FORTRESS: Adding Intrusion-Resilience To Primary-Backup Server Systems

Dylan Clarke
School of Computing Science
Newcastle University, UK
Email: dylan.clarke@ncl.ac.uk

Paul Ezhilchelvan
School of Computing Science
Newcastle University, UK
Email: paul.ezhilchelvan@ncl.ac.uk

Abstract—**Primary-backup replication enables arbitrary services, which need not be built as deterministic state machines, to be reliable against server crashes. Further, when the primary does not crash, the performance can be close to that of an un-replicated, 1-server system and is arguably far better than what state machine replication can offer. These advantages have made primary-backup replication a widely used technique in commercial provisioning of services, even though the technique assumes that residual software bugs in a server system can lead only to crashes and cannot result in state corruption. This assumption cannot hold against an attacker intent on exploiting vulnerabilities and corrupting the service state when attacks lead to intrusions. This paper presents a system, called FORTRESS, which can encapsulate a primary-backup system and safeguard it from being intruded. At its core, FORTRESS applies proactive obfuscation techniques in a manner appropriate to primary-backup replication and deploys proxy servers for additional defence. Gain in intrusion resilience is shown to be substantial when assessed through analytical evaluations and simulations for a range of attacker scenarios. Further, by implementing two web-based applications, the average performance drop is demonstrated to be in the order of tens of milliseconds even when obfuscation intervals are as small as tens of seconds.**

I. Introduction

Literature on fault-tolerance reports two prominent ways of replicating a service. State machine replication [16] (SMR, for short) and primary-backup replication [10] (PB, for short). In SMR, all replicated servers execute every client request and clients accept identical responses from a majority of servers as the correct response. So long as faulty servers are a minority, their failures are masked and hence failure detection and fault replacement can be second order concerns in SMR.

SMR however imposes a requirement [16] that all replicas reach an identical state after processing any given client request. Hence, (i) they must reach consensus [7] on the order in which they process client requests and, (ii) all sources of non-determinism in the software must be removed or their manifestations be handled during run-time. Request ordering adds a constant run-time overhead; enforcing deterministic execution needs to address two major problems [6], [11]: being highly architecture-specific, it requires a comprehensive understanding of the instruction set being executed and the sources of external events; secondly, it results in an unacceptable overhead when applied in multi-processor systems where shared-memory communication between processors must be accurately tracked and propagated.

In PB replication, one of the replicated servers is designated as the primary and the other(s) as backup(s); client requests are processed only by the primary which continually sends state updates to backups; if the primary ever crashes, a backup takes over as the new primary. (See Chapter 8 of [10] for details.)

When the primary is operative, PB performance can be close to that of a 1-server system and the shortfall is due only to the primary having to reliably state-update the backups [9]. Recent investigations [6] affirm that this shortfall can be minimized considerably by performing updates asynchronously without compromising correctness when the primary crashes.

Furthermore, unlike in SMR, PB replication does not require that a service be built as a deterministic state machine. This removes the burden of having to remove or handle sources of non-determinism, in particular, when the service is composed using a legacy software or evolves over its lifetime. These advantages have made PB replication a useful and widely-used technique in a variety of application contexts, including real-time [22] and mission-critical contexts [6].

With the proliferation of web-accessed services, attacks by malicious users remain a growing threat to service reliability. An attacker seeks to exploit a vulnerability in software. When an attack succeeds, an *intrusion* occurs and the intruded server crashes or, in the worst case, operates under the attacker's control. Modern service software is so complex and evolving that removing all vulnerabilities and maintaining vulnerability-freeness are near impossible. This situation has led to the *intrusion tolerance* approach which regards that intrusions can occur despite the preventive measures in place and seeks to maintain reliability even if a few replicas are intruded.

Due to its masking ability, SMR readily progresses into being intrusion-tolerant, provided that the replicas are built to be intrusion-independent: an exploit that succeeds on one replica cannot intrude others. Most intrusion-tolerant systems use SMR, e.g., [3], [4], [12], [17], [19]. PB replication, on the other hand, can tolerate only crash failures and the replicated system, in its entirety, fails or, even worse, gets *compromised* if an intrusion corrupts primary state or enables the attacker to control primary execution, respectively.

In practice, PB systems assume (often implicitly) that the measures put in place prevent attacks from gaining intrusions or block intrusions from corrupting service state. This paper aims to strengthen this assumption by making a PB system resilient to those attacks and intrusions against which conventional preventive measures may not be effective.

An obvious, intrusion-tolerance approach is to swiftly detect and isolate the intruded primary and then initiate a take over as though the primary had "crashed". It poses two obstacles. Perfect and swift detection of intrusions still remains an elusive goal [14]; even if it were feasible, intrusion-independence requires that backups cannot be exact replicas of the primary which complicates the normal-mode, state-update process.

Here, we pursue a different approach that does not warrant perfect intrusion detection or even any structural changes to an existing PB system, yet adds intrusion-resilience capabilities to a significant degree in return for a very small performance drop. Presenting this approach and demonstrating the end results are our main contributions. To the best of our knowledge, this is the first approach to address the problem of making an existing PB system resilient to intrusions when conventional measures cannot prevent an intrusion from corrupting primary state, and to do so without requiring prompt and perfect detection of intrusions.

At the heart of our approach are the recent developments in program obfuscation [15] which make it difficult for an attacker to exploit a common class of vulnerabilities, even if the attacker knows that such a vulnerability is present in a server system; the attacker's difficulty is upheld over time, when obfuscation is repeated periodically. These techniques, collectively called *proactive obfuscation* [13], are suitably adapted to preserve the useful features of a PB system.

