Last Scheduled Update and Migration Varies

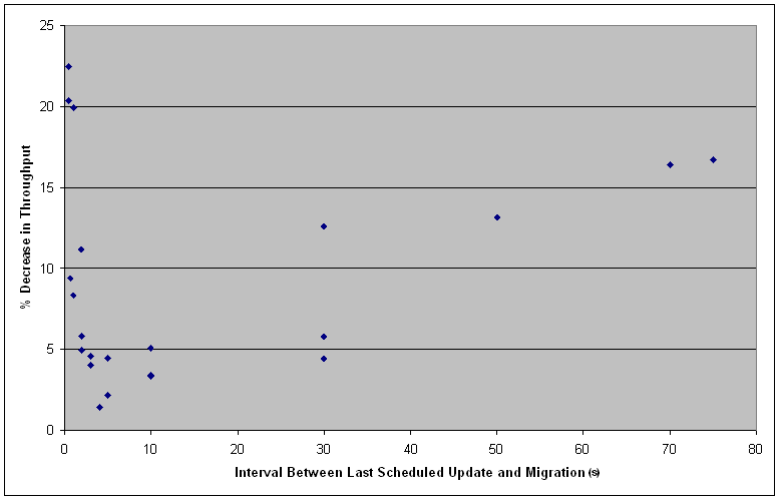


Table 7.2: Correlation Coefficients or Comparison between Throughput and Interval between Last Update and Migration

Interval Between Last Scheduled Update and Migration	Correlation Coefficient
≥ 4	0.925
< 4	-0.808

7.6.2 Heartbeat Interval

The heartbeat intervals were found to have a weak correlation with performance overhead, with the system becoming more efficient as heartbeat interval increased in the range 500ms to 6 seconds.

The absolute values obtained are presented in Appendix F, along with 95% confidence intervals for each value.

Correlation coefficients for these results are shown in Table 7.3. This shows a weak negative correlation between latency and heartbeat interval and a weak positive correlation between throughput and heartbeat interval with a migration interval of 150s over the range considered. It also shows no correlation between latency and heartbeat interval and no correlation between throughput and heartbeat interval in a system without migration over the range considered.

The increase in latency caused by proactive fortification is shown in Figure 7.8. None of the heartbeat intervals considered results in an increase in latency of more than 8%. The decrease in throughput caused by proactive fortification is shown in Figure 7.9. None of the heartbeat intervals considered results in a decrease in throughput of more than 7%. This shows that the overhead of proactive fortification is minimal for a 150 second migration interval and 40 second update interval, for any of the range of heartbeat intervals considered.

Table 7.3: Correlation Coefficients for the Comparison of Latency and Throughput with Heartbeat Interval

Data Series	Correlation Coefficient
Latency with 150s Migration Interval	-0.46602
Throughput with 150s Migration Interval	0.46627
Latency without Migration	-0.10919
Throughput without Migration	0.105922

Figure 7.8: Percentage Increase in Latency as Heartbeat Interval Varies

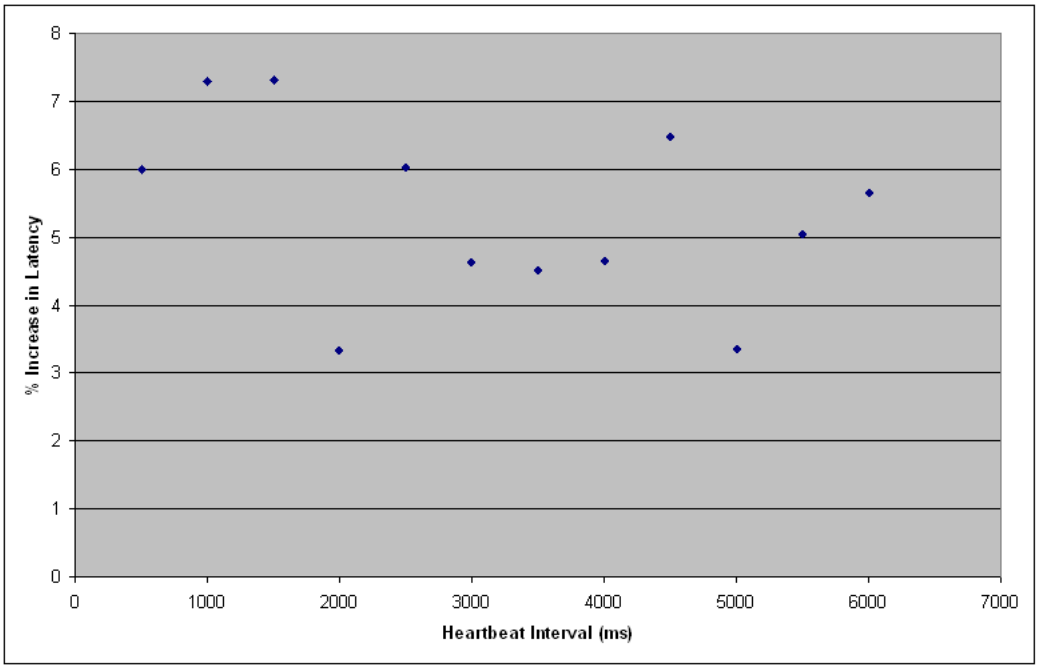


Figure 7.9: Percentage Decrease in Throughput as Heartbeat Interval Varies

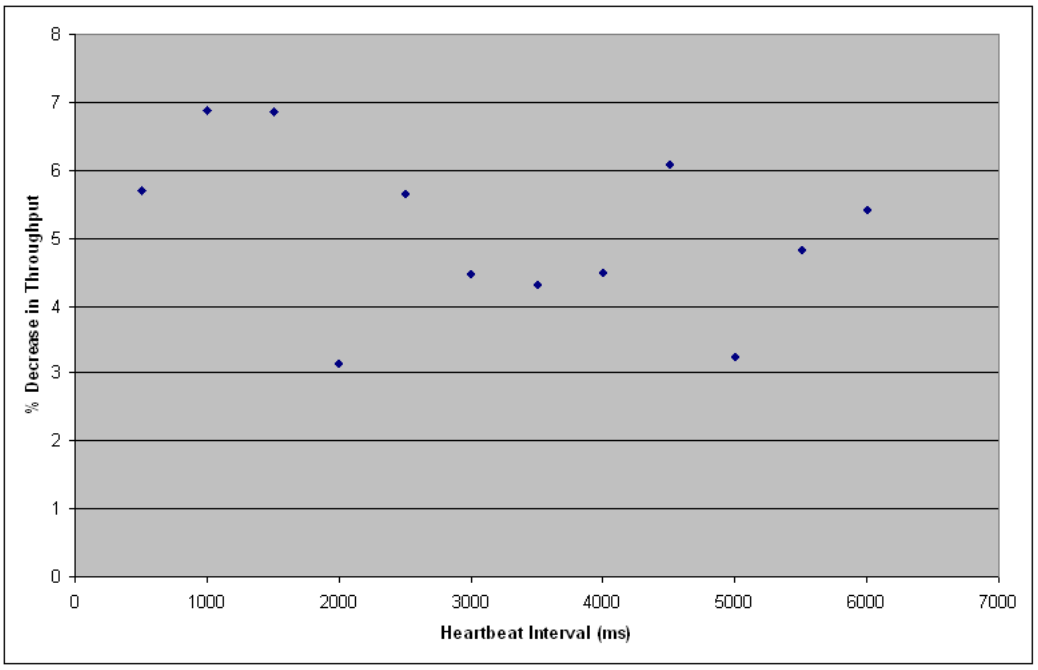
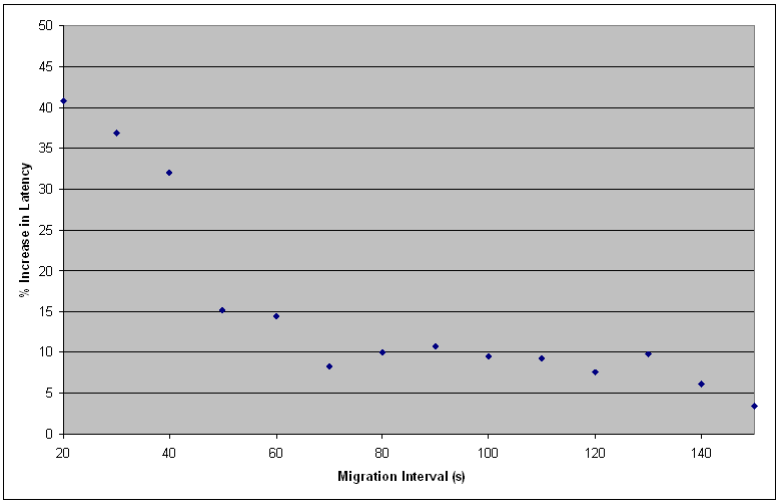


Table 7.4: Correlation Coefficients for the Comparison of Latency and Throughput with Migration Interval

Data Series	Correlation Coefficient
Latency	-0.84035
Throughput	0.863091

Figure 7.10: Percentage Increase in Latency as Migration Interval Varies



7.6.3 Migration Interval

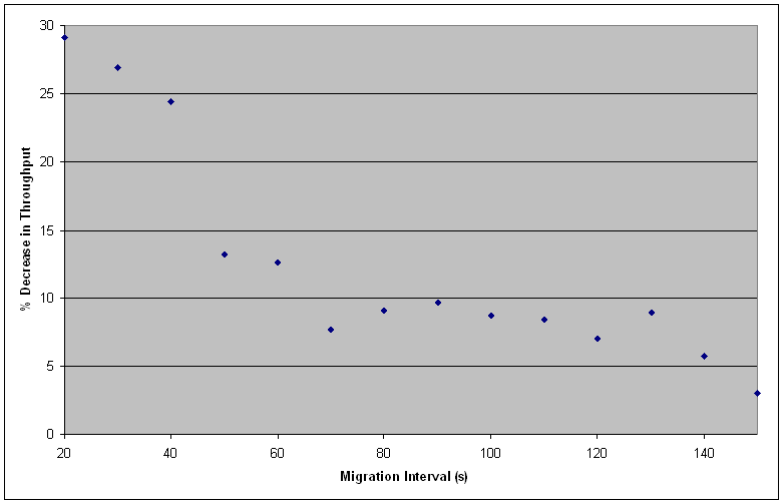
The migration intervals were found to have a strong correlation with performance overhead, with efficiency increasing as the migration interval increased.

The absolute values obtained, along with 95% confidence intervals are presented in Appendix F.

Correlation coefficients for these results are shown in Table 7.4. This shows a strong negative correlation between migration interval and latency and a strong positive correlation between migration interval and throughput for the range considered.

The increase in latency caused by proactive fortification is shown in Figure 7.10. This shows that the increase in latency is fairly small for migration intervals of 50 seconds and larger. The overhead becomes considerably larger for migration intervals of less than 50 seconds. The decrease in throughput caused by proactive fortification is shown in Figure 7.11. This shows that the decrease in throughput is fairly small for migration intervals of 50 seconds and larger. The overhead becomes larger for migration intervals of less than 50 seconds, but is still less than 30% for a migration interval of 20 seconds.

Figure 7.11: Percentage Decrease in Throughput as Migration Interval Varies



7.7 Summary

The efficiency measurement shows us that, for a migration interval of 150 seconds it is possible, through careful tuning of update interval and heartbeat interval to reduce the increase in latency and decrease in throughput relative to that of a primary-backup system to less than 5%. Tuning of the update interval, or more specifically the interval between the last update and the time of migration, can make a significant difference to performance, while tuning of the heartbeat interval has the potential to make a difference of up to 3%-4% in both latency and throughput.

As the migration interval is reduced to 100 seconds it is still possible to achieve an increase in latency and decrease in throughput of less than 10%.

As the migration interval is reduced to 70 seconds it is still possible to achieve a decrease in throughput of less than 10%, although the increase in latency does stray into the 10%-15% band for some values in this range. Further reduction of the migration interval to 50 seconds, still results in a decrease in throughput of less than 15% and an increase in latency of just over 15%. Even smaller migration intervals result in larger increases in latency and larger decreases in throughput, but even a migration interval of 20 seconds results in a decrease in throughput of less than 30% and an increase in latency of less than 45%.

The practical implications of these overheads is very much dependent on the way in which the systems to be proactively fortified are used. For example, in the test system studied, a 40.83% increase of latency results in latency jumping from 81.8ms to 115.2ms, an increase of 33.4ms. If a user is making a request of the system and using that request when it is returned, a 33.4ms increase in the time to receive it will be irrelevant, and unnoticeable. On the other hand, if another system is making a series of requests, each depending on the result of the request before, then these 33.4ms delays may add up and cause an actual performance decrease.

The decrease in throughput is more likely to be a factor to be considered in most production systems. Generally, a system will be designed to have the minimum amount and specification of hardware to handle expected demand. Hence a significant decrease in throughput will result in a significant increase in the amount and specification of hardware required to handle expected demand.

One additional factor worth considering here is why the migration interval would be reduced. The migration interval is essentially the window of time that an attacker has to compromise nodes before they are re-randomised. Hence, it needs to be short enough to make the task of compromising the system sufficiently difficult for an attacker. This means that the migration interval only needs to be made smaller in response to a more powerful attacker, or the need for an even stronger likelihood that the system will not be fatally intruded.

This suggests that it is possible to choose a point in the trade-off between cost and intrusion resilience, and provision sufficient amounts and powers of hardware to allow the system to handle peak demand with a sufficiently small migration interval to provide the degree of intrusion resilience required.

Chapter 8

Applying Proactive Fortification in a Large Scale Web Application Context

This chapter considers the possibility of applying the FORTRESS approach to a large scale web application, examining both the possible differences in intrusion resilience and in performance overhead.

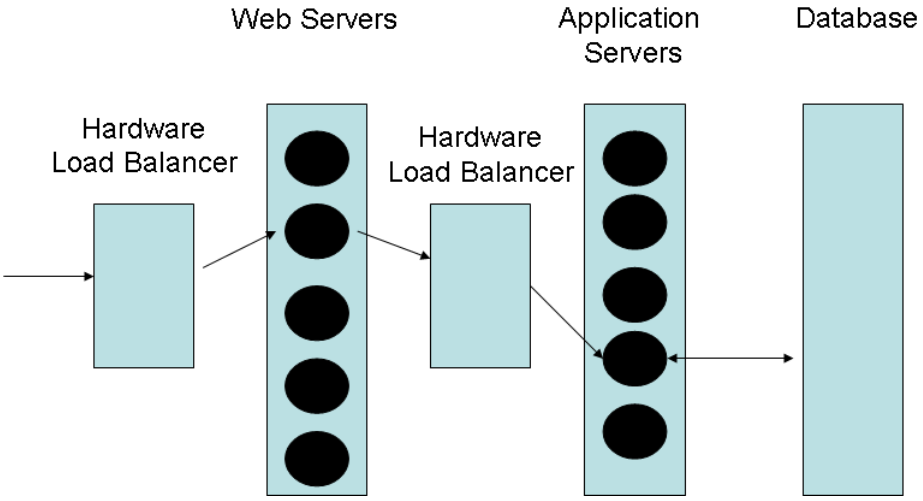
We begin by considering the structure of large scale web applications, and the information flow during normal operation. These large scale web applications generally make use of a number of application servers for load balancing purposes, all of which share a separate database layer. This is followed by an analysis of how this architecture affects a malicious attacker's chances of success relative to the smaller distributed systems we have considered previously. We show that an attacker will, in general, find a large scale web application with proactively fortified application servers more challenging to attack than the system we considered in Chapter 6 and 7 .

Next we show how the clustering and load balancing features in Apache Tomcat and Apache HTTP server can be used to produce a lightweight implementation of a proactive fortification system for a large scale web application, and outline an overhead measurement strategy for comparing the efficiency of such a system to the unfortified system it is based on. Finally, we present overhead data from a series of overhead evaluation experiments based on a simple online shopping application.

This overhead data was obtained both for static pages and dynamic, user data dependent pages. It shows that state is correctly maintained across migrations, and demonstrates latency and throughput changes relative to an unfortified system as migration and heartbeat interval are varied.

This overhead data shows that, for both static and dynamic pages, configuring the time between node replacements can have a significant effect on performance overhead, resulting in a trade-off needing to be made between the increase in intrusion resilience

Figure 8.1: Large Scale Web Application Architecture



from shortening the time and the increase in efficiency from lengthening it. The results also show that the heartbeat interval can have an effect on performance overhead and will need to be configured correctly for maximum efficiency.

8.1 Architecture

A common method for load balancing in large scale web based systems is to use the following five tiers [27]. These tiers are illustrated in Figure 8.1.

8.1.1 Load Balancing

The first tier consists of hardware based load balancing that takes incoming HTTP requests and applies a load balancing algorithm to distribute them among the second tier.

8.1.2 Web Servers

The second tier consists of lightweight web servers. Each web server receives HTTP requests from the previous tier and determines what static content it can return, such as standard images for buttons or logos, and what content will need to be dynamically generated. It makes HTTP requests to the next tier, and returns them to the requester.

8.1.3 Load Balancing

The third tier consists of more hardware based load balancing that takes the HTTP requests for dynamic content and applies a load balancing algorithm to distribute

them among the fourth tier.

8.1.4 Application Servers

The fourth tier consists of fully featured application servers that are capable of producing dynamic content using information from the fifth tier.

8.1.5 Database

The fifth tier consists of a database cluster holding information to be accessed or modified by the application server tier. This information is not accessible by the web servers in the second tier.

8.2 Normal Operation

A client generates a request for a web page and sends it via HTTP to the published URL. The first tier of load balancing chooses a web server with sufficient free resources to handle the request. This web server determines what part of the request will require dynamic content and generates a request for this dynamic content. The third tier allocates this request to an application server with sufficient free resources to handle it. The application server generates the dynamic content, making use of information from the database cluster, and sends it to the web server. The web server responds to the client with the full content.

If a client is participating in a stateful session, such as using a shopping cart before making a purchase, then every request for that session will be directed to the same application server, to remove the need to propagate system state to all application servers.

8.3 Adding Proactive Fortification

If a malicious client wishes to alter system data then it will need to successfully compromise an application server. If they were merely able to compromise a web server then they would be able to generate malicious requests for dynamic content to one or more application servers. However, these requests would not be able to perform unauthorised actions at the application server level, such as generating purchases, and would only result in unrequested pages being generated, something which a malicious client could do simply by requesting those pages in the first place. A compromised web server could pose more of a problem from the point of view of a client as it could intercept client information and return malicious content. This would however be

partially mitigated by the fact that the client would get malicious content only when the load balancer randomly chose that web server to deliver their request to.

We note that using a compromised web server to intercept client information or return malicious content is essentially a man-in-the-middle attack, and HTTP requests are generally vulnerable to this [50]. The usual solution to prevent these attacks is to use a system such as HTTPS to encrypt communication. A compromised web server would compromise HTTPS, as the web server would need to hold the necessary key to decrypt communication from the client and encrypt communication to the client. This vulnerability can be seen to stem from the fact that the web server is not acting strictly as a proxy in this case, but is also providing some of the characteristics of a server. One way to remove this vulnerability for HTTPS requests would be to require that all communication over HTTPS was handled by the application servers. Thus, when HTTPS is required, such as for a financial transaction, the web server is behaving purely as a proxy. In this case the web server would not have the necessary certificate to handle HTTPS requests, so would be unable to impersonate the application server.