The paper presents, in Section 2, a class of vulnerabilities found quite commonly in distributed systems, proactive obfuscation, and the techniques that can be used to circumvent obfuscation. Section 3 presents the architecture of the FORTRESS system. Section 4 analyzes the improvement in intrusion resilience achieved through the use of FORTRESS. Section 5 experimentally evaluates the performance overhead caused by FORTRESS. Finally, section 6 concludes the paper.

II. Background

First, we discuss UCIT vulnerabilities that can be exploited to inject and execute an arbitrary piece of code on target systems. Then we examine proactive obfuscation and, finally, discuss distributed attacks and the implications these attacks have on our modeling of intrusion attempts.

A. UCIT Vulnerabilities

Unauthorized Control Information Tampering vulnerabilities (UCIT vulnerabilities) were first categorised in [5]. They are vulnerabilities, such as unchecked buffers, that make it possible for maliciously crafted messages to overflow control structures such as the stack or the heap. This allows an attacker to inject his own malicious code into a target system and change control information so that his code gets executed.

Exploitation of these vulnerabilities to cause an injected piece of code to be executed requires the attacker to have a considerable knowledge of the internal layout and details of the target system. This can be achieved, as noted in [5], by running a local system with identical hardware and software.

B. Proactive Obfuscation

Proactive obfuscation combines two techniques: program obfuscation and periodic re-obfuscation. An obfuscator (see [2], [13], [15]) takes two inputs, a program P and a secret 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 if P' is used (in stead of P).

Program obfuscation is essentially randomization of executables obtained through a variety of techniques, such as address space layout randomization (ASLR) [21] and instruction set randomization (ISR) [8], or a combination of them. The number of keys available while using a given technique is implementation- and system-dependent but is typically quite large; say, for ASLR, it can be up to 2^{16} for 32 bit systems and between 2^{32} and 2^{40} for 64 bit systems. Since k is chosen from a large space, the probability of guessing it out-right is negligible, e.g., less than 0.1% if the key space is 2^{10}.

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 key. Consequently, if the SMR system is designed to tolerate at most $f, f > 1$ intruded replicas, then at least $(f+1)$ of the keys used must be known to the attacker for exploiting a vulnerability in P. 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 derandomization attacks (discussed in § II-C). To spoil this advantage that an attacker has, 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 re-obfuscation period is referred to as *unit time-step* or *migration* interval.

We will make three important observations on the re-obfuscation process. First, it involves several activities, all of which must be carried out in a secure manner. Secondly, a given key may be selected for use, through the random selection process, several times over the system lifetime. However, each selection allows it to be used *continuously* for only one unit time-step; at the end of that time-step, a used key emigrates out of the SMR system and does not become eligible for selection until another key is chosen. Finally, since the number of replicas deployed within practical SMR systems is considerably smaller than the key-space, forbidding the emigrating keys from being candidates for just one selection would still leave a large enough space to select from.

The infrastructure mechanisms needed for supporting proactive obfuscation make use of several trusted components and are detailed in [13]. Two trial implementations in [13] show that proactive obfuscation is a practical technique. Similar infrastructure has also been implemented by [3], [4], [19] without an explicit use of program obfuscation.

Finally, we note that all replicas of an SMR system need not be re-obfuscated at the same time after a unit time-step; they can be, and are in [13], re-obfuscated at different timing instances while ensuring that at least $(2f+1)$ replicas are servicing client requests at any given time. Thus, periodic re-obfuscation can be done without affecting SMR performance. This feature cannot be preserved in a primary-backup system as only the primary is doing the computation.

C. Derandomization Attacks

The rationale behind the randomization techniques, such as ASLR and ISR, is that if the attacker builds his exploit using a key that is different from the one used on a given replica, he cannot inject his malicious code at the correct location within that replica's address space (when ASLR is used) nor can he build his code with correctly randomized instructions (when ISR is used). Consequently, the attack merely causes the replica to crash but can never lead to an intrusion.

Note that an attacker choosing a wrong key for his exploit causes a replica to crash; conversely, if he learns that his attack has caused a replica to crash, he can deduce that the key he chose for his exploit is not the same as the one used in that replica. This aspect and the fact that the key space is publicly known form the basis for derandomization attacks in [18] and [20] which deduce the key used in a reasonable number of trials, when ASLR and ISR are used respectively. The derandomization process pursued in [18], [20] is basically one of elimination: choose a key to construct an exploit, eliminate it from the search space if the target is observed to crash, and thus continue to narrow down the search space until the key chosen leads to the injected code being executed. (In [20], the right key needs to be discovered in parts.)

Though the experiments of [18] and [20] were directed at ASLR and ISR, derandomisation attacks could, in principle, be extended to any form of randomisation, as the underlying rationale behind any form of program obfuscation is the same.

The following observation by [18] has a profound implication on the choice of unit time-step duration for SMR: the derandomisation attacks of [18] succeeded after a good number of attempts, even if the target was re-obfuscated every time after it crashed. So, if re-obfuscation of SMR replicas is done infrequently, say, to reduce the overhead, an attacker could manage to have the keys of (at least) $(f+1)$ replicas deduced at any given time; in that case, he could have more than f replicas intruded *at the same time*. This would violate the basic SMR assumption; hence, during re-obfuscation, the re-booted replicas are not guaranteed to be initialized with the correct service state and, subsequently, the SMR system could fail, i.e., the *system compromise* could occur.