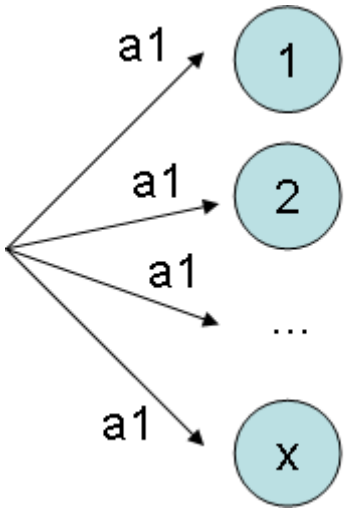
We also note that compromise of a web server would not breach system confidentiality. The confidentiality of information contained in individual client requests handled by that web server could be compromised, in the same way that man-in-the-middle attacks can compromise confidentiality of individual requests, but all of the system information held by the application servers would remain confidential. If all HTTPS communication was handled purely by the application servers, then all information sent by HTTPS would also remain confidential, even if it was initially directed to a compromised web server.

A malicious client will not be able to initially select which application server they wish to send requests to. However, if they can generate attacks as part of a stateful session, they will be able to send all of a series of malicious requests to the same server.

This suggests that proactive fortification should be performed at the application server level. This then gives us a choice of how to incorporate the proxies used in proactive fortification. One possibility is to simply replace every application server with a full proactive fortification system. The other possibility is to instead modify the web server tier to incorporate regular replacement and re-randomisation of servers. Then, we have the web server tier acting as a large proxy layer. This will also reduce the potential for an attacker to cause harm to clients through a compromised web server, as the time for which a web server stays compromised will be relatively short.

This large proxy layer could be a problem in a standard proactive fortification system, as every proxy would be visible to an attacker and they would only need to compromise one to then launch direct attacks against the primary node. This issue may however be mitigated by the fact that the malicious requests are first filtered through a load balancer, and hence the attacker is not able to choose how malicious requests are

Figure 8.2: Optimal Allocation of Requests to Web Servers



distributed to proxies. We consider how this will affect the attacker's optimal strategy, and show that, for any practically sized system using a load balancer and 3 or more web servers, we can expect a proactively fortified system using the web servers as a proxy layer to be at least as secure as our standard proactive fortification system model.

8.3.1 Consequences of Load Balancing on an Optimal Attack Strategy

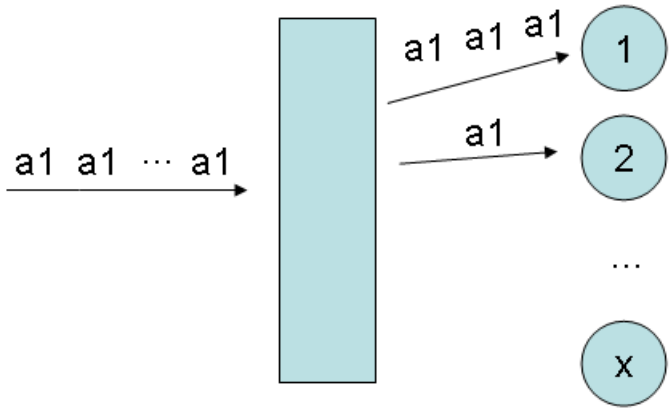
We have presented, in Chapters 4 and 5, attack models that assume the attacker can launch malicious requests at a constant rate against every publicly accessible node. This can be achieved in practice by the attacker constructing a malicious request containing the first possible randomisation key, and sending a copy of that request to each node, as shown in Figure 8.2. The attacker then follows the same procedure for all subsequent possible randomisation keys.

Here we illustrate the effect that having a load balancer distribute requests can have on this strategy.

We start by considering the simplified case where an attacker wishes to try one key against every web server. First of all we consider what will happen when a load balancer is not used and an incoming request is sent to all web servers. Here the attacker simply creates a malicious request using the key and sends a copy to each web server, as shown in Figure 8.2. Hence the expected number of requests required to try the key against all x web servers is x requests. Furthermore, in this case, this strategy will always result in all web servers receiving a malicious request, and there is no strategy available that uses less than x requests.

We then consider what will happen if a load balancer is present and the same strategy is used. Here, rather than the attacker being able to choose to send each of the x

Figure 8.3: One Possible Allocation of Requests to Web Servers with a Load Balancer



requests to a different web server, the attacker has instead to send all x requests to the load balancer and hope that each is allocated to a different web server. This may result in all x web servers receiving a malicious request, as happened in the case shown in Figure 8.2, but there is also the possibility of some servers receiving several requests, and other servers not receiving any requests, as shown in Figure 8.3.

We illustrate this difference by using a further simplification. We consider the case where there are two servers, and analyse the distribution of malicious requests with and without load balancing. When load balancing is not present, the attacker sends two malicious requests containing the key; one to each server. This guarantees that the key has been tried against both servers. As two requests is also the minimum number that can be used to try the key against both servers, we see that the expected number of requests needed to try the key against both servers is two.

When load balancing is present, calculating the expected number of requests needed to try the key against both servers is more complicated. Here we use the expected value formula $E(X) = \sum ip_i$ where p_i is the probability of needing exactly i requests before the key has been tried against both servers. This gives us an expected value of 2.67 to 2 decimal places, which is significantly higher then the expected value of 2 without load balancing. We also note that the probability of more than 2 requests being required to try the key against both servers is 0.5, the probability of more than 3 requests being required to try the key against both servers is 0.125 the probability of more than 4 requests being required to try the key against both servers is 0.03125, and to guarantee that key was tried against both servers would require an infinite number of requests. This is in sharp contrast to the case without a load balancer where it is always possible to try the key against both servers with 2 requests.

The probabilities of trying a key against both servers with i or less requests are shown in Table 8.1 for $1 \leq i \leq 10$.

Having seen an illustration of the effects of a load balancer on the optimal attack strategy, we then analyse how strong an effect the presence of a load balancer will

Table 8.1: Probability of Successfully Trying a Key Against 2 Servers with i or Less Requests

i	p_i
1	0
2	0.5
3	0.875
4	0.96875
5	0.992188
6	0.998047
7	0.998194
8	0.99823
9	0.998322
10	0.998327

have, and whether this is sufficient to out-weigh the advantage an attacker receives from having extra web servers to attack.

8.3.2 Analysis of the Effect of a Load Balancer on Malicious Request Distribution

Here we begin by deriving a formula for the probability that n keys are correctly allocated to x servers and show that this probability will be incredibly small for realistic key spaces. Next we prove a preliminary result; that if the presence of a load balancer decreases the effectiveness of an attacker's optimal strategy when n keys are tried against x servers, then the presence of a load balancer also decreases the effectiveness of an attacker's optimal strategy when k keys are tried against x servers, for any $k > n$.

Finally, we use Monte-Carlo simulations to calculate two probabilities for a range of x and n values; the probability that all n keys will be allocated to some subset of more than 3 of the x servers (and hence give the attacker an advantage relative to the system modelled in Chapter 5) and the probability that all n keys will be allocated to some subset of less than 3 of the x servers (and hence give the attacker a disadvantage relative to the system modelled in Chapter 5). This shows that, for a relatively small number of keys, the probability of the attacker being at a disadvantage is significantly larger than the probability of the attacker being at an advantage even for large numbers of servers. As we have proven that increasing the number of keys decreases the effectiveness of the attacker's strategy, we can see that this result holds for any realistic key space.

8.3.2.1 The Probability of Every Request Being Allocated Optimally for an Attacker

Here we show tha the probability of every request being allocated optimally for an attacker is negligible for any realistic key space.

We begin by taking the general case where there are x web servers, then, assuming totally random allocation of requests, we can consider how x specific requests could be allocated so that each server receives one of them.

The first request can be allocated to any server; this happens with probability 1 as the load balancer is guaranteed to give the request to one of the servers. The second request can be allocated to any server other than the one that the first request was allocated to; this happens with probability $\frac{x-1}{x}$ as there are x servers to choose from and the allocation is random. Similarly the third request is allocated to a server that has not yet had a request with probability $\frac{x-2}{x}$, and in general the yth request is allocated to a server that has not previously had a request with probability $\frac{x-y+1}{x}$. This gives us the general probability of

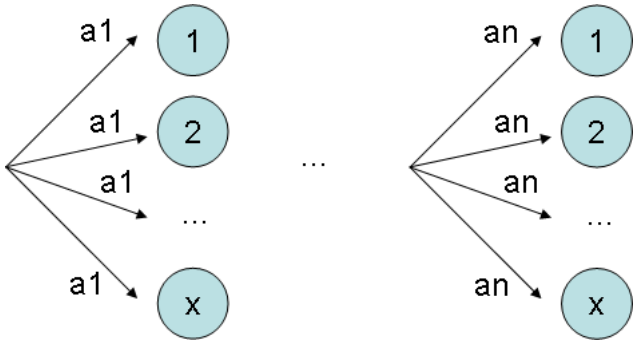
$\prod_{y=1}^{y=x} \frac{x-y+1}{x} = \prod_{y=1}^{y=x} \frac{y}{x}$

for x requests being allocated to x servers in such a way that each server receives one request. So, as x increases, the denominator of every term increases and the number of terms, all of which are less than 1, increase. This gives us a probability that decreases rapidly as x increases. For example when $x = 2$ the requests are correctly allocated with probability 0.5, when $x = 3$ the requests are correctly allocated with probability 0.22, when $x = 4$ the requests are correctly allocated with probability 0.09375 and when $x = 10$ the requests are correctly allocated with probability 0.00036288.

These probabilities are only for all of the requests with one key being allocated as the attacker would wish. However, a real attacker is going to try some large set of n keys, as illustrated in the case without a load balancer in Figure 8.4. The probability of this allocation happening for all the requests in an attack with n different keys when a load balancer is present is $p(x)^n$ where $p(x)$ is the probability of all of the requests for 1 key being allocated as the attacker would wish. This, combined with a realistically sized key space, shows us that the probability of an attacker being lucky enough to get all attacks allocated as they would wish is incredibly small. For example if there were 1000 possible keys, a key space much smaller than is seen in practice, and 4 servers, the probability of 4 requests per key being allocated as the attacker would wish them to be is 0.09375^{1000}.

This shows us that, for any realistic system, the chances of an attacker being able to distribute malicious requests to all web servers as if there was no load balancer is negligible.

Figure 8.4: Optimal Strategy for Allocation of Malicious Requests to Web Servers for a Set of n Keys



8.3.2.2 A Preliminary Result

Having shown that the chance of an attacker being able to optimally distribute malicious requests to all web servers in the presence of a load balancer is negligible, we now consider the possibility that the attacker cannot do as well if there was no load balancer, but can do better than if there were three proxies and no load balancer, the case that we have modelled in Chapter 5. We begin by proving the preliminary result that if the presence of a load balancer decreases the effectiveness of an attacker's optimal strategy when n keys are tried against x servers, then the presence of a load balancer also decreases the effectiveness of an attacker's optimal strategy when k keys are tried against x servers, for any $k > n$.

Assume that, for n keys and x servers, where n is a positive integer and $x > 3$, we have the probability of the attacker managing to successfully target more than 3 servers is a_1, the probability of the attacker managing to successfully target less than 3 servers is a_2 and the probability of the attacker managing to target exactly 3 servers is a_3. Now, we consider what happens if we increase n to $n + 1$. The probability of the attacker successfully targeting more than three servers for the key $n+1$ is some value b_1, the probability of the attacker successfully targeting less than three servers for the key $n + 1$ is some value b_2, and the probability of the attacker successfully targeting exactly 3 servers is some value b_3. We note that

$$0 < b_1, 0 < b_2, 0 < b_3$$

and

$$b_1 + b_2 + b_3 = 1$$

Now we consider the possibility of the attacker managing to successfully target more than 3 servers for all $n + 1$ keys.

This requires the attacker to successfully target more than three servers for the first

n keys and successfully target more than 3 servers for the last key. We note that this is necessary, but not sufficient, as there is also a requirement for the two parts to have more than 3 servers in common. So, the probability of the attacker managing to successfully target more than 3 servers for all $n+1$ keys $\leq a_1 b_1 < a_1$ as $0 < b_1 < 1$.

We then go on to consider the possibility of the attacker managing to successfully target less than 3 servers for all $n+1$ keys. This will happen when the attacker manages to target less than 3 servers for either the first n keys or the last key. We note that these conditions are sufficient but not necessary as it is also possible to have more servers targeted in the each of the two parts, but still have less than 3 servers in common.

So, the probability of the attacker managing to successfully target less than 3 servers for all $n+1$ keys $\geq (a_1 + a_3)b_2 + a_2 \geq a_2$ as $a_1, a_3, b_2 > 0$.

This means that, for any number of servers x and number of keys n that, if the attacker is worse off than in the case modelled in Section 5, it is also true that the attacker is worse off for x servers and k keys, where k is any integer greater than n.

8.3.2.3 Probability Simulations

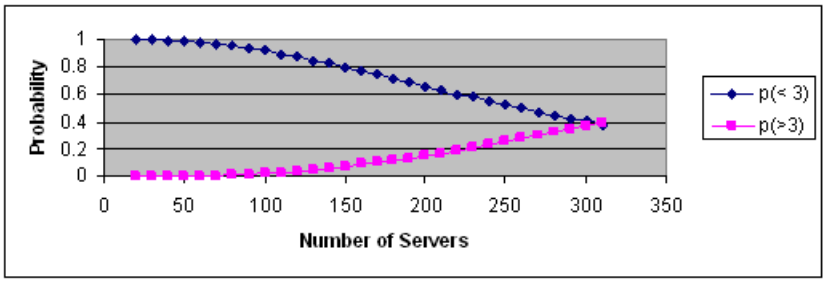
We now use probability situations to show that, for any practically sized system using a load balancer and 3 or more web servers, we can expect a proactively fortified system using the web servers as a proxy layer to be at least as secure as our standard proactive fortification system model.

We assume that there are x web servers, where x is some integer larger than 3 and there are n keys that the attacker wishes to use in attacks against them. We then need to calculate the probability of the attacker being able to try all n keys against 4 or more servers by launching nx attacks; i.e launching x attacks for each of n keys. This is the probability that the same group of 4 or more web servers receive a malicious request from each of the n groups of x malicious requests with the same key. Calculating this probability analytically is not feasible, so we instead use Monte-Carlo simulations to estimate the probability for a range of possible values.

In each case we allocate x requests to x servers with an equal random chance of each request going to each server. We repeat this n times and check to see how many servers received a request all n times. This is repeated a large number of times, and the probability of the attacker managing to target more than 3 web servers is taken to be the proportion of times that more than 3 servers received a request all n times. Similarly, the probability of the attacker managing to target less than 3 web servers is taken to be the proportion of times that less than 3 servers received a request all n times.

We consider the system to perform as well or better than our previous analysis with 3 servers if the probability of the attacker managing to target less than 3 servers is

Figure 8.5: Probabilities of More Than 3 and Less Than 3 Web Servers Being Successfully Targeted by a Malicious Client for 10 Keys



equal to or greater than the probability of the attacker managing to target more than 3 servers.

We first simulated cases where the number of keys $n = 10$. This is not intended to be representative of a real system. However, as we have proved that if a result is true for $n = a$ then it is true for any $n > a$, these results, which are relatively quick to calculate will allow us to identify a range of numbers of web servers for which the an attacker will not do better than in the three proxy system modelled in Chapters 4 and 5. The results are presented in Figure 8.5 and show us that, for 300 or less web servers, the web server model with load balancing is more difficult to attack than the three proxy system.

We then simulated cases where $n = 20$, with more than 20 web servers. This is again an unrealistically small number of keys, but will relatively quickly allow us to identify a range of numbers of web servers for which the an attacker will not do better than in the three proxy system. The results are presented in Figures 8.6 and 8.7 and show that the web server model is not easier to attack than the three proxy system when 3000 or less web servers are used.

We also note that these simulations show us a large range of numbers of servers for which the probability of less than 3 servers getting a request with each key is very large, and the probability of more than 3 servers getting a request with each key is negligible. The fact that n keys were considered means that this is the probability of less than 3 servers getting a request with every key, for any block of n keys. Hence we can reason that, for a system with 10 or less servers, less than three servers will get all of the malicious requests for each block of ten keys tried. Similarly, for a system with 460 or less servers, less than three servers will get all of the malicious requests for each block of twenty keys tried, and it is highly likely that less than three servers will get all of the malicious requests for most or all blocks of twenty keys tried for a system with 3000 or less servers.

Figure 8.6: Probability of Less Than 3 Web Servers Being Successfully Targeted by a Malicious Client for 20 Keys.

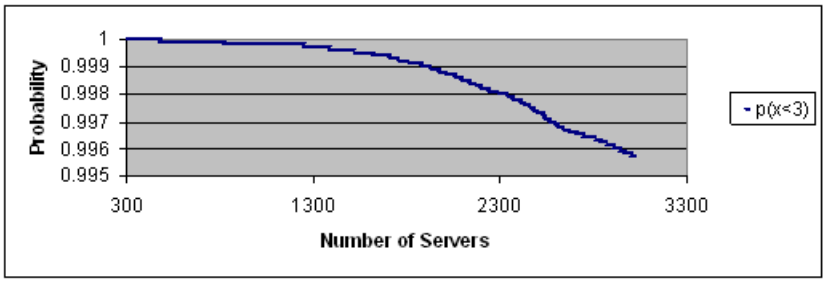
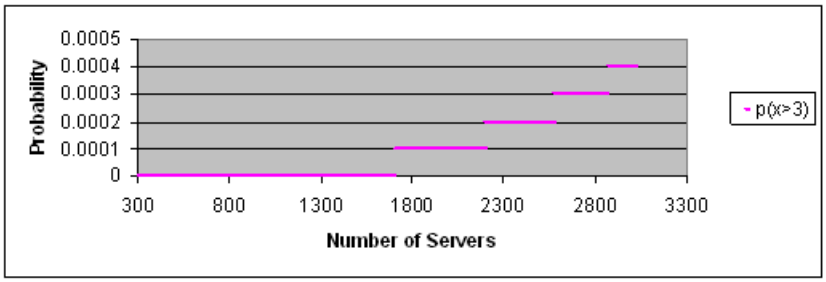


Figure 8.7: Probability of More Than 3 Web Servers Being Successfully Targeted by a Malicious Client for 20 Keys.



8.4 Implementing a Proactive Fortification System Using the Apache Tomcat Application Server

This implementation of a proactive fortification system uses the five tier architecture shown in Section 8.1 and the second proactive fortification method discussed in Section 8.3; treating the web server tier as the proxy tier, and using proactive fortification techniques to turn each application server in the application server tier into a server tier for the FORTRESS system.

Apache Tomcat has a clustering function that allows system state to be propagated between different instances of itself using TCP, and which periodically checks membership of the clustering group. This functionality will eliminate the need for the proactive fortification system to handle heartbeats and update messages itself. Instead, the proactive fortification system can simply initialise instances of Apache Tomcat in the same cluster.

The system architecture will also remove the need for us to explicitly introduce a name server. The initial load balancer will sit on a fixed IP address, giving the client the necessary information to send requests. This load balancer will know the addresses of the web servers and distribute requests accordingly. Each web server will only need to know the address of the second load balancer which will be fixed. The second load balancer will know the addresses of each cluster of application servers. This address will not change, with migration of servers only changing the membership of the cluster

group.

This allows us to simplify the implementation structure and use the following three components:

8.4.1 Fortress Timer

The Fortress Timer will be largely unchanged from the more general case. It will still send messages to initialise nodes, stop nodes, initialise proxies, stop proxies, inform proxies of the identities of the current primary and backups, and inform the name server of the addresses of the current proxies. However, it will not be necessary to send messages to inform standbys to become the primary or backup, as this will automatically happen when the previous primary and backups are stopped and hence leave the cluster.

8.4.2 Legacy Code Wrapper

The clustering functionality of Apache Tomcat allows us to use a relatively simple legacy code wrapper. This is simply a service that will run on each node when it is initialised, and wait for a start message from the Fortress Timer. When this start message is received it will start an instance of Apache Tomcat with the appropriate clustering information for it to join the current cluster. Apache Tomcat will then handle the sending of heartbeat messages and propagation of system state itself. When the Legacy Code Wrapper receives a stop message from the Fortress Timer it will force Apache Tomcat to stop, causing it to leave the cluster.

8.4.3 Proxy

Each proxy will run an Apache HTTP Server, which can be started and stopped by a service similar to the Legacy Code Wrapper, or rebooted and refreshed by a hardware based reboot server. The web server will contain a simple application that receives requests, forwards them to the current primary and backups, and forwards the replies to the clients.

8.5 Apache Tomcat Implementation: Performance Overhead Evaluation

8.5.1 Evaluation Set-up

Apache Tomcat servers were set up on 12 machines. Each of these machines also had a legacy code wrapper installed. An Apache HTTP Server was set up on another

machine. This machine also had a proxy service installed. A Fortress Timer was set up on another machine.

The Fortress Timer was configured to send messages to the Legacy Code Wrappers to start and cluster the first 6 Apache Tomcat servers, and to send a message to the proxy service to start the Apache HTTP Server with a fail-over scheme that will default to the first Apache Tomcat server, and fail-over to the next two in order.

After the migration interval expired, the Fortress Timer was configured to send a message to the proxy service to change the fail-over scheme so that it will default to the fourth Apache Tomcat server, and fail-over to the next two in order. The Legacy Code Wrappers on the first, second and third Apache Tomcat servers were sent messages to stop, and the Legacy Code Wrappers on the seventh, eighth and ninth Apache Tomcat servers were sent messages to start and join the cluster. This Fortress Timer was configured to continue this pattern at the end of each migration interval.

8.5.2 Measurement Strategy

Apache Tomcat server performance was measured for two web pages. The first of these was a simple JSP page containing text and an associated session. The second was a product search page from an online shopping cart, requiring database access to generate dynamic content, including a number of pictures. Apache JMeter was used for all tests other than session handling. JMeter was set to generate 20000 requests from each of 100 clients running simultaneously for the simple JSP page. JMeter was set to generate 20000 requests from each of 50 clients running simultaneously for the product search page.

The use of Apache Tomcat clustering results in state transfer being a special form of the progressive state transfer mechanism that was presented in Section 6.1.2.2. Every time the primary processes a request that results in a state change, this state change is propagated to the standby nodes as well as the backups. This results in the concept of an update interval being irrelevant to this system.

Heartbeat interval and migration interval are both under the control of the system administrator, so testing included the variation of each of these variables while holding the other constant.

8.6 Overhead Measurement for a Simple Web Page with Sessions

Here we accessed a simple JSP page containing static text and displaying the value of a session variable. The session variable was set at the beginning of the experiment,

resulting in each request for the page requiring dynamic page generation, but no database access or updating of application server state. Latency and throughput testing was performed using Apache JMeter to generate 20000 requests from each of 100 clients running simultaneously. This number of requests was chosen based on preliminary testing showing that it would be sufficient to cover a time interval including at least one migration for all parameters.

8.6.1 Session Handling

The ability of the system to maintain session data as it migrates was tested by setting a session variable and then reading this session variable after migration for each of 20 migrations. This test was performed a total of 10 times, and in every case the session variable held the same value at all times. We note that this session handling test is analogous to testing the correctness of responses and state, as described in Section 7.6.

8.6.2 Heartbeat Interval

The migration interval was fixed at 100 seconds for the proactively fortified system and the heartbeat interval was varied between 1 second and 6 seconds in 1 second steps.

The increase in latency caused by proactive fortification, calculated as a percentage of the latency of the non-fortified system is shown in Figure 8.8. The decrease in throughput caused by proactive fortification, calculated as a percentage of the throughput of the non-fortified system is shown in Figure 8.9. The absolute values, confidence intervals for these values, and an analysis of how overhead varies with heartbeat interval are presented in Appendix G.

The increases in latency appear, at first glance, to have the potential to be problematic. They range from 11.1% to 42.9%, with an increase of 28.57% when the heartbeat interval is set to 6000ms, which is the point at which both the proactively fortified and primary-backup systems get the lowest latency figures. However, the latency figures are so small that these large increases may not make much difference in practice. A user accessing a page is not likely to be inconvenienced by, or even notice a difference of 2ms, as was found when the the heartbeat interval is set to 6000ms.

However, comparing the mean latencies does not tell us what causes this difference of 2ms. If, at one extreme, every response time measured increased by 2ms, this would not affect the user. On the other hand, if most response times stayed the same, but there was an occasional latency of 10000ms, then some user would be inconvenienced. This suggests that a more relevant measure of performance with regard to latency in

Figure 8.8: Increase in Latency as Heartbeat Interval Varies

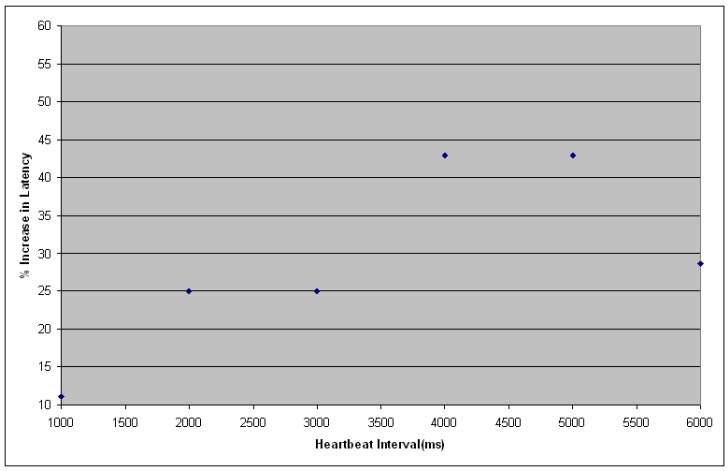
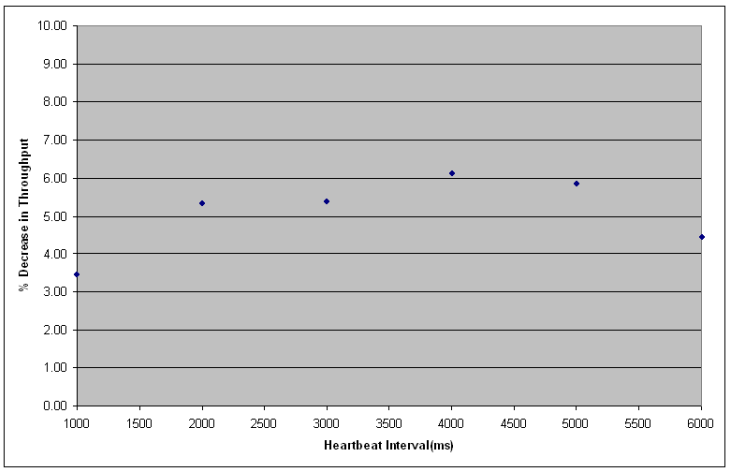


Figure 8.9: Decrease in Throughput as Heartbeat Interval Varies



this case would be to consider the maximum latency encountered in our tests. These maximum latencies are presented in Table 8.2.

Here we see that the maximum latencies are higher for the proactively fortified system. However, none of these latencies are sufficiently large to cause a noticeable delay for a user. This leads us to conclude that, for this range of values, throughput is a more relevant measure of relative system performance from the point of view of a system user.

The decrease in throughput ranges from 3.47% to 6.11% with a decrease of 4.45% when the heartbeat interval is set to 6000ms, which is the point at which both the proactively fortified and primary-backup systems get the highest throughput figures. This shows that the amount of requests that can be processed in a given period of time is only slightly reduced by the addition of proactive fortification for the range of heartbeat values considered.

Table 8.2: Maximum Latencies as Heartbeat Interval Varies

Heartbeat Interval	Proactively Fortified System	Primary-Backup System
1000ms	285ms	184ms
2000ms	276ms	182ms
3000ms	283ms	175ms
4000ms	274ms	142ms
5000ms	269ms	190ms
6000ms	281ms	183ms

8.6.3 Migration Interval

The migration intervals were found to have a strong relation with performance overhead, with efficiency increasing as the migration interval increased.

The heartbeat interval was fixed at 6 seconds for the proactively fortified system and the migration interval was varied between 20 seconds and 100 seconds in 10 second steps.

The decrease in throughput caused by proactive fortification, calculated as a percentage of the throughput of the non-fortified system is shown in Figure 8.10. The increase in latency caused by proactive fortification, calculated as a percentage of the latency of the non-fortified system is shown in Figure 8.11. The absolute values, confidence intervals for these absolute values and an analysis of how overhead varies with migration interval are presented in Appendix G.

The increase in mean latency ranges from 28.5% to 771.4%, showing an incredibly large relative increase in latency as the migration interval becomes very small. However, when we examine the actual figures involved we notice that even the 771.4% increase in mean latency only results in a increase of 54ms, a difference in response time that is unlikely to be inconvenient, or even noticeable for the user.

This difference in mean latency does not tell us necessarily that request times are being uniformly increased by the Figure of 54ms. It is equally possible that many requests take the same amount of time as in the primary-backup system, while some requests take a lot longer. This suggests that we may get a fuller picture of whether this change in mean latency will be relevant to the user, by considering the maximum latencies measured. These values are presented in Table 8.3. As the highest maximum latency value measured was 283ms, we can conclude that the presence of proactive fortification is not likely to make a noticeable difference to the user with regard to response time in any of the cases considered here, and hence throughput is likely to be a more meaningful measure of performance overhead.

The decrease in throughput ranges from 4.45% to 75.79%. This suggests that proactive fortification causes only a relatively small drop in throughput for larger migration intervals, but can cause a significant decrease when migration intervals as small as 20

Figure 8.10: Increase in Latency as Migration Interval Varies

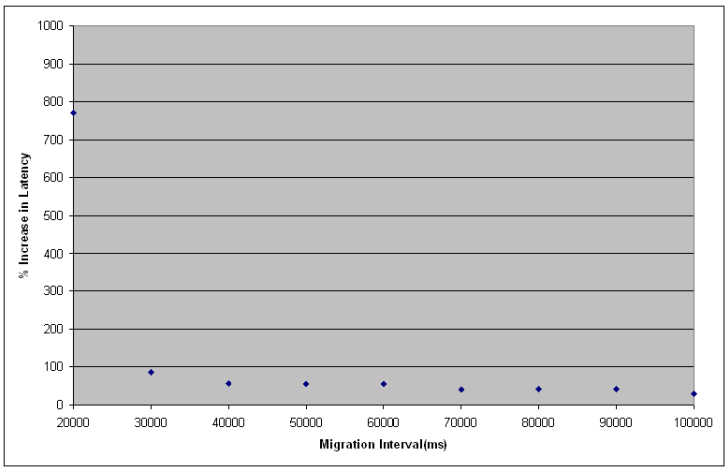
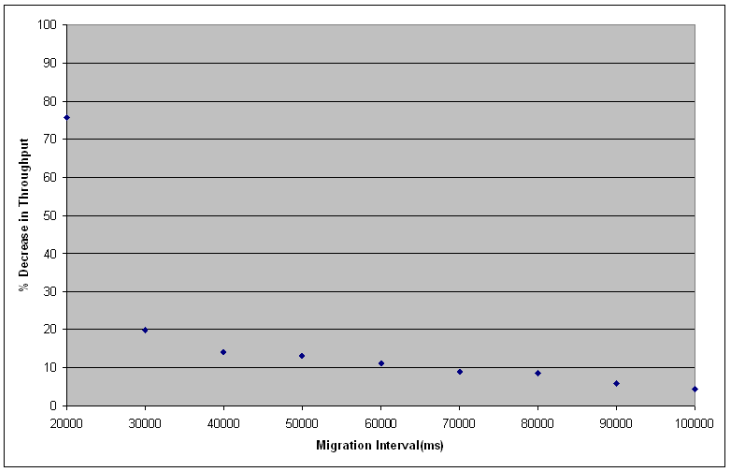


Figure 8.11: Decrease in Throughput as Migration Interval Varies



seconds are required.

8.7 Testing an Online Shopping Search Page

Here, searches were performed using a small set of search terms. Each search resulted in the application server that handled the request performing a database search and then producing a dynamic page containing the appropriate graphics for the items returned by the search. Latency and throughput testing was performed using Apache JMeter to generate 20000 requests from each of 50 clients running simultaneously. This number of requests was chosen based on preliminary testing showing that it would be sufficient to cover a time interval included at least one migration for all parameters. Each client was given a different search term.

Table 8.3: Maximum Latencies as Migration Interval Varies

Migration Interval	Maximum Latency
20s	279ms
30s	265ms
40s	283ms
50s	265ms
60s	276ms
70s	271ms
80s	279ms
90s	280ms
100s	281ms

8.7.1 Session Handling

The ability of the system to maintain session data as it migrates was tested by logging in to the shopping site and then checking that the user was still logged in after migration for each of 20 migrations. This test was performed a total of 10 times, and in every case the user stayed logged in.

8.7.2 Heartbeat Interval

The migration interval was fixed at 100 seconds for the proactively fortified system and the heartbeat interval was varied between 1 second and 6 seconds in 1 second steps.

The increase in latency caused by proactive fortification, calculated as a percentage of the latency of the non-fortified system is shown in Figure 8.12. The decrease in throughput caused by proactive fortification, calculated as a percentage of the throughput of the non-fortified system is shown in Figure 8.13. The absolute values, confidence intervals for these absolute values and an analysis of how overhead varies with heartbeat interval are presented in Appendix G.

The increase in mean latency ranges from 16.06% to 19.6%. This is a considerable increase, but much smaller than the increases seen in Section 8.6. However, as in Section 8.6, we need to consider how this increase in mean latency occurred. All of the mean latencies measured are in the range 450ms - 600ms, and an increase of 150ms in response time would not inconvenience a user, so we need to examine the maximum latencies recorded to see if some response times were sufficiently long to cause a possible issue. The maximum latencies are presented in Table 8.4.

This shows us that, in the cases with the highest latencies, a user will have to wait over 2 seconds for a response. This will certainly be noticeable, but may not be unacceptable if it happens occasionally, or in circumstances where the user is expecting to see a lot of data. We also note that the primary-backup system has maximum

Figure 8.12: Increase in Latency Caused by Proactive Fortification as Heartbeat Interval Varies - 100s Migration Interval

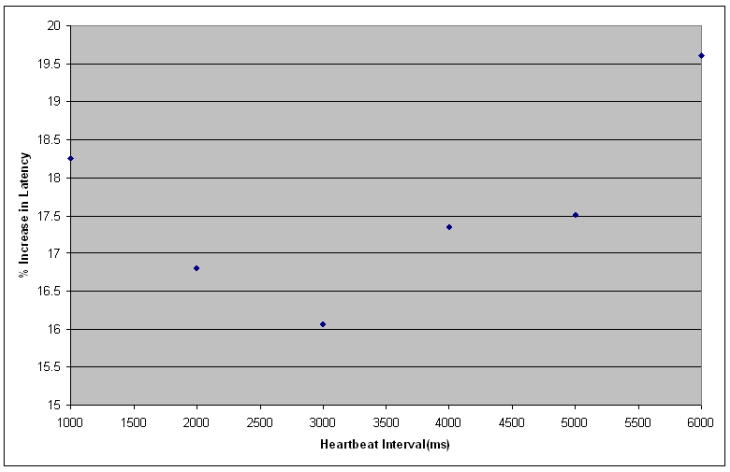
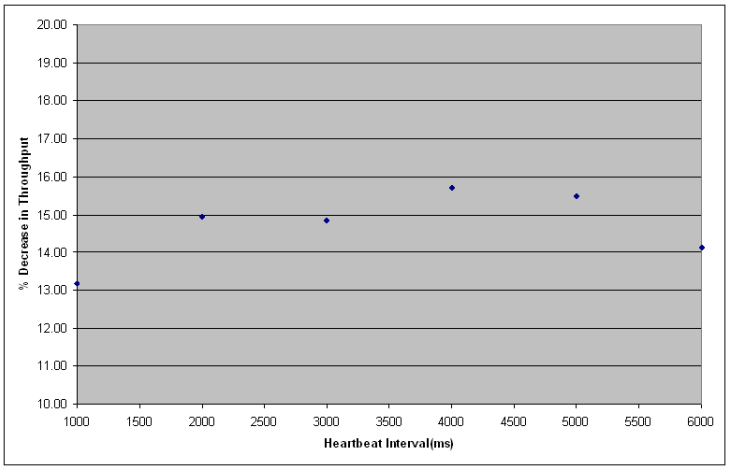


Figure 8.13: Decrease in Throughput as Heartbeat Interval Varies



latencies of over 1 second. This suggests that the combination of operations performed by the online shopping page and the database and network used may have resulted in a system which is slower than would normally be expected in large scale web applications.

The decrease in throughput ranges from 13.16% to 15.70% with a value of 14.13% for a heartbeat interval of 6000ms, the heartbeat interval for which both systems provide the highest throughput over the values considered.

8.7.3 Migration Interval

The migration intervals were found to have a strong correlation with performance overhead, with efficiency increasing as the migration interval increased.

The heartbeat interval was fixed at 6 seconds and the migration interval was varied between 20 seconds and 100 seconds in 10 second steps.

Table 8.4: Maximum Latencies as Heartbeat Interval Varies

Heartbeat Interval	Proactively Fortified System	Primary-Backup System
1000ms	2309ms	1012ms
2000ms	2286ms	1043ms
3000ms	2283ms	1028ms
4000ms	2295ms	1019ms
5000ms	2312ms	1047ms
6000ms	2247ms	1032ms

Table 8.5: Maximum Latencies as Migration Interval Varies

Migration Interval	Maximum Latency
20s	2261ms
30s	2257ms
40s	2293ms
50s	2303ms
60s	2287ms
70s	2291ms
80s	2259ms
90s	2264ms
100s	2247ms

The decrease in throughput caused by proactive fortification, calculated as a percentage of the throughput of the non-fortified system is shown in Figure 8.12. The increase in latency caused by proactive fortification, calculated as a percentage of the latency of the non-fortified system is shown in Figure 8.13. The absolute values, confidence intervals for these absolute values and an analysis of how overhead varies with migration interval are presented in Appendix G.

The increase in mean latency ranges from 18.25% to 244.04%. This not only gives a very large increase in latency when a short migration interval is required, but it also results in a mean latency that will give a noticeable delay to a user in seeing page results; 1734ms. The maximum latencies measured are shown in Table 8.5. The maximum latencies measured do not vary significantly as the migration interval is decreased. This, coupled with the increase in the mean latency suggests that the reduction in migration interval is not increasing the maximum time taken to give a response, and is instead increasing the number of responses that fall close to this maximum time.

The decrease in throughput ranges from 14.13% when the migration interval is 100 seconds to 82.30% when the migration interval is 20 seconds. This is a significant but manageable amount when the migration interval is 100 seconds, increasing to a very severe overhead when the migration interval is 20 seconds.