Choice of unit time-step duration thus influences the *system lifetime* which is defined as the time elapsed between the deployment of an intrusion-tolerant system and the instance when the number of simultaneously intruded replicas exceed the tolerance threshold. Smaller unit time-step durations can result in longer lifetimes; they do increase the number of re-obfuscations carried out per hour of lifetime and hence they can increase the ratio of re-obfuscation overheads to lifetime.

One way to increase the unit time-step duration without risking a reduction in lifetime is to reduce the possibility of derandomization attacks leading to simultaneous intrusions exceeding the tolerance threshold. This can be accomplished by placing proxy servers between replicas and clients. These proxies act as intermediaries by forwarding client requests to replicas and responses to clients. They thus prevent an attacker from monitoring the response times over direct TCP connections to the replicas and thereby remove the most direct way to discover if a guess on the key has caused a crash or not. (So, proxies are used in FORTRESS - see § III.)

D. Distributed attacks

Distributed attacks involve an attacker using multiple machines, e.g., by using bot-nets [1], to launch attacks on a target system (which in our case is a replicated server system). This means that an attacker will generally be able to launch as many attacks as he wishes, and hence will be able to attack *every* publicly accessible replica or proxy (if used) in the system at the same rate, regardless of the number of these replicas/proxies present. That is, the attack rate on publicly accessible replicas/proxies can be arbitrarily high.

We note that distributed attacks can be divided into two types, regarding the ultimate purpose they are launched for: those aiming to attack the availability of a system (denial of service, DoS, attacks) and those aiming to compromise service integrity by corrupting the service state and/or breach confidentiality by gaining access to the information stored in the system. The techniques considered in this work are designed specifically against the attacks of the latter type; other techniques will be required to safeguard against DoS attacks.

When the attacker's aim is only to breach the integrity or confidentiality of a target system, it is in his interest not to flood the system to the point that it shuts down. His goal would be to keep the system available so that integrity and confidentiality could be breached by compromising a sufficient number of replicas.

This means that, even for a distributed attacker, there exists a finite, upper bound on the number of attacks launched on a replicated server system in a given period of time. In practice, this bound is likely to be set either by the number of requests the system can process, or the number of malicious requests that can be safely sent without raising suspicion among system administrators. (Note: attack detection and monitoring measures do have a significant role to play here in raising alerts; on the other hand, SMR and FORTRESS do not have to rely on an intrusion being detected when an undetected attack leads to an intrusion; While SMR tolerates intrusions, FORTRESS makes it harder for an attack to cause intrusions.)

Throughout the paper, we assume that a server can be subject to at most β attacks in any unit time-step. In modeling terms, existence of β means that the probability of an attacker deducing the key used in any given replica within the unit time-step duration is less than 1 if the number of available keys is larger than β.

III. System Architecture

The FORTRESS approach involves two key techniques. The first of these is to introduce a new tier of nodes, which we call the *proxy tier*, in front of the server or application tier in two or three tier systems, respectively. A 3-tier system comprises web, application, and data tiers, and the latter two form a single, server tier in a 2-tier system. In what follows, the tier right next to the proxy tier would be referred to as the server tier that houses server replicas. Proxy nodes forward the (signed) client requests to all nodes in the server tier and the (signed) responses from the primary back to the client. Figure 1 illustrates these flows and, for clarity, only one proxy node is shown to forward client requests.

The server tier is configured to accept requests only from the proxy nodes, preventing an attacker from *directly* communicating with the server nodes, resulting in a series of difficulties for the attacker as discussed in sub-section II-C (and re-visited in § IV-C). The proxy tier offers, like ramparts, a defensive platform against attacks being launched directly on the server nodes (hence the name FORTRESS).

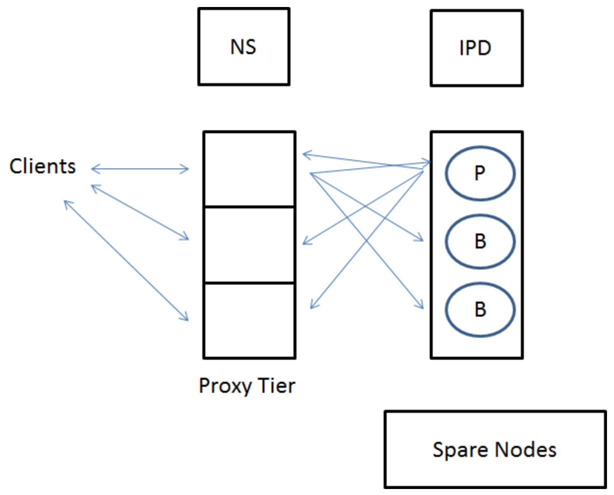
The second technique used is proactive obfuscation which is implemented using the infrastructure for proactive obfuscation (IPO) of [13] but adapted for primary-backup systems. The adaptation requires additional infrastructure components, namely a name server (NS) and a pool of spare nodes (see Figure 1), whose roles will be discussed shortly. In FORTRESS, the nodes of both the proxy and the server tiers are obfuscated at the start and re-obfuscated periodically thereafter.

As noted earlier (in § II-B), replicas within an SMR system can be re-obfuscated at distinct timing instances. Pursuing the same approach for a primary-backup system introduces two sources of overhead, as we discuss below.

Suppose that the nodes of the primary-backup server tier are re-obfuscated at distinct instances. Let these nodes be A, B and C, with A currently acting as the primary. Just before A's re-obfuscation is due, say at t_A, A must relinquish its primary role, say, to B. That is, re-obfuscation of A invokes a fail-over which any primary-backup system must be able to handle. (In a normal system, fail-over occurs whenever the primary crashes and handling an obfuscation-induced fail-over is relatively easier since it is an anticipated event.) B must be due for its re-obfuscation sometime at or before $t_A + \Delta$, where Δ is the unit time-step duration. So, before $t_A + \Delta$, B must transfer the primary role to A or C. Thus, handling at least one obfuscation-induced fail-over for every Δ is inevitable. A fail-over slows down the system responsiveness; so, the number of fail-overs handled per Δ must be kept small and is exactly one in FORTRESS.