Figure 8.14: Increase in Latency as Migration Interval Varies

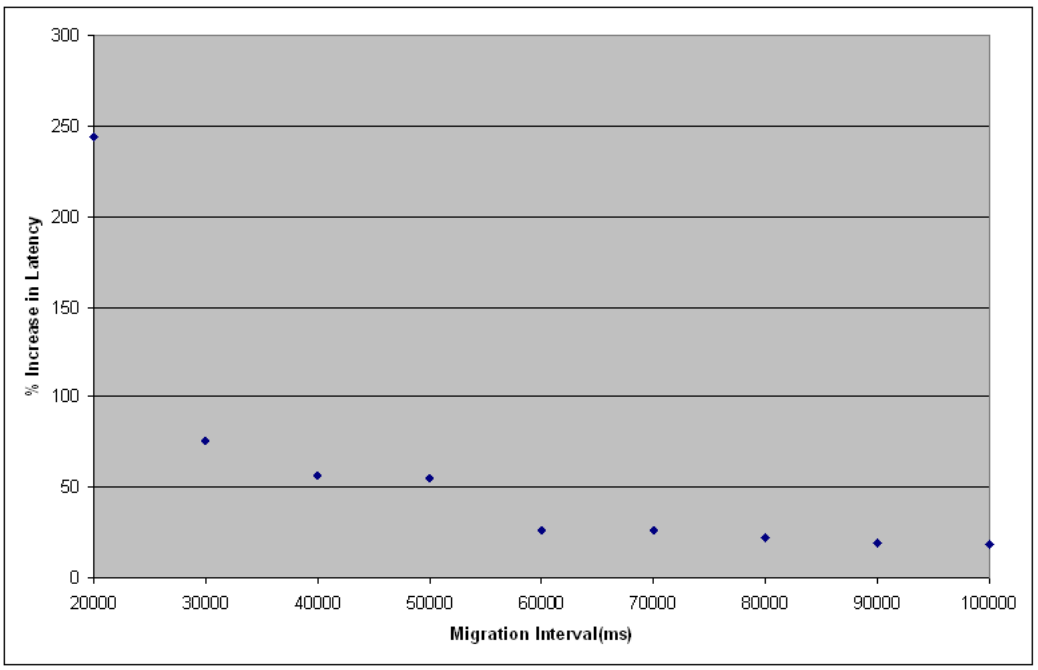
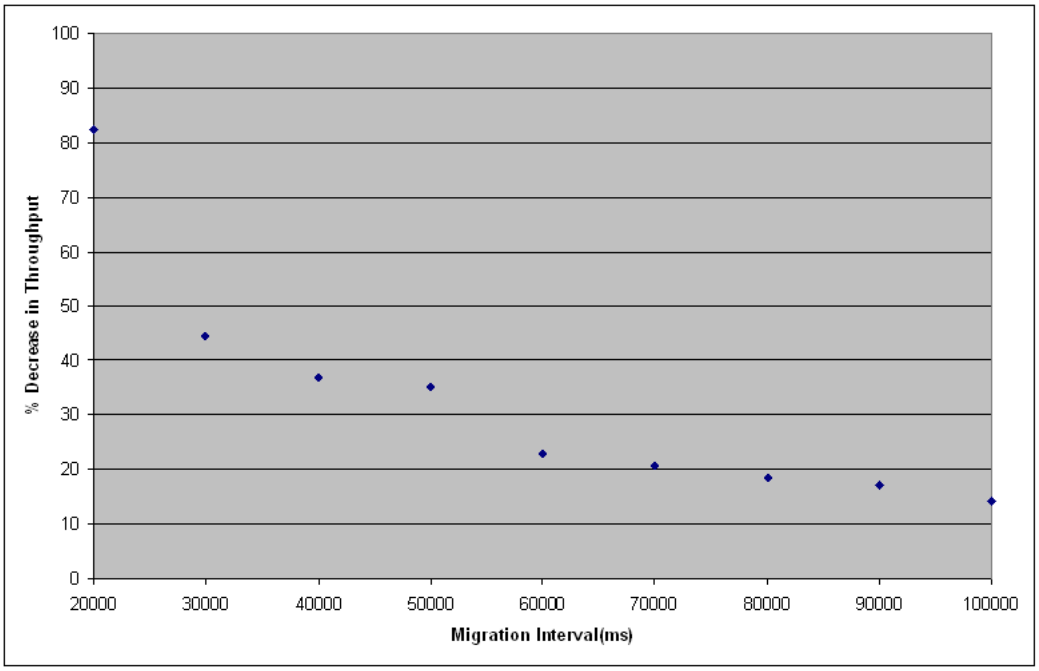


Figure 8.15: Decrease in Throughput as Migration Interval Varies



8.8 Summary

The preceding data shows us two key results. Firstly, proactive fortification of an online shopping system does not have to cause a large performance overhead. The increase in mean latency and decrease in throughput caused by proactive fortification can be kept reasonably small except when very small migration periods are required. Even when very small migration periods are required, mean latency is noticeably increased, but not unacceptably high, and throughput is still large enough to make the use of extra application servers to restore the needed throughput a possibility.

When migration periods are small, mean latency is increased by the fact that a larger percentage of time is spent in transferring state to new nodes that have just joined the cluster, resulting in more requests that are delayed by this state transfer. One possibility for reducing the mean latency in a production system is to offset the migration periods of several different proactively fortified application servers, and use the load balancer to allocate the majority of requests to nodes that are not currently involved in migration. As each application server only maintains state for sessions that were started with that server, or members of its cluster, the load balancer would not be able to redirect later requests in a session to new application servers. However, the load balancer would be able to stop any new sessions being started with the application server during system migration, thereby reducing the overall load at this point.

We also note that very small migration periods are only needed when an attacker must be presented with very small windows of time in which to attempt to compromise application servers. Hence very small migration periods are only needed when presented with an attacker that has a relatively strong ability to compromise application servers when not confronted by an intrusion resilient system. These are just the circumstances when it may be worth the cost of providing additional servers to cope with the overhead of regularly migrating servers.

Chapter 9

Summary and Conclusions

This thesis introduced proactive fortification, a new architectural framework for intrusion resilience in distributed systems, leveraging proactive fortification and the use of cheap off-the-shelf hardware. The intrusion resilience of this protocol was analysed over a variety of assumptions, and compared to intrusion tolerant active replication systems and crash tolerant primary-backup systems. A family of state transfer protocols were introduced for server migration, and the possible effects of these protocols on intrusion resilience were examined.

The framework was shown to augment legacy systems with proactive fortification, and the overhead of an evaluation system was measured. Finally, a lightweight implementation of proactive fortification in a large scale web application context was presented. Overhead was again measured, and in both cases promising results were obtained.

This section will summarise the work that has been presented, discuss the implications of this work and then go on to look at possible directions for future work.

9.1 Summary

We began by examining related work in intrusion tolerance, and considering the types of attacks that are likely to be used against publicly accessible distributed systems. This was coupled with considering the use of proxies in defending distributed systems against code injection attacks, and the relative costs of additional hardware in intrusion resilient systems. This was followed by a definition of the FORTRESS system in Chapter 3 including the supporting mechanisms needed to allow such a system to function, and an analysis of the possibility of an attacker targeting these mechanisms.

A generic model was produced for assessing the intrusion resilience of distributed systems in Chapter 4, using modelling assumptions based on the examination of likely attacks in Section 2. This model was used in Chapter 5 to calculate the expected lifetime until system compromise of three key systems, the FORTRESS system, the SMR

system, and the PB system. These systems were analysed with proactive obfuscation, the case where each node is replaced with a differently randomised node at the end of each unit time-step, start-up only obfuscation, the case where each node is differently randomised at system start-up and is not re-booted after this and proactive recovery, the case where each node is re-booted but not re-randomised at the end of each unit time-step.

Systems with proactive obfuscation were considered with three classes of diversity. Infinite diversity is the class where a large number of diverse executables are available and the attacker is not able to track whether a given executable has been encountered before. Hence the attacker is effectively left in the situation that, upon encountering an executable they have no prior knowledge of it.

Finite diversity is divided into two classes. Relatively large finite diversity results from proactive obfuscation generating a large array of executables, with some characteristic that allows attackers to identify an executable that they have previously been able to intrude into. This means that, should a previously intruded executable be chosen for a new node, the attacker will be able to compromise it immediately.

Relatively small finite diversity results from an inability of the system to successfully refresh a compromised replica to the point that all malicious code is removed from it. Then, we have a situation where, regardless of the diverse executable chosen for a new node, if it has been compromised before then it is still compromised. Hence, as far as choosing new compromised nodes is concerned, the level of diversity is equal to the number of physical machines that are being used for node replacement. In this case, a large degree of randomisation is still advantageous for the reduction in the probability of a vulnerability being initially compromised that it provides.

This analysis of expected system lifetimes until compromise showed that proactive obfuscation causes a major increase in intrusion resilience in all systems, and that proactive fortification is a viable method for intrusion resilience except when very small amounts of diversity are available. There is a large class of situations in which FORTRESS actually out-performs proactive SMR for publicly accessible distributed systems. There are also other classes of situations where SMR out-performs FORTRESS, but FORTRESS still performs sufficiently well to be considered as an alternative if the removal of non-determinism is overly restrictive.

We then went on to consider state transfer protocols for FORTRESS systems and designed a family of protocols, as shown in Chapter 6. An analysis of the possible effects of these protocols on intrusion resilience is presented in Appendix E and shows that they do not open up additional avenues of attack. Chapter 6 then presented the design of the software components needed for state transfer within a proactive fortification framework, at a conceptual level of abstraction. The necessary processes were identified as well as the interactions between them. This conceptual design was

refined to produce a concrete architecture, identifying the necessary objects for the framework to be implemented using EJB technology running on JBoss servers.

Chapter 7 outlined an overhead evaluation strategy, alongside a concrete design of a set of components for a simple legacy code system allowing jobs to be booked and job details to be checked. The results of this overhead evaluation were then presented, demonstrating that the system has a reasonably small performance overhead except when very small migration intervals are required.

Chapter 8 considered the possibility of augmenting the intrusion resilience of a large scale web application with proactive fortification. First, a common structure for large scale web applications was presented and the effects of this structure on our previous expected lifetime analyses was considered. This lead us to conclude that proactively fortifying individual application servers, using the web tier as proxies that are periodically replaced would result in a system at least as intrusion resilient as the proactive fortification system models considered in Chapter 5.

A lightweight proactive fortification system was designed that made use of the clustering features of the Apache Tomcat application server, and this was used to perform overhead measurement experiments on two types of user interaction. The first of these interactions involved retrieving a web page combining static content with some dynamic session information. The second of these interactions used session information to query a database and return a dynamic page made up of several complex elements. This results of these overhead measurement experiments were presented, showing that the performance overheads caused by proactive fortification are reasonably small, except when very short migration intervals are required.

9.2 Conclusions

We have successfully developed an intrusion-resilience scheme that does not require the underlying system to be represented as a deterministic state machine. The intrusion-resilience of this scheme has been evaluated under a range of likely conditions for publicly accessible distributed systems. This has shown that the scheme provides a significant increase in intrusion-resilience over an unaugmented system for a wide range of situations, and even provides a larger increase in intrusion-resilience than active replication in some cases.

We also note that this intrusion-resilience scheme maintains both confidentiality and integrity of data until the entire system is compromised. This is in contrast to SMR schemes in which intrusion into one replica results in a breach of confidentiality while still maintaining integrity.

This evaluation of intrusion resilience has been based on an attack model that makes assumptions about how an attack is carried out and if these assumptions fail to

hold then the evaluations may be invalidated. One of the most basic assumptions is that attacks have some (generally small) probability of succeeding, rather than succeeding instantly. This assumption is supported by the nature of real world buffer overflow attacks against systems using obfuscation schemes, but may not hold for every type of attack. One specific type of attack for which it may not hold is the exploitation of an error at the system design stage. If a badly designed system allows an unusual combination of user actions to change system state in an undesirable way, then exploitation of this vulnerability is likely to be deterministic for all replicas, regardless of obfuscation. We note however that a vulnerability of this type would affect all of the intrusion resilient systems that we have compared equally, and that defending against this type of vulnerability may require orthogonal techniques such as audit systems that identify invalid system states.

Another key assumption made is that attacks against the system follow an attack model where an attacker uses malicious clients to attempt to compromise the intrusion resilience scheme. Hence the intrusion resilience evaluations do not consider situations where an attacker may use other attack vectors. Attacks that first attempt to trick a system administrator into changing the system configuration to a less resilient one (e.g. by using relatively weak denial of service attacks to increase the number of requests that need to be processed) before launching malicious attacks fall outside the scope of our evaluation. Similarly, attacks that attempt to intrude the personal computer of a system administrator and steal the administrator's authentication credentials for the system to be intruded are not covered by our attack model. Both of these types of attacks have been seen in the real-world, illustrating that our evaluations are not applicable to all possible attacks.

A further assumption that may not always hold is that all servers and proxies are equally difficult to compromise. This is reasonable when considering our motivating attacks where the attacker needs only to determine the obfuscation key used to obfuscate each system, but may not hold if different nodes have different vulnerabilities (e.g. the FORTRESS system may have different vulnerabilities in the proxy and server nodes).

Measuring performance impact has shown that it is possible to build proactive fortification systems with primary-backup in the server tier that do not result in a large increase in latency or decrease in throughput, except when very small migration intervals are required. This applies both to standard distributed applications and large scale web applications.

9.3 Future Work

The possibility of using proactive fortification alongside active replication has been mentioned, and state transfer mechanisms have been discussed. However, there is still

a need to examine how state transfer mechanisms might affect intrusion resilience when active replication is used in the server tier. This information could be used alongside the expected lifetime evaluation techniques we have used here to determine how much proactive fortification can increase the intrusion resilience of an active replication system, and how much the introduction of active replication and proactive fortification can increase the intrusion resilience of an unreplicated system.

The expected lifetime figures shown in this thesis for proactively fortified primary-backup systems differ significantly depending on whether indirect attacks are possible, and if they are then how likely they are to succeed relative to direct attacks. Some discussion of the factors that may affect the possibility of indirect attacks have been presented, but it would be worthwhile to analyse real world attacks and quantify the factors that affect indirect attacks and, if possible, how much they are likely to reduce the speed at which indirect attacks can be launched.

Probabilities of attacks succeeding against individual nodes have been varied over a wide range, to make the system lifetime comparisons as generally applicable as possible. A number of factors that may affect these probabilities have been identified such as the number of possible randomisation keys and the rate at which requests can be processed. However, further analysis of real world attacks may be able to identify smaller ranges of probabilities that are likely to occur for different vulnerabilities. This may make it possible to predict how intrusion resilient a system may be.

Such a real world analysis would also be useful in determining how the time interval between replacement of nodes relates to intrusion resilience. Intuitively we can see that reducing the time interval reducing the intrusion probability as it will give an attacker less time to attempt to determine the obfuscation keys of servers before they are replaced, but we do not have a model to relate specific time intervals to specific intrusion probabilities. As we have already presented analytical techniques to calculate expected lifetimes until system compromise from intrusion probabilities and experimental techniques to calculate performance overhead as time intervals between node replacement vary, such a model would allow expected lifetimes until system compromise to be directly related to performance overhead.

The performance overhead of FORTRESS systems has also been found to be dependent on how the time intervals between the sending of update and heartbeat messages are configured, parameters that do not affect intrusion resilience. The relationship between update intervals and performance overhead is complex and it would be beneficial to develop a model that enables the optimum update interval to be determined without having to use trial and error.

Bibliography

[1] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A multifaceted approach to understanding the botnet phenomenon. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 41–52, New York, NY, USA, 2006. ACM.

[2] Niv Ahituv, Yeheskel Lapid, and Seev Neumann. Processing encrypted data. *Commun. ACM*, 30:777–780, September 1987.

[3] Ross J. Anderson. *Security engineering - a guide to building dependable distributed systems (2. ed.)*. Wiley, 2008.

[4] Anonymous. Once upon a free(). *Phrack Magazine*, 57(9), August 2001.

[5] R. Baldoni, J.-M. Helary, M. Raynal, and L. Tanguy. Consensus in byzantine asynchronous systems. In *JOURNAL OF DISCRETE ALGORITHMS*, pages 1–16, 2000.

[6] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, PODC '83, pages 27–30, New York, NY, USA, 1983. ACM.

[7] Michael Ben-Or. Fast asynchronous byzantine agreement (extended abstract). In *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, PODC '85, pages 149–151, New York, NY, USA, 1985. ACM.

[8] G. R. Blakley. Safeguarding cryptographic keys. *Managing Requirements Knowledge, International Workshop on*, 0:313, 1979.

[9] Stephen W. Boyd, Gaurav S. Kc, Michael E. Locasto, Angelos D. Keromytis, and Vassilis Prevelakis. On the general applicability of instruction-set randomization. *IEEE Trans. Dependable Secur. Comput.*, 7:255–270, July 2010.

[10] bulba and ki13r. Bypassing stackguard and stackshield. *Phrack*, 11(56), May 2000.

[11] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. In *in Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC*, pages 123–132, 2000.

[12] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 42–51, New York, NY, USA, 1993. ACM.

[13] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[14] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[15] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43:225–267, March 1996.

[16] Dylan Clarke and Paul Ezhilchelvan. Assessing the attack resilience capabilities of a fortified primary backup system. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*. IEEE, 2010.

[17] Jean claude Laprie and Brian Randell. Fundamental concepts of computer systems dependability. In *Proc. of the Workshop on Robot Dep. , Seoul, Korea*, pages 21–22, 2001.

[18] Solar Designer. "return-to-libc" attack. Bugtraq, August 1997.

[19] Assia Doudou, Benoï¿œt Garbinato, and Rachid Guerraoui. Encapsulating failure detection: from crash to byzantine failures. In *Proc. International Conference on Reliable Software Technologies*, pages 24–50. Springer-Verlag, 2002.

[20] Assia Doudou and Andrï¿œ Schiper. Muteness detectors for consensus with byzantine processes. In *in Proceedings of the 17th ACM Symposium on Principle of Distributed Computing*, New York, NY, USA, 1997. ACM.

[21] Paul Ezhilchelvan, Dylan Clarke, Isi Mitrani, and Santosh Shrivastava. Proactive Fortification of Fault-Tolerant Services. In *Proceedings of the 13th International Conference On Principle Of DIstributed Systems*. Springer Science+Business Media, 2009.

[22] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[23] Felix C. Freiling, Thorsten Holz, and Georg Wicherski. Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. In *ESORICS*, pages 319–335, 2005.

[24] Roy Friedman, Achour Mostï¿œfaoui, and Michel Raynal. Simple and efficient oracle-based consensus protocols for asynchronous byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2:46–56, 2005.

[25] Charles Grinstead and Laurie Snell. *Introduction to Probability*. American Mathematical Society, 1997.

[26] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, March 2006.

[27] Perrin Hawkins and Bill Hilf. Building a large-scale e-commerce site with apache and mod_perl. http://perl.apache.org/docs/tutorials/apps/scale_etoys/etoys.html.

[28] Maurice P. Herlihy and J. D. Tygar. How to make replicated data secure. In *Advances in Cryptology - CRYPTO*, pages 379–391. Springer-Verlag, 1988.

[29] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.

[30] Kim Potter Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The securering protocols for securing group communication. *Hawaii International Conference on System Sciences*, 3:317, 1998.

[31] Kim Potter Kihlstrom, Louise E. Moser, and P. M . Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46:2003, 2003.

[32] Achmad I. Kistijantoro, Graham Morgan, Santosh K. Shrivastava, and Mark C. Little. Enhancing an application server to support available components. *IEEE Trans. Softw. Eng.*, 34:531–545, July 2008.

[33] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

[34] Bharat B. Madan, Katerina Goševa-Popstojanova, Kalyanaraman Vaidyanathan, and Kishor S. Trivedi. A method for modeling and quantifying the security attributes of intrusion tolerant systems. *Perform. Eval.*, 56(1-4):167–186, 2004.

[35] Dahlia Malkhi and Michael Reiter. Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th IEEE workshop on Computer Security Foundations*, CSFW '97, pages 116–, Washington, DC, USA, 1997. IEEE Computer Society.

[36] Francis P. Mathur and Algirdas Avižienis. Reliability analysis and architecture of a hybrid-redundant digital system: generalized triple modular redundancy with self-repair. In *Proceedings of the May 5-7, 1970, spring joint computer conference*, AFIPS '70 (Spring), pages 375–383, New York, NY, USA, 1970. ACM.

[37] Sape Mullender, editor. *Distributed systems 2nd Edition*. ACM, New York, NY, USA, 1993.

[38] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), November 1996.

[39] Rodolphe Ortalo, Yves Deswarte, and Mohamed Kaâniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Trans. Softw. Eng.*, 25(5):633–650, 1999.

[40] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[41] Francisco Perez-Sorrosal, Marta Patino-Martinez, Ricardo Jimenez-Peris, and Jaksa Vuckovic. Highly available long running transactions and activities for j2ee applications. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, ICDCS '06, pages 2–, Washington, DC, USA, 2006. IEEE Computer Society.

[42] Michael O. Rabin. Randomized byzantine generals. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, pages 403–409, Washington, DC, USA, 1983. IEEE Computer Society.

[43] Michael K. Reiter. The rampart toolkit for building high-integrity services. In *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, pages 99–110, London, UK, 1995. Springer-Verlag.

[44] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, 1, 2002.

[45] Tom Roeder and Fred B. Schneider. Proactive obfuscation. *ACM Trans. Comput. Syst.*, 28:4:1–4:54, July 2010.

[46] Eric Roman. A survey of checkpoint/restart implementations. Technical report, Lawrence Berkeley National Laboratory, Tech, 2002.

[47] Amit Sahai. Computing on encrypted data. In *ICISS*, pages 148–153, 2008.

[48] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[49] Fred B. Schneider and Lidong Zhou. Implementing trustworthy services using replicated state machines. *IEEE Security and Privacy*, 3(5):34–43, 2005.

[50] Bruce Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[51] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM.

[52] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[53] P. Sousa, N. F. Neves, and P. Verissimo. How resilient are distributed f fault/intrusion-tolerant systems? In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 98–107, Washington, DC, USA, 2005. IEEE Computer Society.

[54] P. Sousa, N. F. Neves, and P. Verissimo. Hidden problems of asynchronous proactive recovery. In *Workshop on Hot Topics in System Dependability*, June 2007.

[55] P. Sousa, N. F. Neves, P. Verissimo, and W. H. Sanders. Proactive resilience revisited: The delicate balance between resisting intrusions and remaining available. In *Proc. 25th IEEE Symposium on Reliable Distributed Systems SRDS '06*, pages 71–82, 2–4 Oct. 2006.

[56] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Trans. Parallel Distrib. Syst.*, 21(4):452–465, 2010.

[57] Ana Nora Sovarel, David Evans, and Nathanael Paul. Where's the feeb? the effectiveness of instruction set randomization. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 10–10, Berkeley, CA, USA, 2005. USENIX Association.

[58] Colin Tankard. Advanced persistent threats and how to monitor and deter them. *Network Security*, 2011(8):16 – 19, 2011.

[59] PAX team. PAX documentation on ASLR. `http://pax.grsecurity.net/docs/aslr.txt`.

[60] PAX team. PAX documentation on NoExec. `http://pax.grsecurity.net/docs/noexec.txt`.

[61] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2000.

[62] Sam Toueg. Randomized byzantine agreements. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, PODC '84, pages 163–178, New York, NY, USA, 1984. ACM.

[63] Perry Wagle and Crispin Cowan. Stackguard: Simple stack smash protection for gcc. In *Proc. of the GCC Developers Summit*, pages 243–255, 2003.

[64] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *IN PROC. SOSP*, pages 253–267. ACM Press, 2003.

[65] Tao Zhang, Xiaotong Zhuang, and Santosh Pande. Building intrusion-tolerant secure software. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 255–266, Washington, DC, USA, 2005. IEEE Computer Society.

[66] Wenbing Zhao and Honglei Zhang. Proactive service migration for long-running byzantine fault-tolerant systems. *IET Software*, 3(2):154–164, April 2009.

Appendix A

Procedures for Calculating Expected Lifetimes

The procedures for calculating the expected lifetimes defined in Chapter 5 are as follows:

A.1 Expected Lifetime for the SMR System using the SO or PR Obfuscation Scheme

```
1  double probCompromise = 0;
2  double probZero = 1;
3  double probOne = 0;
4  double alphaZero = α0;
5  double currentAlpha = α0;
6  double expectedValue = 0;
7  for (int x =0; x<=m;x++)  //m is a sufficently large constant
       to cause the expected value to converge
8  {
9          if (alphaZero < (1-x*alphaZero))
10         {
11                 currentAlpha = alphaZero/(1-x*alphaZero);
12         }
13         else
14         {
15                 currentAlpha = 1;
16         }
17         probCompromise = probZero*(6*currentAlpha^2*(1-
               currentAlpha)^2+4*currentAlpha^3*(1-currentAlpha)+
```

```
            currentAlpha^4)+probOne*(3*currentAlpha*(1-
            currentAlpha)^2+3*currentAlpha^2*(1-currentAlpha)+
            currentAlpha^3);
18          probOne = probOne*(1-currentAlpha)^3 + 4*probZero*(1-
            currentAlpha)^3*currentAlpha;
19          probZero = probZero*(1-currentAlpha)^4;
20          expectedValue=expectedValue+(probCompromise*x);
21  }
22  Print expectedValue;
```

A.2 Expected Lifetime for the PB System using the SO or PR Obfuscation Scheme

```
1   double probCompromise = 0;
2   double probAlreadyCompromised = 0;
3   double alphaZero = α0;
4   double currentAlpha = α0;
5   double expectedValue = 0;
6   for (int x =0; x<=m;x++)  //m is a sufficently large constant
        to cause the expected value to converge
7   {
8           if (alphaZero < (1-x*alphaZero))
9           {
10                  currentAlpha = alphaZero/(1-x*alphaZero);
11          }
12          else
13          {
14                  currentAlpha = 1;
15          }
16          probCompromise = (1-probAlreadyCompromised)*
                currentAlpha;
17          expectedValue=expectedValue+(probCompromise*x);
18          probAlreadyCompromised = probAlreadyCompromised +
                probCompromise;
19  }
20  Print expectedValue;
```

A.3 Expected Lifetime for the FORTRESS System using the SO or PR Obfuscation Scheme

```
1  double currentProbZero = 1;
2  double currentProbOne = 0;
3  double currentProbTwo = 0;
4  double probZero = 1;
5  double probOne = 0;
6  double probTwo = 0;
7  double probCompromise = 0;
8  double probAlreadyCompromised = 0;
9  double alphaZero = α0;
10 double currentAlpha = α0;
11 double currentAlpha = α0;
12 double expectedValue = 0;
13 for (int x =0; x<=m;x++)  //m is a sufficently large constant
       to cause the expected value to converge
14 {
15         if (alphaZero < (1-x*alphaZero))
16         {
17                 currentAlpha = alphaZero/(1-x*alphaZero);
18         }
19         else
20         {
21                 currentAlpha = 1;
22         }
23         if (serverAlpha < (1-x*serverAlpha))
24         {
25                 serverAlpha = (probZero*alphaZero)+((probOne+
                       probTwo)*(1-(x-1)*serverAlpha)/(1-x*
                       serverAlpha));
26         }
27         else
28         {
29                 serverAlpha = (probZero*alphaZero)+probOne+
                       probTwo;
30         }
31         probZero = currentProbZero;
32         probOne = currentProbOne;
33         probTwo = currentProbTwo;
34         probCompromise = (1-probAlreadyCompromised)*(probZero
               *currentAlpha^3+probOne*currentAlpha^2+probTwo*
               currentAlpha+(probOne(1-currentAlpha^2)+probTwo(1-
               currentAlpha))serverAlpha)/(probZero+probOne+
```

```
            probTwo);
35          currentProbTwo = currentProbTwo*(1-currentAlpha)+2*
            currentProbOne*(1-currentAlpha)*currentAlpha+3*
            currentProbZero*(1-currentAlpha)*currentAlpha^2;
36          currentProbOne = currentProbOne*(1-currentAlpha)^2+3*
            currentProbZero*(1-currentAlpha)^2*currentAlpha;
37          currentProbZero = currentProbZero*(1-currentAlpha)^3;
38          expectedValue=expectedValue+(probCompromise*x);
39          probAlreadyCompromised = probAlreadyCompromised +
            probCompromise;
40  }
41  Print expectedValue;
```

A.4 Expected Lifetime for the SMR System using the PO Obfuscation Scheme

```
 1  int nodesCompromised = 0;
 2  int totalCompromised = 0;
 3  int diversity = m; //where m is the amount of diversity
        available
 4  int rounds = -1;
 5  int temp = 0;
 6  double alpha = α;
 7  while (nodesCompromised < 2)
 8  {
 9          rounds++
10          int randomInt = a; //where a is a random number
            between 0 and diversity;
11          if (a<totalCompromised)
12          {
13                  nodesCompromised++;
14          }
15          randomInt = a; //where a is a random number between 0
             and diversity-1;
16          if (a<totalCompromised-nodesCompromised)
17          {
18                  nodesCompromised++;
19          }
20          randomInt = a; //where a is a random number between 0
             and diversity-2;
```

```
21          if (a<totalCompromised-nodesCompromised)
22          {
23                  nodesCompromised++;
24          }
25          randomInt = a; //where a is a random number between 0
                and diversity-3;
26          if (a<totalCompromised-nodesCompromised)
27          {
28                  nodesCompromised++;
29          }
30          temp = nodesCompromised;
31          for (int i =0; i<4-temp; i++)
32          {
33                  double randomDouble = b; //where b is a
                        random number between 0 and 1
34                  if (randomDouble < alpha)
35                  {
36                          nodesCompromised++;
37                  }
38          }
39  }
```

A.5 Expected Lifetime for the SMR System using the PO Obfuscation Scheme and Checkpointing Method *CP2*

```
1  boolean[4] nodesCompromised = new boolean[4];
2  int compromised = 0;
3  for (int i=0; i<4;i++)
4  {
5          nodesCompromised[i] = false;
6  }
7  int totalCompromised = 0;
8  int diversity = m; //where m is the amount of diversity
       available
9  double rounds = -1;
10 int temp = 0;
11 double alpha = α;
12 double beta = 1-(1-α)^0.25)
13 while (compromised < 2)
```

```
14 {
15         rounds = rounds+0.25;
16         for (int i=0;i<3;i++)
17         {
18                 nodesCompromised[i] = nodesCompromised[i+1];
19         }
20
21         int randomInt = a; //where a is a random number
               between 0 and diversity;
22         if (a<totalCompromised)
23         {
24                 nodesCompromised[3] = true;
25         }
26         else
27         {
28                 nodesCompromised[3] = false;
29         }
30
31         temp = nodesCompromised;
32         for (int i =1; i<4-temp; i++)
33         {
34                 double randomDouble = b; //where b is a
                       random number between 0 and 1
35                 if (randomDouble < beta)
36                 {
37                         nodesCompromised[i] = true;
38                 }
39         }
40         compromised = 0;
41         for (int i=1;i<4-temp;i++)
42         {
43                 if (nodesCompromised[i])
44                 {
45                         compromised++;
46                 }
47         }
48 }
```

A.6 Expected Lifetime for the FORTRESS System using the PO Obfuscation Scheme

A.6.1 Indirect Attacks Impossible

```
1  int proxiesCompromised = 0;
2  int totalCompromised = 0;
3  int diversity = m; //where m is the amount of diversity
       available
4  int rounds = -1;
5  int temp = 0;
6  boolean serverCompromised = false;
7  double alpha = α;
8  while (proxiesCompromised < 3 && !serverCompromised)
9  {
10          rounds ++;
11          int randomInt = a; //where a is a random number
               between 0 and diversity;
12          if (a<totalCompromised)
13          {
14                  proxiesCompromised++;
15          }
16          randomInt = a; //where a is a random number between 0
                and diversity-1;
17          if (a<totalCompromised-proxiesCompromised)
18          {
19                  proxiesCompromised++;
20          }
21          randomInt = a; //where a is a random number between 0
                and diversity-2;
22          if (a<totalCompromised-proxiesCompromised)
23          {
24                  proxiesCompromised++;
25          }
26          temp = proxiesCompromised;
27          for (int i =0; i<3-temp; i++)
28          {
29                  double randomDouble = b; //where b is a
                       random number between 0 and 1
30                  if (randomDouble < alpha)
```

```
31                  {
32                          proxiesCompromised++;
33                  }
34          }
35          if (proxiesCompromised > 0)
36          {
37                  double randomDouble = b; //where b is a
                        random number between 0 and 1
38                  if (randomDouble < alpha)
39                  {
40                          serverCompromised = true;
41                  }
42          }
43  }
```

A.6.2 Indirect Attacks Possible

```
1  int proxiesCompromised = 0;
2  int totalCompromised = 0;
3  int diversity = m; //where m is the amount of diversity
       available
4  int rounds = -1;
5  int temp = 0;
6  boolean serverCompromised = false;
7  double alpha = α;
8  double kappa = κ;
9  while (proxiesCompromised < 3 && !serverCompromised)
10 {
11          rounds ++;
12          int randomInt = a; //where a is a random number
               between 0 and diversity;
13          if (a<totalCompromised)
14          {
15                  proxiesCompromised++;
16          }
17          randomInt = a; //where a is a random number between 0
                and diversity-1;
18          if (a<totalCompromised-proxiesCompromised)
19          {
20                  proxiesCompromised++;
21          }
```

```
22          randomInt = a; //where a is a random number between 0
                and diversity-2;
23          if (a<totalCompromised-proxiesCompromised)
24          {
25                  proxiesCompromised++;
26          }
27          temp = proxiesCompromised;
28          for (int i =0; i<3-temp; i++)
29          {
30                  double randomDouble = b; //where b is a
                        random number between 0 and 1
31                  if (randomDouble < alpha)
32                  {
33                          proxiesCompromised++;
34                  }
35          }
36          if (proxiesCompromised > 0)
37          {
38                  double randomDouble = b; //where b is a
                        random number between 0 and 1
39                  if (randomDouble < alpha)
40                  {k
41                          serverCompromised = true;
42                  }
43          }
44          else
45          {
46                  double randomDouble = b; //where b is a
                        random number between 0 and 1
47                  if (randomDouble < kappa*alpha)
48                  {
49                          serverCompromised = true;
50                  }
51          }
52  }
```

Appendix B

System Models with Checkpointing Methods CP1 and CP2

When considering refresh or replacement using checkpointing with methods *CP1* and *CP2* we no longer have a unit time-step in which all nodes process client requests for the whole of the unit time-step, followed by a short migration period in which all nodes are replaced. Instead we have a situation where each node is unavailable due to refresh or replacement for part of the unit time-step during which the other nodes are continuing to process requests. We also have each incoming node receiving its state from every node other than the outgoing node.

This results in a compromised outgoing node being unable to influence the state of the incoming node that will replace it. Hence it is no longer sufficient to compromise any two nodes at any point in a unit time-step to compromise the system state. Instead, an attacker must compromise two of the three nodes that will provide the system state to the next incoming node.

It is however possible for two compromised nodes, one of which will be the next outgoing node, to collaborate to compromise the system state returned to clients before the outgoing node leaves. So, a lesser failure state will be reached when the failure conditions for the standard *SMR* system using the *PO* obfuscation scheme are reached, but the failure conditions detailed in this section are not.

This lesser failure state is not reachable however when checkpoining method *CP2* is used and each node has a reboot period equal to 1/4 of the unit time step. Here, there are only ever three nodes processing client requests at any given time. Hence, there are only three nodes that can be compromised at any given time, and the only possible failure state is when two of these three nodes are compromised, and will hence be able to give a corrupted state to the incoming node when it finishes rebooting.

We also note that systems using these checkpointing mechanisms will have the undesirable property relative to systems using en-masse replacement that, while a node

is rebooting, one intrusion will be sufficient to allow an attacker to prevent the ordering, and hence processing, of client requests. This becomes less of a problem the smaller that reboot periods are relative to the unit time-step. However, as the intrusion resilience, both against a permanent corruption of system state and a corruption of client state, of systems using these checkpointing methods increases as the reboot periods increase relative to the unit time-step, there is a trade-off between the increase in intrusion resilience and the possible decrease in availability.

Appendix C

Comparison Between Expected Lifetimes for SMR Systems using the PO Obfuscation Scheme with En-masse Replacement and Checkpointing Method CP2

Here we present comparisons between the SMR system with en-masse replacement and the SMR system using checkpointing method *CP2* for a representative range of values. Expected lifetimes are shown for the range of intrusion probabilities $\alpha = 0.0001$ to $\alpha = 0.001$ with the diversity levels: $2^8, 2^{16}$ and infinite diversity in Table C.1, showing that the use of the *CP2* checkpointing method increases the expected lifetime of the SMR system.

Table C.1: Expected Lifetimes of Systems with Proactive Obfuscation

0.0001	*SMR*	7726	99073	1.7×10^7
	SMR-CP2	9046	231393	7.11×10^8
0.0002	*SMR*	4464	59599	4.2×10^6
	SMR-CP2	5264	143600	1.78×10^8
0.0003	*SMR*	3270	43864	1.9×10^6
	SMR-CP2	3851	105691	7.9×10^7
0.0004	*SMR*	2633	35163	1.0×10^6
	SMR-CP2	3086	83950	4.44×10^7
0.0005	*SMR*	2190	29529	667111
	SMR-CP2	2591	71763	2.84×10^7
0.0006	*SMR*	1935	25235	463333
	SMR-CP2	2267	62175	1.97×10^7

α_i	System	$EL(2^8)$	$EL(2^{16})$	$EL(\infty)$
0.0007	*SMR*	1716	22339	340454
	SMR-CP2	2017	54447	1.45×10^7
0.0008	*SMR*	1555	19831	260695
	SMR-CP2	1836	49792	1.11×10^7
0.0009	*SMR*	1429	17949	206008
	SMR-CP2	1673	45335	8776953
0.001	*SMR*	1310	16165	166889
	SMR-CP2	1536	41324	7109134

We note that a comparison between the SMR system using checkpointing method *CP2* and the *FORTRESS* system may not strictly be a like-to-like comparison. The *SMR* system using checkpointing method *CP2* is considered not to be compromised when one node is compromised, despite the fact that this will prevent system availability for between 1/4 and 3/4 of a unit time-step. This is in direct contrast to the inclusion of the compromise of all three proxies in the compromise conditions for the *FORTRESS* system, a situation that will only prevent the availability of the system for one unit time-step.

Appendix D

Transfer Mechanisms for Systems with State Machine Replication in the Server Tier

We note that the transfer mechanisms given in Section 6.1 all assume a primary-backup system, although any of them other than progressive transfer with primary load reduction could be implemented with a server tier consisting of a single node. However, all of these mechanisms, with the exception of progressive transfer with primary load reduction, can easily be modified to be used with a server tier using active replication. This is illustrated in the following sections.

D.1 Single Transfer

The single transfer mechanism behaves as in Section 6.1.2.1 during the processing section of each unit time-step, and ceases processing and queues outstanding client requests when the processing section ends. Each node now generates a checkpoint and sends it to every server that will perform processing in the next time-step. Each of these new servers waits until it has received $n + 1$ identical checkpoints, where n is the number of intrusions that the active replication system is designed to be able to survive in any given unit time-step. Each server sets its state from this checkpoint.

D.2 Progressive Transfer

Two subsets of server nodes are used as in Section 6.1.2.2, except that every server node is differently randomised, as active replication is designed to tolerate intrusion into some server nodes. Every node in the current server tier periodically generates a checkpoint and sends it to every node in the second subset of server nodes. These

nodes update their state every time they receive $n+1$ identical checkpoints, where n is the number of intrusions that the system is designed to be able to survive in any given unit time-step. At system migration the second subset of server nodes become the new server tier, and a new subset becomes the new second subset.

D.3 Progressive Transfer with Primary Load Reduction

Progressive transfer with primary load reduction is not a viable scheme with active replication, as the nature of active replication requires that all correct servers process all requests. Hence all servers will be expected to handle an equal load. Furthermore, as some nodes may be intruded into, having only some nodes send state transfer messages to the second subset of server nodes would have security implications.