Secondly, when, say, A is acting as the primary, it must continually provide state updates to B and C. (An aspect of primary-backup replication for crash-tolerance.) Since the nodes are obfuscated at different instances, their keys would be different; so, A must marshall the state-updates (to a

Figure 1: FORTRESS Architecture



standard form), and B and C must unmarshall the updates received to suit their respective randomization. To avoid this constant, running overhead of marshalling/unmarshalling of updates (not present in a normal primary-backup system), FORTRESS makes the following design decisions.

1) all server and proxy nodes are re-obfuscated at the same timing instance; hence, the start and end times of each unit time-step are identical for all nodes;
2) all server nodes are re-obfuscated with the same, randomly-selected key; and
3) each proxy node is re-obfuscated with a distinct, randomly-selected key (that is not used in any other node in the system at the same time).

Note that the keys used for a given unit time-step are not candidates for selecting keys for the next unit time-step. FORTRESS speeds up the re-obfuscation process by obfuscating spare nodes as server/proxy nodes during a unit time-step and replacing the existing ones with the freshly obfuscated ones at the end of that unit time-step. Thus, re-obfuscation becomes node *replacement* and *state-transfer* whereby incoming server nodes receive the service state from the ones being replaced. (This state transfer requires marshalling and unmarshalling.)

Thus, the IPO performs the periodic replacement of proxy and server nodes (at the end of each unit time-step), the shutting down and re-booting of replaced nodes, the updating of the name server (NS) with the addresses of the new proxies, the updating of the proxies with addresses of the current server nodes, and obfuscating spare nodes for the next replacement. We refer the reader to [13], [19] to see that the IPO could securely accomplish all of the above. We simply note here that the IPO of [13], [19] makes use of several trusted components, synchronized clocks and timely communication links for the purposes of security and timeliness in its operations.

The Proxy Tier. It consists of nodes running simple proxy software that can take a client request and pass it to the nodes in the server tier. Similarly, each proxy can take a response from the server tier and pass it back to the client that made the original request. Each node in the proxy tier runs a uniquely randomized diverse executable. It is publicly accessible, and hence open to attacks from anyone with an Internet connection. Throughout this paper we assume that the proxy tier has 3 proxies.

The Server Tier. It hosts a primary-backup system; throughout this paper, we assume two backups in addition to a primary. These nodes are identically randomized to reduce state transfer overhead, as the compromise of the primary will result in system compromise even when the backups are not compromised. Nodes in the server tier are not directly accessible to clients. They only accept requests from nodes in the proxy tier, update messages from other server nodes where appropriate, and control messages from the IPO.

A. *Replacement and State Transfer*

After a set period of time the current proxy and server tiers will be replaced by nodes from the spare pool 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 duration in which requests are not being processed should be minimal.

Requirements 1-3 are provided by the IPO. Requirements 4 and 5 are provided by the NS, with the IPO ensuring that the NS has the correct information to satisfy requirement 5. We use spare nodes to achieve requirement 6. The exact overhead is measured in section V.

IV. Intrusion Resilience in FORTRESS

A. *Modelling Node Intrusion*

An attack against a node with a UCIT vulnerability will succeed when a malicious request sent to the node fulfils two criteria:

1) The malicious request is appropriately constructed to exploit that vulnerability if no randomisation is in place.
2) The randomisation key chosen for the malicious request is the same as the randomisation key used to produce the diverse executable present on the node.

We assume that the attacker is sufficiently skilled that all malicious requests fulfil criterion 1, and that the node is intruded into as soon as the derandomisation attack is successful. If these assumptions do not hold then the attacker would not be able to compromise the system at all. This leads to node intrusion being entirely dependent on meeting criterion 2.

Hence, the probability of a node being intruded during a given unit time-step is the probability that one of the malicious requests processed during that unit time-step had the correct randomization key. This probability depends on the the number of malicious requests β that can be processed during a unit time-step and on the total number γ of keys available for obfuscation.

The presence of these variables leads us to model the probability of a given node being successfully intruded into in a given unit time-step as $0 < \alpha \leq 1$. We assume that α is identical for all nodes that are directly accessible by an attacker. When a server node is accessible only via proxy nodes and all proxies are correct, we consider indirect attacks on servers which will be discussed in Section IV-C. We note that α is constant when proactive obfuscation is used and is time dependent when it is not. This time-dependence arises if an attacker can safely discount some keys an ineffective simply because they were not effective in an earlier attempt.

In our comparative evaluation we consider a wide range of α values, in an attempt to cover all likely values. There is of course the possibility that, for a given combination of system and attacker, α could be outside the parameters considered. These outliers represent attackers who are either so powerful that they manage to achieve β so close to γ that no effective defence can be provided, or so ineffective that no additinal defence needs to be provided.

B. *System Models*

1) PBSO: The *PBSO* system is a primary-backup system with start-up only obfuscation. That is, primary and backups are initially obfuscated and are not re-obfuscated thereafter. The architecture shown in Figure 1 would therefore consist only of the server tier and nothing else. PBSO represents the typical deployment of commercial primary-backup systems.

A primary-backup system is compromised when the primary is intruded. The primary is the only node processing client requests, so we model PBSO as one node, with the system compromise occurring when that node is intruded. Hence, the probability of system compromise in one unit time-step is simply the probability of one node being intruded in one unit time-step.