D.4 Transfer with Trusted Components

Here we extend the transfer mechanism from Section D.1 or Section D.2 with the aid of of trusted components. In Section 6.1.2.4, these trusted components took a single system checkpoint that was randomised correctly to be used by the current server tier, and converted it to be correctly randomised to be used by the new server tier. Instead, we convert each checkpoint into an intermediate state, determine the correct checkpoint that $n+1$ or more nodes agree on, and convert this to make a valid checkpoint for each server in the new server tier. Here, there is a possibility for a greater gain in efficiency than simply using this procedure to translate each checkpoint received into a valid checkpoint for each new server.

The efficiency gain in this technique can be illustrated with the following example:

First, we assume that there are m nodes in a server tier, and all of the m nodes in the current server tier send checkpoints to the trusted components. If we were to use the trusted components simply to send each checkpoint to each node in the new server tier we would transform each of the m checkpoints into a set of m checkpoints and send that to the new server tier. This would result in a total of m^2 transformations and $m + m^2$ messages. On the other hand, if we convert each of the checkpoints into an intermediate form, this takes m transformations. Then, we take the majority checkpoint and transform it for every new server node, requiring another m transformations. This results in a total of $2m$ transformations and $2m$ messages.

If we take an active replication system with the commonly used number of 4 nodes, we see that the unoptimised method would require 16 transformations and 20 messages, whereas the optimised method would require 8 transformations and 8 messages.

As a side remark we note that when using active replication, the assumption that all server nodes are uncompromised no longer holds. This means that we have potentially compromised nodes sending checkpoint messages to the trusted components, possibly resulting in a higher requirement for attack resilience in the trusted components.

Appendix E

Correctness, Liveness and Attack Resilience Analysis of State Transfer Mechanisms

We first consider the correctness assumptions we can make about the FORTRESS system itself, in light of the performance evaluations in Chapter 5. Then, we consider the correctness and liveness requirements for each state transfer mechanism, and also what impact that state transfer mechanism can have on the assumptions we have made when assessing attack resilience. Finally, we consider the changes made to our correctness assumptions and analysis of these mechanisms when active replication is used in the server tier.

E.1 Correctness Assumptions

In Chapter 5 we have evaluated the expected lifetime of a FORTRESS system with primary backup replication in the server tier. The expected lifetime is the average number of unit time-steps that the system will run until it is compromised. Another way of stating this is that the expected lifetime is the average number of unit time-steps for which the following assumptions will hold.

1. No server replica is compromised by a malicious intruder.
2. At least one proxy is not compromised by a malicious intruder.

It is also possible, by including sufficient backups in the replication system to make the following assumption hold

3. At least one backup does not crash during the unit-time step.

This allows us to make the following correctness assumptions for an uncompromised system, and expect them to hold for the calculated expected lifetime.

1. There is at least one correct server replica available.
2. There is at least one correct proxy available.
3. No server replica has been maliciously intruded.

E.2 Single Transfer

Here we consider the single transfer technique presented in Section 6.1.2.1.

E.2.1 Correctness

We consider the case where a node has set its state from a message received. We know that this message came from a node that was a server in the last unit time-step due to assumption 6.1.3 and that this node was correct due to assumption 3 in Section E.1. Thus the node has received a correct state.

The only other threat to correctness is that the replica could have received a malicious message from the controller unit or reboot server, causing it to re-start in such a way that some or all of the system state being transferred is lost. However, the controller unit and reboot server are assumed to be uncompromised for the lifetime of the system, so we can discount this possibility.

E.2.2 Liveness

The time at which a correct node sends a state message is bounded due to assumption 1 and the state transfer message will be received by all new nodes within a bounded time interval from this, due to assumption 6.1.3. Hence, if a correct node exists for all of the unit time-step than a state transfer message will be received within a bounded time by all new nodes.

We can guarantee that at least one correct node exists due to assumption 1 in Section E.1. Hence a state transfer message will be received within a bounded time by every new node.

E.2.3 Attack Resilience

The new set of nodes is accessible only by the old primary and the controller unit until the end of the transfer phase. The controller unit is assumed to be uncompromised and

hence will not send any malicious messages to the new set of nodes. The accessibility from the old primary is one-way with the new set of nodes sending no information back to the old set. This ensures that no external attacks can be launched against the new nodes until the start of their processing phase. The old primary is assumed to be correct until the end of the transfer phase by assumption 3 in Section E.1. Hence there is no alteration to the attack resilience assumptions made when calculating the expected lifetime.

E.3 Progressive Transfer

Here we consider the progressive transfer technique presented in Section 6.1.2.2.

E.3.1 Correctness

We know that each new node has only received updates from the subset of old nodes due to assumption 6.1.3 and that the only other messages it has received are from the controller unit or reboot server. The controller unit and reboot server are assumed to be uncompromised, and hence have not sent any malicious messages. We also know that all updates have been sent and the updates that have been sent are correct due to assumption E.1. This, coupled with the fact that all sent updates will have been received due to assumption 1 and assumption 6.1.3, ensures correct state in the new nodes.

E.3.2 Liveness

The times at which a correct nodes sends state messages are bounded due to requirement 1 in Section 1 and the state transfer message will be received by all new nodes within a bounded time interval from this, due to requirement 2 6.1.3 in Section . Hence, if a correct server node exists for all of the unit time-step than a full set of state transfer messages will be received within a bounded time by all new nodes.

We can guarantee that at least one correct server node exists due to assumption 1 in Section E.1. Hence a full set of state transfer messages will be received within a bounded time by every new node.

E.3.3 Attack Resilience

The new subset of nodes are only one way accessible by the old subset of nodes, and they do not perform processing or receive client requests until the unit time-step when they become the current server nodes. This results in attack resilience being

unchanged if we add the assumption that there is no way in which the new nodes can be attacked independently.

The only other possible avenue of attack is malicious messages sent by the controller unit. However, we assume that the controller unit is uncompromised for the lifetime of the system, and thus we can discount this possibility.

E.4 Progressive Transfer with Primary Load Reduction

Here we consider the technique presented in Section 6.1.2.3.

E.4.1 Correctness

The only change to the state transfer scheme in Section E.3 is to have one of the backups send checkpoints rather than the primary, if a backup is available. The backups are just as difficult to attack as the primary, do not perform any processing until they become the primary, and are not directly contactable by an attacker in the same way that the primary is not directly contactable. The use of primary load reduction makes no difference to the attack resilience of the controller unit or reboot server.

Hence the same correctness arguments made in Section E.3 apply.

E.4.2 Liveness

There are two possible cases for this transfer scheme. When the only non-crashed node is the primary then we have liveness as in Section E.3 due to assumption E.1 stating that the primary will not crash. When there is at least one non-crashed node other than the primary then the liveness assumptions in Section E.2.2 hold for both the checkpoints sent from the primary to the backup, and the checkpoints sent from the backup to the new nodes. Similarly, when there is at least one non-crashed node other than the primary and the primary crashes, the liveness assumptions in Section E.2.2 mean that the highest ordered backup will have received, or will receive within bounded time the most recent checkpoint. Then, the liveness assumptions in Section E.2.2 hold for this node now it has taken over as the primary. Hence liveness holds, as it holds in either case, and is not affected by the transition between them.

E.4.3 Attack Resilience

The attack resilience arguments in E.3.3 hold for this case as we are simply changing the node sending the state messages to the new nodes. The backup node sending the state messages only applies and sends updates rather than performing processing itself, it is identically randomised to the primary, and does not directly accept client requests unless it becomes the primary.

E.5 Transfer with Trusted Components

Here we consider the technique presented in Section 6.1.2.4.

E.5.1 Correctness

The correctness arguments in Section E.3 hold for a correct trusted server producing correct system state from checkpoints sent by the old servers, and the correctness arguments in Section E.2 hold for a correct trusted server receiving a correct system state from the old servers in single transfer. Similarly these correctness arguments can be applied to the transfer of system state from a correct trusted server to the new servers.

Hence, as long as the assumptions hold that the trusted server is uncompromised and performs correctly, the correctness requirement is satisfied.

E.5.2 Liveness

The liveness arguments in Section E.3 and Section E.2 hold for a correct trusted server receiving state transfer information from the old servers and transmitting it to the new servers. Hence, as long as the assumptions hold that the trusted server is correct and performs in a timely manner, the liveness requirement is satisfied.

E.5.3 Attack Resilience

The attack resilience arguments in Section E.3 and Section E.2 will hold for the old and new servers. Hence, as long as we assume that the trusted server is not open to outside attack, this scheme will cause no change to our attack resilience measurements. This assumption is fairly safe to make as we know that the trusted component only receives state messages from the current primary, which is assumed to be correct, and does not perform any processing beyond applying a transformation of the state messages received to cause them to be randomised in the correct way for the new subset of servers.

E.6 Hardware Requirements

For each transfer mechanism we will consider the additional hardware needed beyond that for one FORTRESS system. We will not consider additional hardware that may be needed to prevent exhaustion of the spare pool, only hardware that is needed to be take part in current computations or state transfer. This is in line with the consideration in Section 3.7.2 that machines that are not currently performing computations or receiving updates can be left in power saving modes, and the observation in Section 2.12 that the cost of running machines is a major part of the hardware cost of a system.

E.6.1 Single Transfer

Here we will require a second set of servers to be available for the transfer period. These servers are not required during the processing period, as no updates will be sent until the transfer period.

E.6.2 Progressive Transfer

A second set of servers are required for all of the unit time-step, as updates may be sent for the whole of the unit time-step.

E.6.3 Progressive Transfer with Primary Load Reduction

A second set of servers is required for all of the unit time-step, as updates may be sent for all of the unit time-step.

E.6.4 Transfer with Trusted Components

Here we require the additional hardware for the type of transfer that the trusted components support and the trusted components themselves. The trusted components will require at least one server and possibly more, depending on any requirements for replication, these trusted components will need to be available for all of the transfer period if single transfer is being used, or all of the unit time-step if progressive transfer is being used.

E.7 Transfer Mechanisms for Systems with Active Replication in the Server Tier

The use of active replication immediately invalidates our assumption in E.1 that no server has been maliciously intruded into. This is replaced by an assumption that the number of servers that have been maliciously intruded into is bounded by some number f. We will outline how this affects correctness, liveness and attack resilience in the following sections.

E.7.1 Correctness

The previous assumption made for each mechanism was that the state transfer message received was correct as it was sent from a correct server. Here, instead we have a situation where a number of state transfer messages are received, some of which may be incorrect. However, as at most f of these messages can be incorrect, the active replication system can use $3f + 1$ or more nodes. Then, we know that, as soon as $2f + 1$ messages are received that $f + 1$ of them must be correct and identical. Hence, a correct message can be identified by finding $f + 1$ identical messages. Once this correct message is identified, the correctness arguments made when primary-backup replication is used will hold.

E.7.2 Liveness

The liveness arguments made when the state transfer mechanisms were considered for primary-backup server tiers show that, as long as a correct node exists to send messages, liveness holds. Hence, for liveness to hold in a system using active replication in the server tier, we need to show that intruded nodes cannot indefinitely delay the identification of a correct message.

First we assume the worst case scenario, that there are f intruded nodes. Then, the attacker has two choices to attempt to disrupt state transfer, either to send incorrect messages, or fail to send messages. If the f intruded nodes send f incorrect messages (the maximum number of unique incorrect messages they can send) then, by the liveness arguments made in sections E.2-E.5, the non-intruded nodes will also send correct messages within bounded time. If we assume that the incorrect messages are among the first $2f + 1$, then a decision is made using n incorrect messages and $f + 1$ correct messages, giving a correct outcome within bounded time.

On the other hand, if the intruded nodes fail to send messages, or delay sending them then, due to the liveness arguments made in sections E.2-E.5, $2f + 1$ correct messages will be received within bounded time. This will again result in a correct outcome within bounded time.

E.7.3 Attack Resilience

We have previously assumed that nodes are not attacked until the unit time-step in which they become the current server and proxy tier. This assumption has been based on two key observations. Firstly, the identities of the nodes that will be used in the next unit time-step are not known externally or by the current proxies, and secondly, no node in the current server tier has been maliciously intruded. When active replication is used in the server tier, the second observation can no longer be assumed to be true.

This means that, when single transfer is used, malicious requests may be sent to the new server tier during the transfer phase. Even more seriously, when progressive transfer is used, malicious requests may be sent to the new server tier at any point during the preceding unit time-step.

We note that this possible cause of reduced attack resilience may be mitigated by the use of trusted components if the trusted components are simple and attack resilient enough themselves to withstand any malicious requests that intruded servers may send.

Appendix F

State Transfer Overhead: Absolute Values

Here we present the absolute values recorded during the state transfer overhead experiments detailed in Chapter 7.

F.1 Update Interval

The migration interval was held constant at 150 seconds, and the heartbeat interval was held constant at 4 seconds.The latencies measured are shown in Figure F.1. The throughputs measured are shown in Figure F.2. Confidence intervals at the 95% significance level are shown for the latencies in Table F.1 and the throughputs in Table F.2.

F.2 Heartbeat Interval

We first present the results obtained from a FORTRESS system with migration interval held constant at 150 seconds and update interval held constant at 40 seconds. The latencies measured are shown in Figure F.3 and the throughputs measured are shown in Figure F.4. Confidence intervals at the 95% significance level are shown for the latencies in Table F.3 and the throughputs in Table F.4.

We then present results for the same system using primary-backup replication without proactive fortification. This latencies measured are shown in Figure F.5. Confidence intervals at the 95% significance level are shown for the latencies in Table F.5 and the throughputs in Table F.6.

Figure F.1: Latency as Migration Interval Varies - 10s Update Interval, 4s Heartbeat Interval

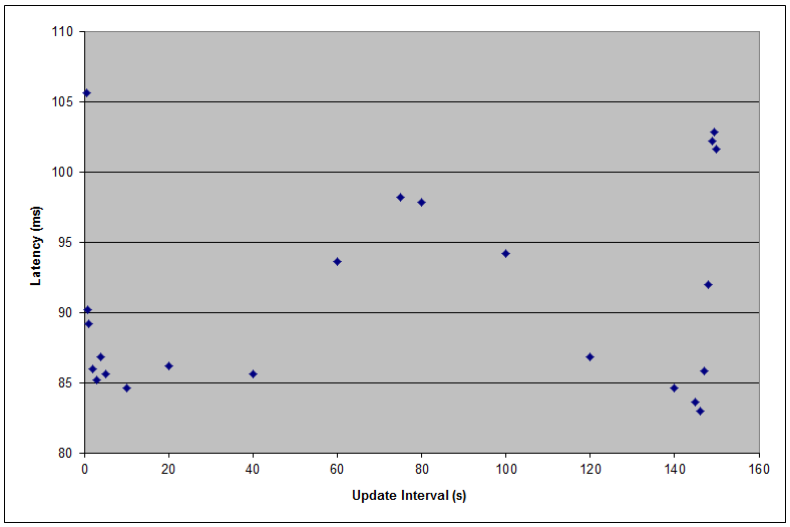


Figure F.2: Throughput as Migration Interval Varies - 10s Update Interval, 4s Heartbeat Interval

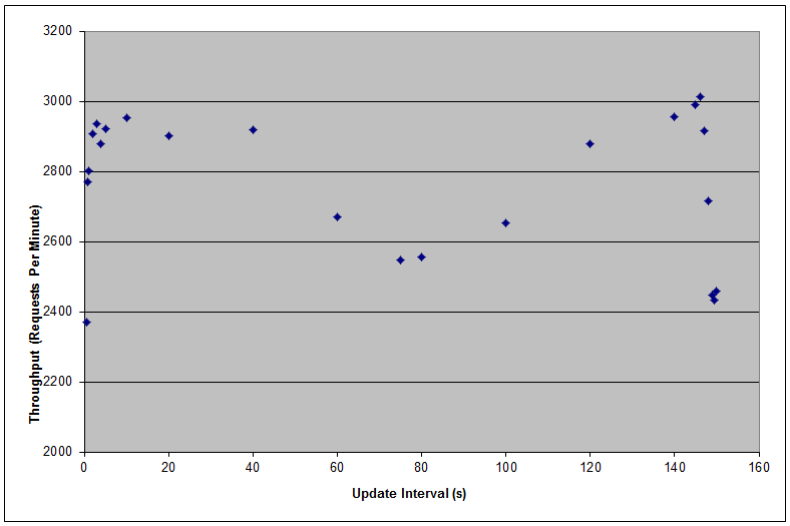


Table F.1: 95% Confidence Intervals for Latencies of the FORTRESS System

Update Interval(s)	Lower Endpoint(ms)	Upper Endpoint (ms)
0.5	104.79	106.41
0.75	89.23	91.17
1	88.38	90.02
2	85.07	86.93
3	84.25	86.15
4	85.79	87.81
5	84.68	86.52
10	83.77	85.43
20	85.33	87.07
40	84.68	86.52
60	92.69	94.51
75	97.31	99.09
80	96.9	98.7
100	93.24	95.16
120	85.87	87.73
140	83.6	85.6
145	82.63	84.57
146	82.01	83.99
147	84.87	86.73
148	91.11	92.89
149	101.28	103.12
149.5	101.93	103.67
150	100.75	102.45

Figure F.3: Latency as Heartbeat Interval Varies - 150s Migration Interval, 40s Update Interval

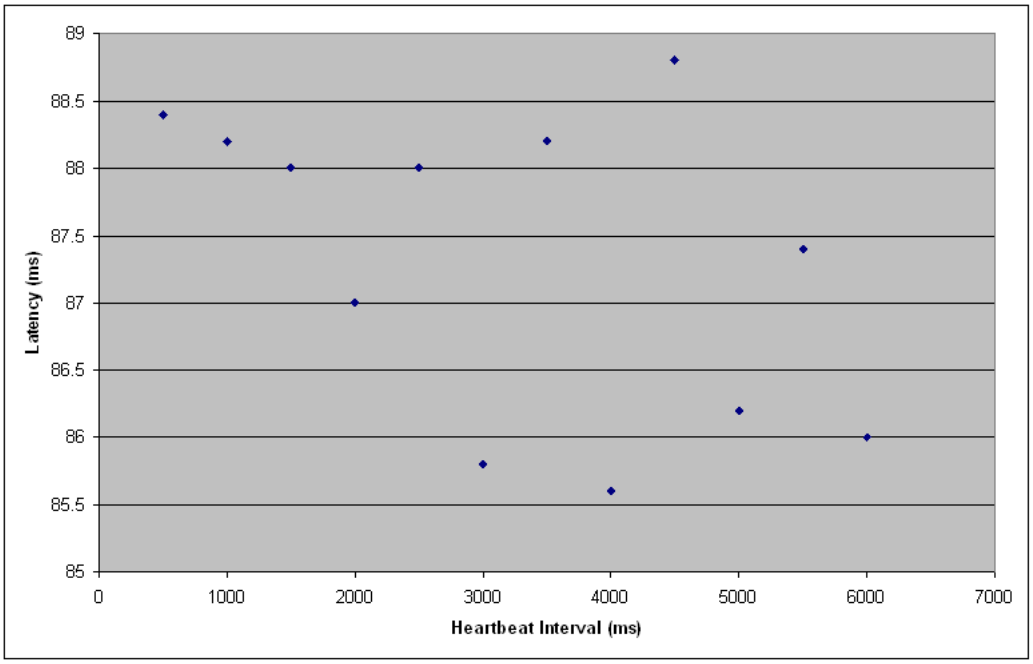


Table F.2: 95% Confidence Intervals for Throughput of the FORTRESS System

Update Interval(s)	Lower Endpoint(Requests per Minute)	Upper Endpoint (Request per Minute)
0.5	2330.96	2407.88
0.75	2730.91	2808.93
1	2762.11	2839.27
2	2866.23	2945.73
3	2894.27	2974.59
4	2839.38	2918.99
5	2881.55	2959.57
10	2912.78	2993.59
20	2860.67	2941.79
40	2880.67	2958.41
60	2631.43	2710.44
75	2507.22	2584.43
80	2516.52	2594.07
100	2614.88	2692.97
120	2839.62	2919.57
140	2915.38	2994.78
145	2950.90	3027.96
146	2972.43	3051.67
147	2876.03	2955.44
148	2675.10	2753.66
149	2405.51	2486.86
149.5	2393.30	2472.51
150	2419.97	2499.46

Figure F.4: Throughput as Heartbeat Interval Varies - 150s Migration Interval, 40s Update Interval

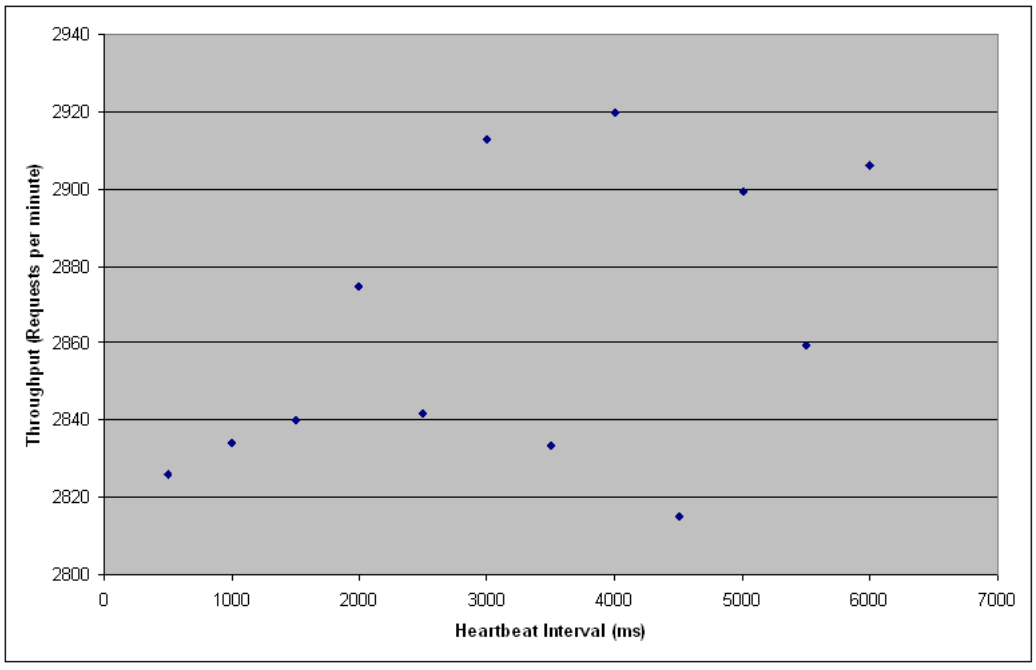


Table F.3: 95% Confidence Intervals for Latencies of the FORTRESS System

Heartbeat Interval(ms)	Lower Endpoint(ms)	Upper Endpoint (ms)
500ms	87.53	89.27
1000ms	87.18	89.22
1500ms	87.11	88.89
2000ms	85.96	88.04
2500ms	86.9	89.1
3000ms	84.87	86.73
3500ms	87.11	89.29
4000ms	84.63	86.57
4500ms	87.86	89.74
5000ms	85.22	87.18
5500ms	86.41	88.39
6000ms	84.96	87.04

Table F.4: 95% Confidence Intervals for Throughputs of the FORTRESS System

Heartbeat Interval(ms)	Lower Endpoint(Requests per Minute)	Upper Endpoint (Requests per Minute)
500ms	2786.917	2865.197
1000ms	2795.233	2873.273
1500ms	2799.828	2880.028
2000ms	2834.713	2914.453
2500ms	2803.359	2880.439
3000ms	2873.523	2951.983
3500ms	2794.649	2872.529
4000ms	2880.573	2958.513
4500ms	2776.037	2854.657
5000ms	2860.173	2938.313
5500ms	2820.662	2898.182
6000ms	2866.868	2945.108

Table F.5: 95% Confidence Intervals for Latencies of the Primary-Backup System

Heartbeat Interval(ms)	Lower Endpoint(ms)	Upper Endpoint (ms)
500ms	82.39	84.41
1000ms	81.24	83.16
1500ms	81.07	82.93
2000ms	83.31	85.09
2500ms	82.04	83.96
3000ms	81.06	82.94
3500ms	83.37	85.43
4000ms	80.83	82.77
4500ms	82.45	84.35
5000ms	82.38	84.42
5500ms	82.22	84.18
6000ms	80.44	82.36

Figure F.5: Latency as Heartbeat Interval Varies in a Primary-Backup System

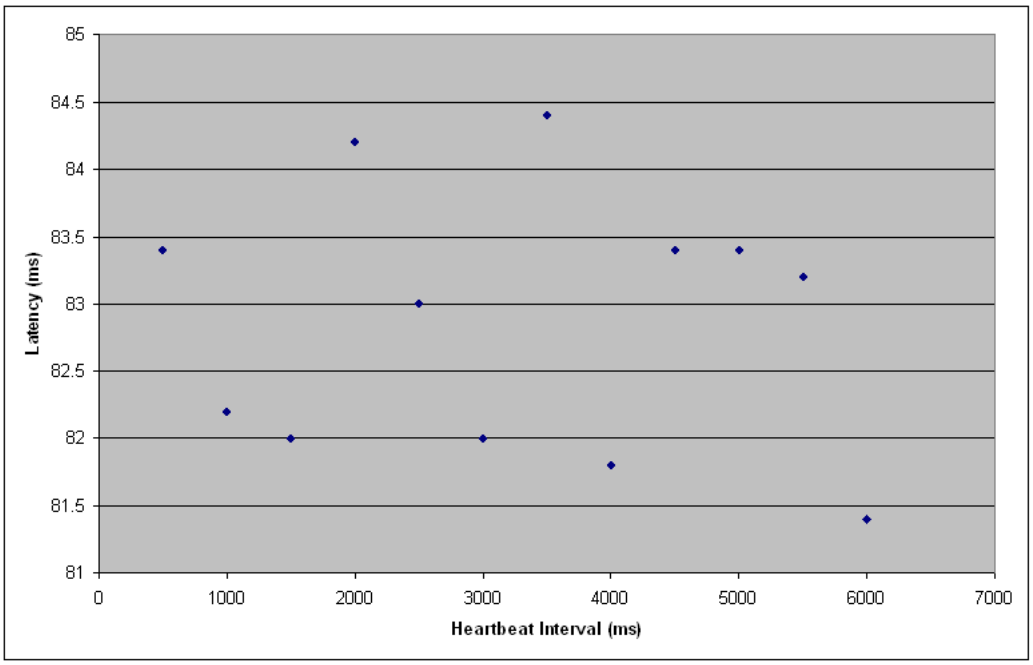


Figure F.6: Throughput as Heartbeat Interval Varies in a Primary-Backup System

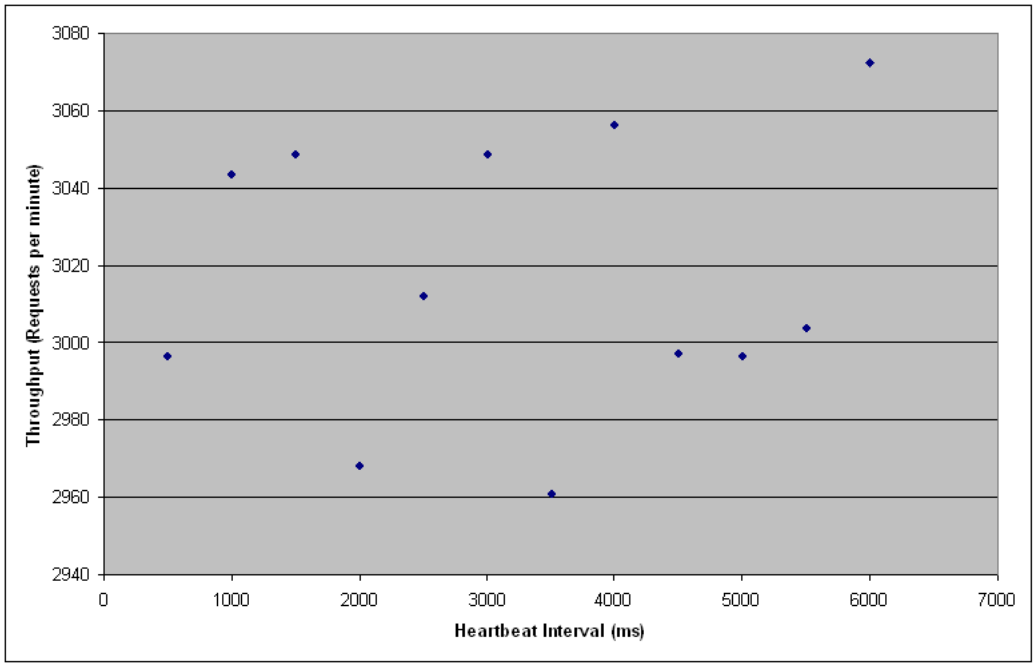


Table F.6: 95% Confidence Intervals for Throughputs of the Primary-Backup System

Heartbeat Interval(ms)	Lower Endpoint(Requests per Minute)	Upper Endpoint (Requests per Minute)
500ms	2894.23	3099.03
1000ms	2943.77	3142.97
1500ms	2948.56	3149.02
2000ms	2870.80	3065.44
2500ms	2910.64	3113.46
3000ms	2949.28	3148.28
3500ms	2861.16	3061.20
4000ms	2958.26	3154.22
4500ms	2898.19	3096.49
5000ms	2895.17	3098.07
5500ms	2901.84	3105.86
6000ms	2972.40	3172.14

F.3 Migration Interval

The update interval was held constant at 10 seconds, and the heartbeat interval was held constant at 4 seconds.The latencies measured are shown in Figure F.7. The throughputs measured are shown in Figure F.8. Confidence intervals at the 95% significance level are shown for the latencies in Table F.7 and the throughputs in Table F.8.

Figure F.7: Latency as Migration Interval Varies - 10s Update Interval, 4s Heartbeat Interval

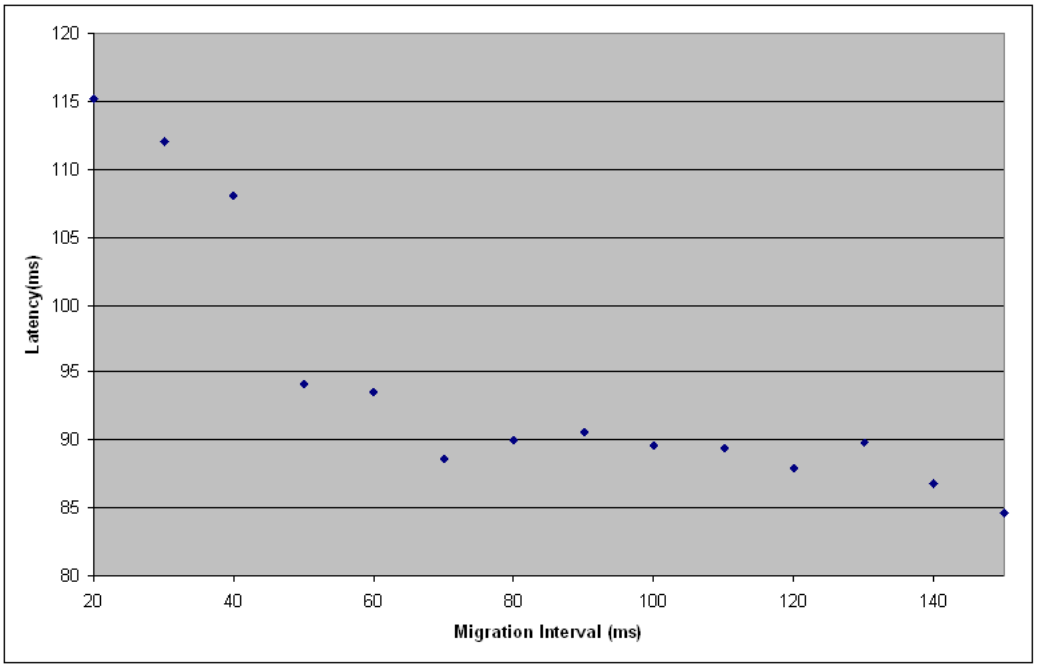


Figure F.8: Throughput as Migration Interval Varies - 10s Update Interval, 4s Heartbeat Interval

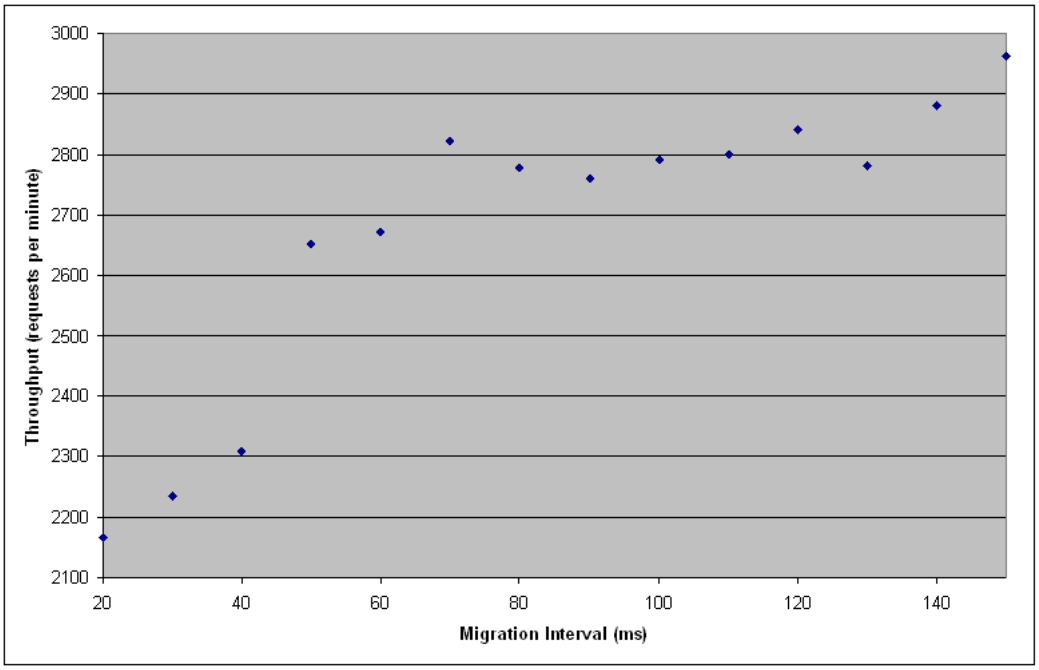


Table F.7: 95% Confidence Intervals for Latencies of the FORTRESS System

Migration Interval (ms)	Lower Endpoint (ms)	Upper Endpoint (ms)
20000	114.21	116.19
30000	110.99	113.01
40000	107.18	108.82
50000	93.00	95.40
60000	92.61	94.59
70000	87.53	89.67
80000	89.10	90.896
90000	89.50	91.70
100000	88.59	90.61
110000	88.45	90.35
120000	87.13	88.87
130000	88.75	90.85
140000	85.87	87.73
150000	83.68	85.52

Table F.8: 95% Confidence Intervals for Throughput of the FORTRESS System

Migration Interval (ms)	Lower Endpoint (Requests per Minute)	Upper Endpoint (Requests per Minute)
20000	2126.85	2203.352
30000	2194.13	2274.153
40000	2270.44	2349.195
50000	2612.81	2691.048
60000	2631.99	2709.89
70000	2781.44	2861.9004
80000	2738.47	2817.088
90000	2720.31	2798.451
100000	2751.68	2828.678
110000	2759.25	2837.59
120000	2801.42	2880.399
130000	2742.97	2820.954
140000	2840.97	2919.3943
150000	2922.93	3003.233

Appendix G

Apache Tomcat Implementation: Absolute Values

G.1 Simple Web Page with Sessions

G.1.1 Heartbeat Interval

The migration interval was fixed at 100 seconds for the proactively fortified system and the heartbeat interval was varied between 1 second and 6 seconds in 1 second steps. The results are shown in Figure G.1 and Figure G.2. Confidence intervals at the 95% significance level are shown for the latencies of the proactively fortified system in Table G.1 and the latencies of the primary-backup system in Table G.2. Confidence intervals at the 95% significance level are shown for the throughputs of the proactively fortified system in Table G.3 and the latencies of the primary-backup system in Table G.4.

This data appears to show latency staying fairly constant for the proactively fortified system as the heartbeat interval increases, until it reaches the 5000ms point, after

Figure G.1: Latency as Heartbeat Interval Varies - 100s Migration Interval

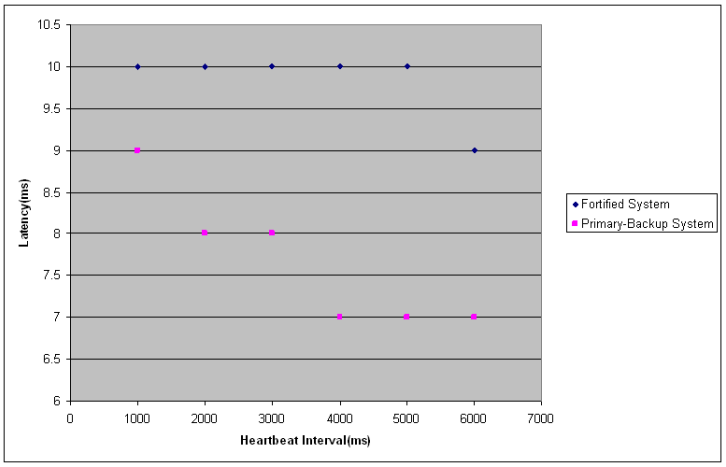


Figure G.2: Throughput as Heartbeat Interval Varies - 100s Migration Interval

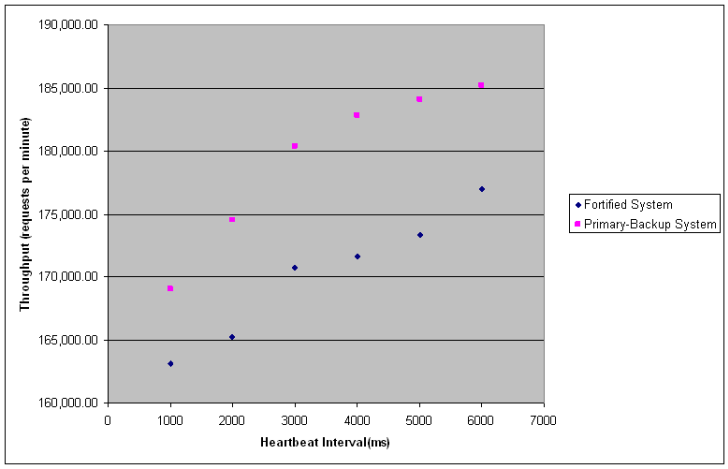


Table G.1: 95% Confidence Intervals for Latencies of the Proactively Fortified System

Heartbeat Interval(ms)	Lower Endpoint (ms)	Upper Endpoint (ms)
1000	9.92	10.08
2000	9.92	10.08
3000	9.92	10.08
4000	9.92	10.08
5000	9.92	10.08
6000	8.92	9.08

Table G.2: 95% Confidence Intervals for Latencies of the Primary-Backup System

Heartbeat Interval(ms)	Lower Endpoint (ms)	Upper Endpoint (ms)
1000	8.94	9.06
2000	7.94	8.06
3000	7.94	8.06
4000	6.96	7.04
5000	6.95	7.05
6000	6.95	7.05

Table G.3: 95% Confidence Intervals for Throughputs of the Proactively Fortified System

Heartbeat Interval(ms)	Lower Endpoint (Requests per Minute)	Upper Endpoint (Requests per Minute)
1000	161,895.34	164,395.08
2000	164,020.98	166,448.28
3000	169,479.65	171,933.81
4000	170,425.42	172,863.28
5000	172,086.32	174,621.94
6000	175,654.59	178,229.61

Table G.4: 95% Confidence Intervals for Throughputs of the Primary-Backup System

Heartbeat Interval(ms)	Lower Endpoint (Requests per Minute)	Upper Endpoint (Requests per Minute)
1000	167,764.72	170,263.46
2000	173,289.82	175,797.02
3000	179,139.03	181,654.71
4000	181,601.14	184,043.30
5000	182,859.63	185,350.37
6000	183,974.95	186,378.91

Table G.5: Correlation Coefficients for the Comparison Between Latency and Throughput and Heartbeat Interval

Data Series	Correlation Coefficient
Latency for Proactively Fortified System	-0.65465
Throughput for Proactively Fortified System	0.980192
Latency for Primary-Backup System	-0.91652
Throughput for Primary-Backup System	0.946763

which there is a decrease. The primary-backup system without fortification appears to show a general decreases in latency as heartbeat interval increases. Throughput appears to increase as heartbeat interval increases both for the proactively fortified system and the primary-backup system without proactive fortification. Correlation coefficients are shown for these results in Table G.5. This shows a weak negative correlation between latency and heartbeat interval, and a strong positive correlation between throughput and heartbeat interval for the proactively fortified system over the values considered. It also shows a strong negative correlation between latency and heartbeat interval and a strong positive correlation between throughput and heartbeat interval for the primary-backup system over the values considered.