As no re-obfuscation takes place, the attacker can safely eliminate some keys as ineffective after every unit time-step, so the intrusion probability (and hence the system compromise probability) increases with time. We define the probability of system compromise in unit time-step i to be α_i and set $\alpha_0 = \frac{\beta}{\gamma}$ as none of the γ keys would be known to be ineffective before the first unit time-step.

The i^{th} unit time-step, $i \geq 1$, begins with $i\beta$ keys having been found ineffective, and hence the probability of system compromise during unit time-step i is $\alpha_i = \frac{\beta}{\gamma - i\beta} = \frac{\alpha_0}{1 - i\alpha_0}$ unless fewer than β keys are left to try during step i, in which case the probability of compromise is $\alpha_i = 1$.

Hence, the probability α_i of system compromise during a unit time-step is: $\frac{\alpha_0}{1-i\alpha_0}$ if $1 - i\alpha_0 > \alpha_0$, or 1 otherwise.

2) PBPO: The *PBPO* system is a primary-backup system with proactive obfuscation. This involves the primary and backups being replaced at the end of each unit time-step in such a way that a new diverse executable is introduced. For PBPO, the system architecture in Figure 1 would have everything except the proxy tier.

The probability of system compromise in one unit time-step is the probability of one node being intruded in one unit time-step, as with the PBSO system. However, unlike PBSO, every unit time-step starts with re-obfuscation; so the probability α of system compromise during *any* unit time-step is $\frac{\beta}{\gamma} = \alpha_0$.

3) FORTRESS: A FORTRESS system is compromised when either all of the nodes in the proxy tier are intruded or the primary is intruded. The primary is the only node processing client requests, and can be *directly* attacked only via an intruded proxy. When a proxy is intruded, it can be used by the attacker to launch attacks directly against the server nodes. Therefore, the probability of the primary being intruded into by a direct attack in a unit time step is α if one or more proxies was already intruded into at the start of that unit time-step and 0 if no proxies were already intruded into at the start of that unit time-step

Since the attacker can launch distributed attacks, we assyme that each proxy can be subject to at most β attacks per unit time-step without raising alarms. Hence, the probability of a proxy being intruded into in a unit time-step is taken to be α as well. In what follows, we consider the probability of primary intrusion by indirect attacks, i.e. where no proxy is intruded.

C. Indirect Attacks

We define an indirect attack to be an attempt to intrude into a server with a malicious request launched from a client, and passed to the server via an unintruded proxy. This is in contrast to a direct attack, which would involve a malicious request sent directly from the source of the attack to the target.

At first glance, indirect attacks may appear as likely to succeed as direct attacks. Proxy servers are merely passing on the malicious request, and the server is executing it as if it had been sent directly. However, a closer examination shows the following impediments to indirect attacks.

Firstly, the proxy may have the same vulnerability as the server. For example, an overly long URL string sent to a proxy with the intention of overflowing the buffer in the server it will be passed to, may instead overflow a buffer in the proxy. In this case an indirect attack on the server is impossible and the attacker must instead first use the buffer overflow to successfully intrude into the proxy.

Secondly, the proxy may have defences against vulnerabilities that prevent it from passing on a malicious request to the server. For example, if the proxy defends against attacks involving an overly long URL string by checking the length of the URL string and refusing or truncating overly long requests, then an indirect attack on the server will be impossible, as the original malicious request will never be passed to the server.

Finally, even if the malicious request is successfully passed to the server, the presence of a proxy may reduce the opportunity for the attacker to get the feedback needed to detect a successful attack. Attacks against randomized systems such as those in [18] typically rely on using many malicious requests to perform a brute force attack against the randomization key. Each of these requests contains a very small piece of malicious code, designed to produce a response from the server that will be measurably different if the correct randomization key was chosen than if it was not.

While the presence of a proxy between the attacker and the server does not generally prevent communication from the server to the attacker (a compromised server could for example, use a subliminal channel) it does have the potential to increase the time required for the attacker to become aware of whether a specific malicious request has succeeded or failed, and the length of code needed for the attack.

D. Modelling Indirect Attacks

When modelling the FORTRESS system, we note that there are two possibilities for system compromise. Either one or more proxies are compromised, and hence direct attacks are launched at the primary, or all proxies are correct and hence only indirect attacks are launched at the primary. So, the definition of the probability of the primary being compromised in a given unit time-step becomes:

1) If a proxy node is compromised by the start of the unit time-step then we allow the attacker to attack the primary with probability α of compromising it during this unit time-step. This is a direct attack as it is generated by the compromised proxy node and sent directly to the server node.
2) If no proxy node is compromised by the start of the unit time-step then we allow the attacker to attack the primary with probability $\kappa\alpha$ of compromising it during this unit time-step. This is an indirect attack as it is generated by the client and sent to the server node via an unintruded proxy node.

We define κ to be the *indirect attack coefficient*, such that $0 \leq \kappa \leq 1$. This models the extra difficulty present in attacking a node indirectly. When $\kappa = 0$ indirect attacks are not possible and when $\kappa = 1$ they are as effective as direct attacks.

E. Expected Lifetime Evaluation

Now that we have developed models that specify the probability of system compromise in each unit time-step, we can use these models to determine the expected lifetime (EL) until system compromise. We determine the expected lifetimes using Markov Chain techniques for PBPO systems and FORTRESS systems with infinite diversity, Time-dependent Stochastic Process techniques for PBSO systems, and Monte-Carlo methods for FORTRESS systems with finite diversity.