G.1.2 Migration Interval

The heartbeat interval was fixed at 6 seconds for the proactively fortified system and the migration interval was varied between 20 seconds and 100 seconds in 10 second steps. The results are shown in Figure G.3 and Figure G.4. Confidence intervals at the 95% significance level are shown for the latencies in Table G.6 and the throughputs in Table G.7.

These results appear to show latency decreasing as migration interval increases and throughput increasing as migration interval increases. The correlation coefficients for these results are shown in Table G.8. This shows a weak negative correlation between latency and migration interval, and a strong positive correlation between throughput and migration interval for the range of migration intervals considered.

Figure G.3: Latency as Migration Interval Varies - 6s Heartbeat Interval

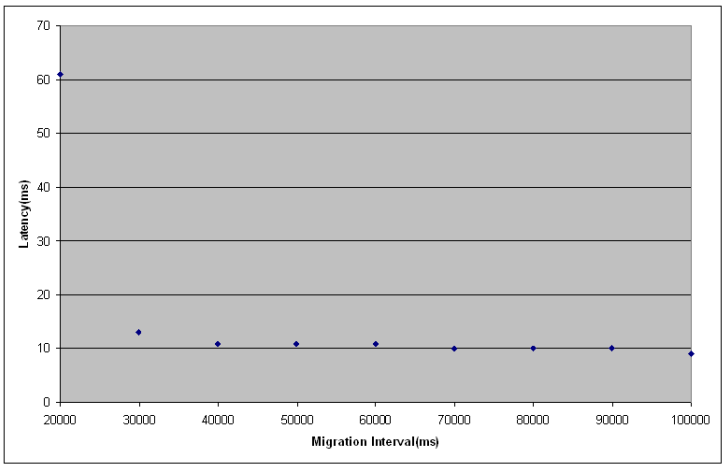


Figure G.4: Throughput as Migration Interval Varies - 6s Heartbeat Interval

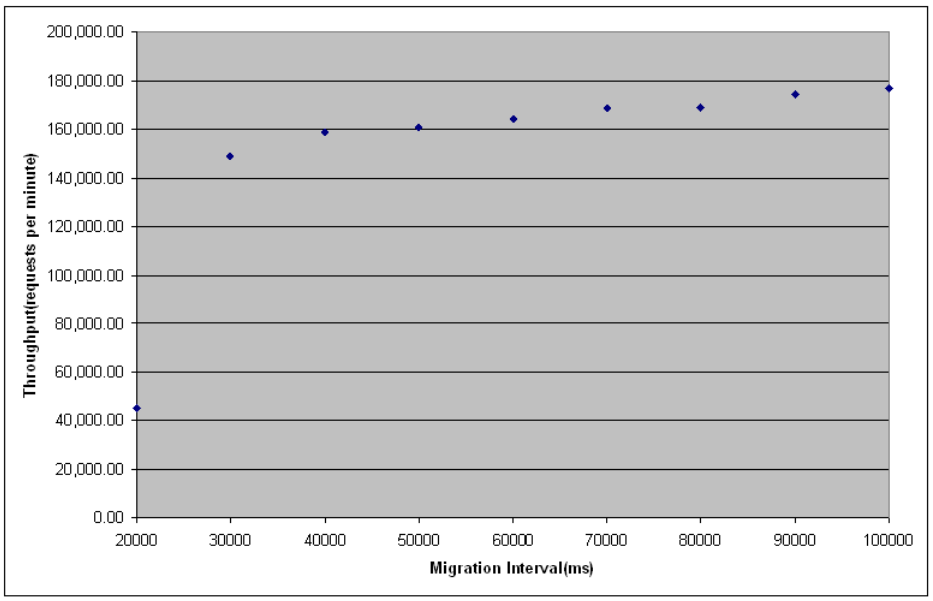


Table G.6: 95% Confidence Intervals for Latencies of the Proactively Fortified System

Migration Interval (ms)	Lower Endpoint(ms)	Upper Endpoint(ms)
20000	60.79	61.21
30000	12.90	13.10
40000	10.91	11.09
50000	10.91	11.09
60000	10.91	11.09
70000	9.92	10.08
80000	9.92	10.08
90000	9.92	10.08
100000	8.92	9.08

Table G.7: 95% Confidence Intervals for Throughputs of the Proactively Fortified System

Migration Interval (ms)	Lower Endpoint(Requests per Minute)	Upper Endpoint(Requests per Minute)
20000	44,014.34	45,643.62
30000	147,344.88	149,846.66
40000	157,490.18	160,020.54
50000	159,457.88	162,006.12
60000	162,838.88	165,417.12
70000	167,150.08	169,569.16
80000	167,866.96	170,369.04
90000	172,942.34	175,429.02
100000	175,654.59	178,229.61

Table G.8: Correlation Coefficients for the Comparison Between Latency and Throughput and Migration Interval

Data Series	Correlation Coefficient
Latency	-0.59671
Throughput	0.705528

G.2 Online Shopping Page

G.2.1 Heartbeat Interval

The migration interval was fixed at 100 seconds for the proactively fortified system and the heartbeat interval was varied between 1 second and 6 seconds in 1 second steps. The results are shown in Figure G.5 and Figure G.6. Confidence intervals at the 95% significance level are shown for the latencies of the proactively fortified system in Table G.9 and the latencies of the primary-backup system in Table G.10. Confidence intervals at the 95% significance level are shown for the throughputs of the proactively fortified system in Table G.11 and the latencies of the primary-backup system in Table G.12.

Table G.9: 95% Confidence Intervals for Latencies of the Proactively Fortified System

Heartbeat Interval(ms)	Lower Endpoint (ms)	Upper Endpoint (ms)
1000	594.46	597.54
2000	582.50	585.50
3000	576.51	579.49
4000	573.51	576.49
5000	555.53	558.47
6000	547.58	550.42

Figure G.5: Latency as Heartbeat Interval Varies - 100s Migration Interval

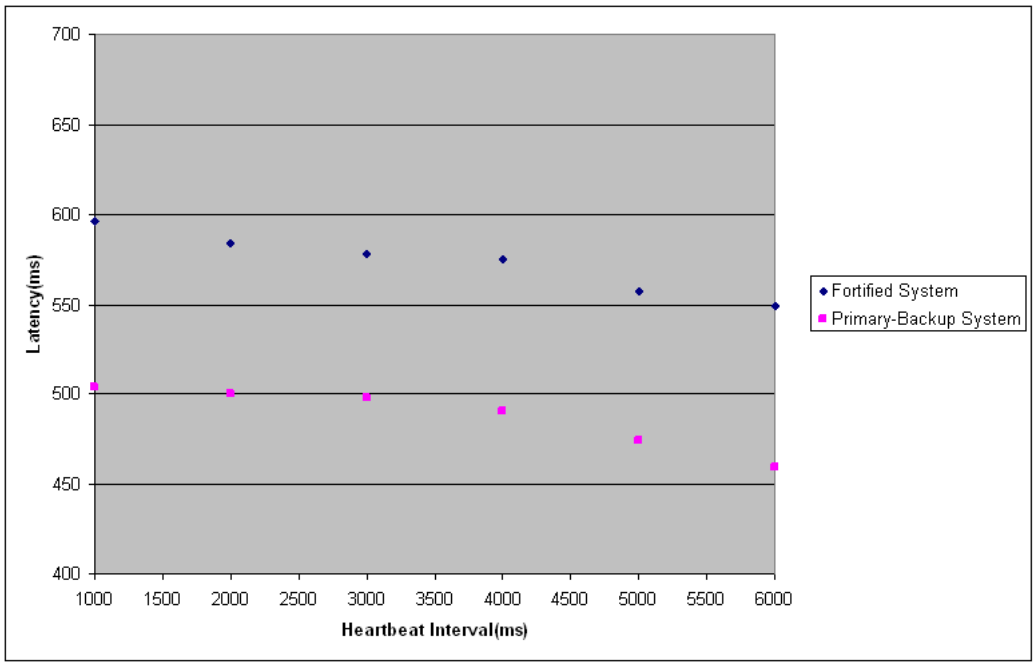


Figure G.6: Throughput as Heartbeat Interval Varies - 100s Migration Interval

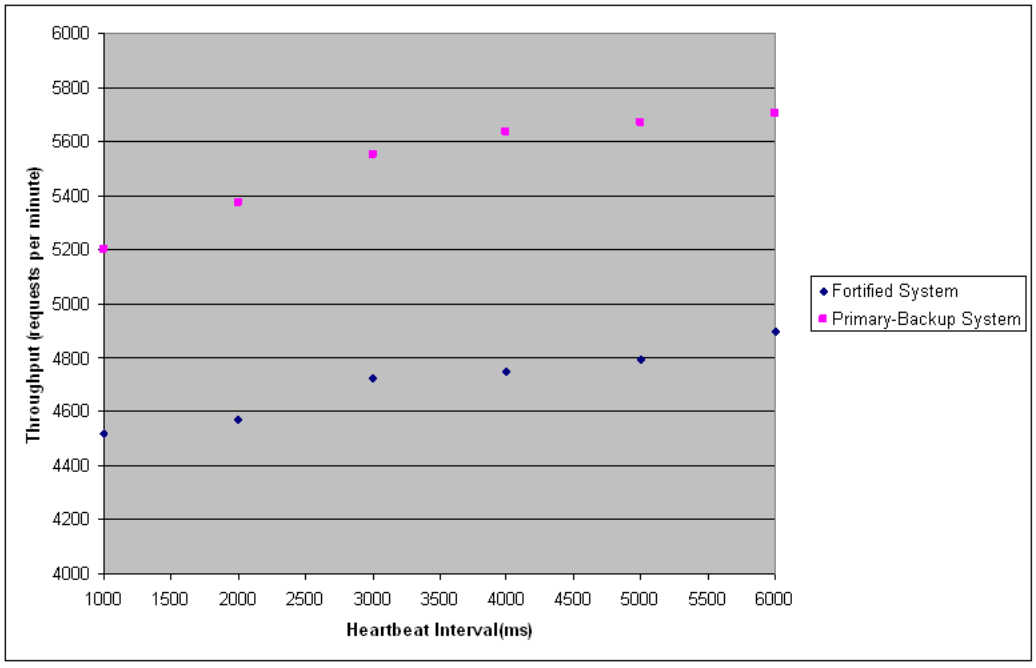


Table G.10: 95% Confidence Intervals for Latencies of the Primary-Backup System

Heartbeat Interval(ms)	Lower Endpoint (ms)	Upper Endpoint (ms)
1000	503.26	504.74
2000	499.24	500.76
3000	497.26	498.74
4000	489.27	490.73
5000	473.27	474.73
6000	458.30	459.70

Table G.11: 95% Confidence Intervals for Throughputs of the Proactively Fortified System

Heartbeat Interval(ms)	Lower Endpoint (Requests per Minute)	Upper Endpoint (Requests per Minute)
1000	4479.99	4556.63
2000	4533.19	4613.21
3000	4686.42	4765.28
4000	4710.57	4788.07
5000	4753.623	4832.37
6000	4856.18	4938.12

Table G.12: 95% Confidence Intervals for Throughputs of the Primary-Backup System

Heartbeat Interval(ms)	Lower Endpoint (Requests per Minute)	Upper Endpoint (Requests per Minute)
1000	5163.16	5243.40
2000	5337.22	5414.72
3000	5511.10	5589.10
4000	5594.34	5673.08
5000	5629.51	5710.51
6000	5664.22	5741.98

This data appears to show latency decreasing for both systems as the heartbeat interval increases. Throughput appears to increase as heartbeat interval increases both for the proactively fortified system and the primary-backup system without proactive fortification. Correlation coefficients are shown for these results in Table G.13. In both cases these correlation coefficients demonstrate a strong negative correlation between latency and heartbeat interval, and a strong positive correlation between throughput and heartbeat interval for the range considered.

Table G.13: Correlation Coefficients for the Comparison Between Latency and Throughput and Heartbeat Interval

Data Series	Correlation Coefficient
Latency for Proactively Fortified System	-0.98082
Throughput for Proactively Fortified System	0.97811
Latency for Primary-Backup System	-0.94737
Throughput for Primary-Backup System	0.947882

Figure G.7: Latency as Migration Interval Varies - 6s Heartbeat Interval

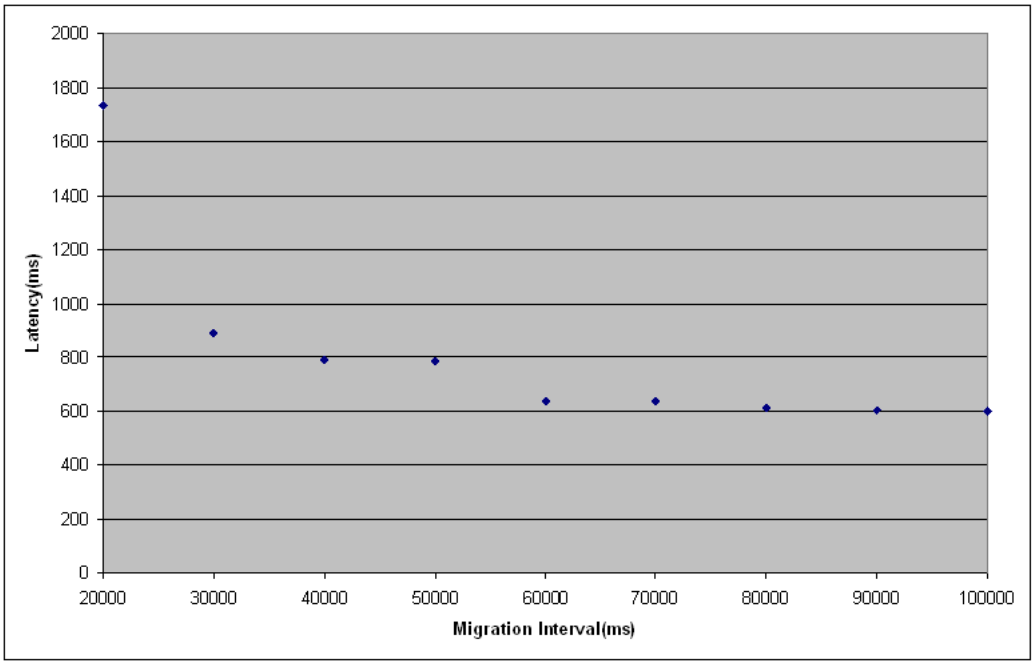


Table G.14: 95% Confidence Intervals for Latencies of the Proactively Fortified System

Migration Interval (ms)	Lower Endpoint(ms)	Upper Endpoint(ms)
20000	1732.39	1735.61
30000	886.42	889.58
40000	788.43	791.57
50000	781.50	784.50
60000	636.73	639.27
70000	636.66	639.34
80000	612.71	615.29
90000	600.64	603.36
100000	547.58	550.42

G.2.2 Migration Interval

The heartbeat interval was fixed at 6 seconds and the migration interval was varied between 20 seconds and 100 seconds in 10 second steps.The results are shown in Figure G.7 and Figure G.8. Confidence intervals at the 95% significance level are shown for the latencies in Table G.14 and the throughputs in Table G.15.

These results appear to show latency decreasing as migration interval increases and throughput increasing as migration interval increases. The correlation coefficients for these results are shown in Table G.16. These correlation coefficients show a strong negative correlation between latency and migration interval, and a strong positive correlation between throughput and migration interval.

Figure G.8: Throughput as Migration Interval Varies - 6s Heartbeat Interval

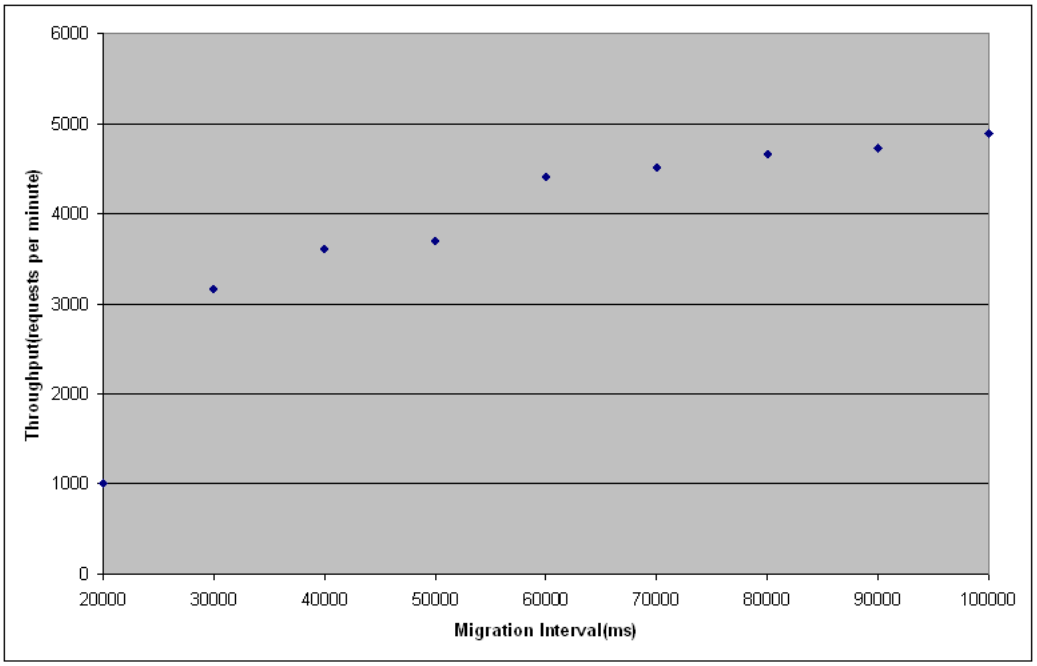


Table G.15: 95% Confidence Intervals for Throughputs of the Proactively Fortified System

Migration Interval (ms)	Lower Endpoint(Requests per Minute)	Upper Endpoint (Requests per Minute)
20000	970.32	1048.30
30000	3120.78	3199.02
40000	3560.51	3638.99
50000	3656.61	3735.35
60000	4358.19	4440.45
70000	4472.10	4550.06
80000	4610.38	4688.80
90000	4685.21	4762.49
100000	4856.18	4938.12

Table G.16: Correlation Coefficients for the Comparison Between Latency and Throughput and Migration Interval

Data Series	Correlation Coefficient
Latency	-0.74545
Throughput	0.8677