1) Finite vs Infinite Diversity: There are two classes of diversity we consider when comparing systems, finite and infinite. In both cases the number of executables created is finite as the size of the randomization key is finite; infinite diversity is only infinite from the point of view of an attacker.

In the finite class, we assume that once a given executable is compromised, the attacker will be able to re-compromise that executable again in a negligible amount of time. The attacker is assumed to have gained some information from compromising the executable that allows him to identify it when it is encountered in the future, and launch a rapid attack that is guaranteed to compromise the executable.

In the infinite class however, either it is not possible for the attacker to identify a previously compromised executable, or to easily re-compromise it. Hence, from the attacker's point of view, every executable he encounters is a new one and this is modelled as if the pool of diverse executables is infinite.

F. Results

All systems were considered across the intrusion probability range $\alpha = 0.00001$ to $\alpha = 0.1$.

We note that the amount of diversity available does not directly affect the expected lifetime of a primary-backup system, even when proactive obfuscation is used. This is due to the fact that the primary-backup system is compromised as soon as the primary is compromised, so the entire system becomes compromised at the same time as the first executable becomes compromised.

The amount of diversity available does indirectly affect the expected lifetime, as, in practice, smaller α values are only likely to be available with larger amounts of possible diversity.

We begin by comparing the PBSO system to the PBPO system. A comparison of the expected lifetimes of these two systems is shown in Figure 2. This demonstrates that proactive obfuscation results in almost a 2-fold increase in expected lifetime for primary-backup systems, a relationship that holds for all α values considered.

We next consider the FORTRESS system when indirect attacks are not possible.

A comparison between the PBPO system and the FORTRESS system with diversity of 2^{16} over the range $\alpha = 0.0001$ to $\alpha = 0.01$ is presented in Figure 3. (In the bar chart of Figure 3, the expected lifetimes of the PBBO and FORTRESS systems are shown from left to right in the stated order.) Figure 3 shows that the FORTRESS system considerably outperforms the PBPO system for this diversity level, for all values of α.

A comparison between the PBPO system and the FORTRESS system when the intrusion probability is fixed and diversity is allowed to vary is shown in Figure 4. This shows that the PBPO system slightly outperforms the FORTRESS system with 2^3 diversity, and is outperformed by the FORTRESS system with all other levels of diversity.

We then consider the situation when indirect attacks are possible. A comparison between the PBPO system and the FORTRESS system with diversity of 2^{16} as the indirect attack coefficient varies is presented in Figure 5. This shows that the FORTRESS system outperforms the PBPO system except when κ is close to 1. We note that, even when $\kappa = 1$, the FORTRESS system outperforms the PBSO system.

Figure 2: Comparison of PBPO and PBSO as α Varies

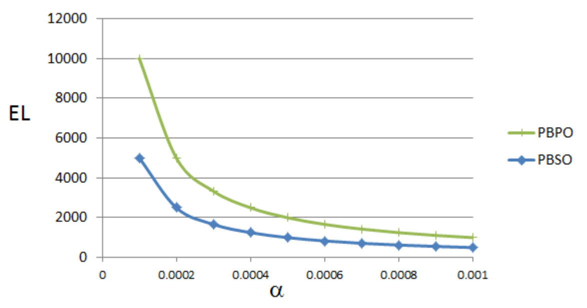
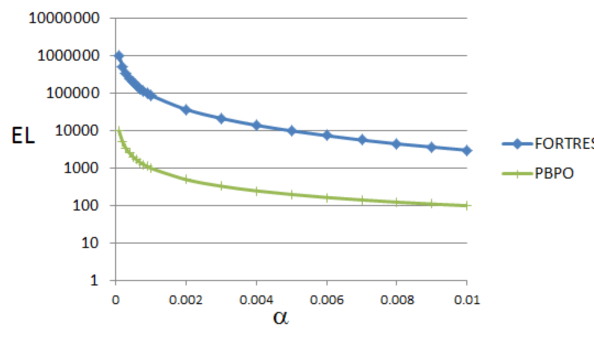


Figure 3: Comparison of PBPO and FORTRESS with 2^{16} Diversity as α Varies



V. Evaluation of Performance Overhead

We considered two types of applications when evaluating the performance overhead cost. The first of these was a relatively small scale web application allowing clients to request a service and check on the progress of that service. The second system considered was a large scale web application using web server and application server layers for load balancing.

In each case we constructed a PBSO system and a FORTRESS system and compared the latency and throughput, and derived an estimate for the upper bound on time taken for migration in the FORTRESS system from the latencies

Figure 4: Comparison of PBPO and FORTRESS with $\alpha = 0.0001$ as Diversity Varies

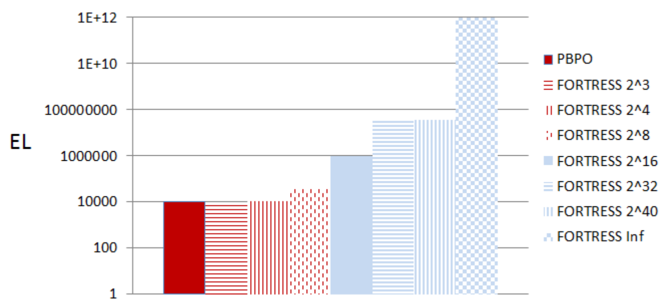
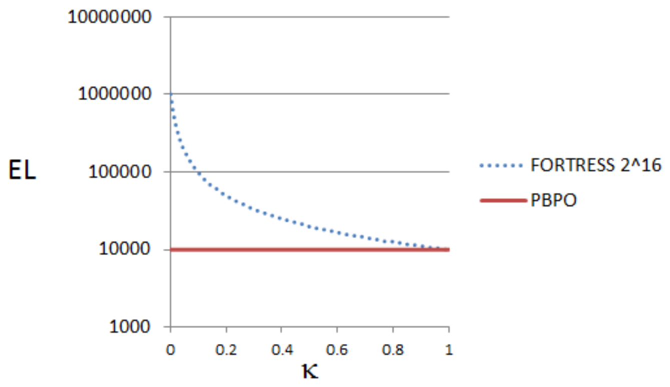


Figure 5: Comparison of PBPO and FORTRESS with $\alpha = 0.0001$, 2^{16} Diversity as κ Varies



measured. (Since the PBSO system was not replaced there was no migration overhead).

Latency and throughput metrics were measured for the FORTRESS system over a range of migration interval values. We note that migration interval, the time between node replacements, is the same time interval that was termed the unit time-step when evaluating expected lifetime until system compromise.

The overhead metrics are calculated as follows:

$$\text{Increase in Latency} = \frac{\text{ML of FORTRESS - ML of PBSO}}{\text{ML of PBSO}}$$

$$\text{Decrease in Throughput} = \frac{\text{T of PBSO - T of FORTRESS}}{\text{T of PBSO}}$$

where ML and T are mean latency and throughput respectively.

A. Small Scale System

The system consists of a simple Java EE application that would take requests for jobs to be added and requests for the status of a current job. These jobs were stored in a database co-located with the application. This application was then expanded to enable database updates to be propagated to backup nodes as a part of the primary-backup system.

1) System Specification: A FORTRESS framework consisting of four components was implemented in Java EE. The first of these components was an application wrapper to handle state transfer between incoming and outgoing server nodes. The second component was a proxy to pass requests and responses between clients and servers. The third component was a name server that provided the IP addresses of the current servers. The fourth component was a controller unit that sent messages to the other components at appropriate intervals.

2) Testing Strategy: Testing was performed using 21 host machines on a local area network. The first 20 of these hosts were used to run 4 sets of 3 servers, 2 sets of 3 proxies, 1 controller unit and 1 name server. The remaining host was used to run 5 instances of the client code.

Figure 6: Percentage Increase in Latency as Migration Interval Varies

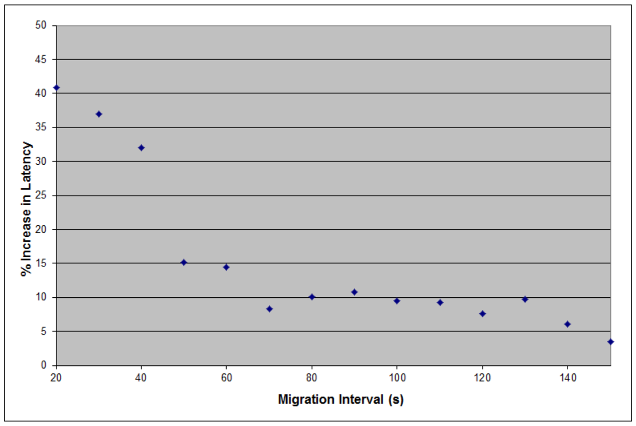
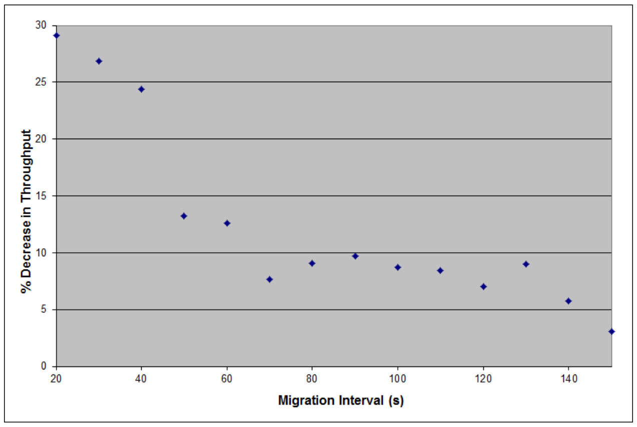


Figure 7: Percentage Decrease in Throughput as Migration Interval Varies



Each instance of the client made a series of 100000 requests, alternating between add job requests and get status requests. 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. We note that each stored job request contained 38 bytes of data, so, as half of the requests sent were add job requests, the system state being transferred increased linearly until it was 9.5MB.

This testing was then repeated using one set of three servers in a primary backup configuration, with the same number of client instances and requests.

3) Results: The contents of the databases on the active primary and backup servers were found to be correct after all tests. The increase in latency and decrease in throughput as migration interval varies are shown in Figures 6 and 7.

The efficiency measurement shows us that, for a migration interval of 70 seconds or greater it is possible to achieve a decrease in throughput of less than 10%, and an increase in latency of less than 15%. 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 implications of these overheads are very much dependent on the way in which the systems to be 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 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 a performance decrease.

The maximum latency recorded for FORTRESS was 396ms (recorded when the migration interval was 90 seconds); maximum latencies for other migration intervals did not differ by more than 28ms. The maximum latency recorded for PBSO was 287ms. The minimum latency recorded for FORTRESS and PBSO were 49ms and 44ms respectively. Thus, the time taken for re-obfuscation can be estimated to have never exceeded $396 - 287 < 110$ms.

Decrease in throughput is more likely to be a concerning factor in commercial systems, rather than the small increase is mean latency or even delays of up to 110ms occurring during re-obfuscation. The smaller the migration interval, the more difficult it is for an attacker to gain an intrusion and the larger the overhead. Thus, a trade-off between overhead and intrusion resilience is necessary. Further, using hardware resources powerful enough to allow the system to handle peak demand with a sufficiently small migration interval helps to achieve a higher degree of intrusion resilience.

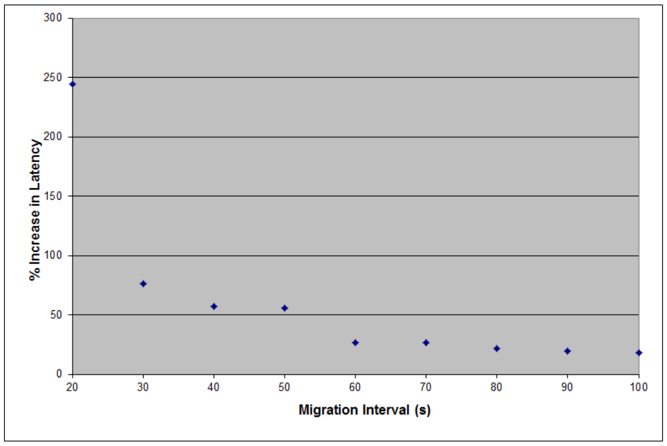
B. Large Scale System

We began by building a simple web application implementing the functionality of an online store. This application consisted of a web server that received client requests and forwarded any request for dynamic content to an application server. The application server then generated the dynamic content, making use of a database, and returned it to the web server. The web server added static content and then returned this to the client. We then replicated this application at the web and application server levels. The application servers became the servers in our FORTRESS system, and the web servers became the proxies. The database was unreplicated.

1) System Specification: Each server node ran two pieces of software; an Apache Tomcat server that executed the web application, and an application wrapper that could start and stop the Tomcat server when it received a message from the controller unit. A shared database was used by the Tomcat servers to retrieve product information.

All state transfer was handled through Tomcat's clustering functionality. The current primary, current backups and the replacement nodes for the next unit time-step were all part of the same cluster. Then, when migration was required, the primary and backups were stopped, and three new nodes were brought into the cluster. The cluster was set up in such a way that processing would always be performed by the primary unless it was necessary to failover to the first backup.

Figure 8: Percentage Increase in Latency as Migration Interval Varies



Each proxy node ran two pieces of software; an Apache HTTP server and a proxy handler that could start and stop the HTTP server when it received a message from the controller unit. The HTTP server was configured to pass any requests it received to the cluster, and return any responses to the client.

The controller unit was implemented as a Java application which kept a list of the proxy and server nodes that were available or in use. It periodically instructed the current proxies and servers to stop, and new proxies and servers to start.

2) Testing Strategy: Testing was performed using 21 host machines on a local area network, configured in the same way as in section V-A2.

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. 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 first page and 20000 requests from each of 50 clients running simultaneously for the second page.

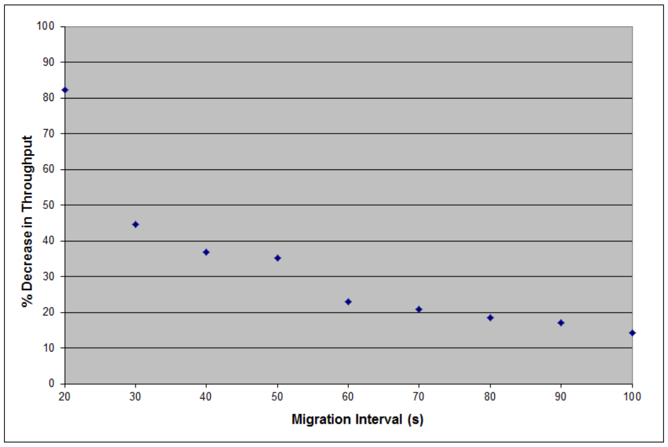
After each test, the system was allowed to run until 20 migrations had occurred and the session variables were checked to determine whether system state had been preserved.

3) Results: In both the experiments, session variables were found to be correct after all migrations. The increase in latency and decrease in throughput are shown against migration intervals in Figures 8 and 9. These results show that encapsulating an online shopping system within FORTRESS does not necessarily lead to a large performance overhead.

We observe in Figures 8 and 9 that increase in mean latency and decrease in throughput caused by FORTRESS are less than 25% in large applications when migration intervals are 60s or mode. Below 60s, mean latency is noticeably increased.

When migration intervals are small, increase in mean latency is due to the fact that a larger percentage of time is spent in transferring state to newly obfuscated nodes, resulting in more requests being delayed from being processed due to

Figure 9: Percentage Decrease in Throughput as Migration Interval Varies



state transfer. One possibility for reducing the mean latency in a production system is to offset the migration periods of several different FOTRESSed application servers, and use load balancing hardware (which is usually present in large scale distributed systems) to allocate the majority of requests to nodes that are currently not engaged in state transfer.

We note that small migration periods are only needed when an attacker must be presented with small windows of time in which to attempt to compromise application servers. Hence small migration periods are only needed when presented with an attacker who has a relatively strong ability to compromise application servers. These are just the circumstances when it may be worth the cost of providing additional servers to cope with the overhead of frequently migrating servers.

VI. Conclusion

We have presented FORTRESS, an intrusion-resilient primary-backup system, that makes use of proactive obfuscation and a proxy tier for reducing the effectiveness of derandomization attacks. The increases in intrusion resilience provided by this system relative to a normal primary-backup system have been evaluated and shown to be significant. The performance overhead of the FORTRESS system has been evaluated experimentally and found to be reasonably low even for a large web-based application, such as an online store, except when intervals of proactive obfuscation become smaller than a minute.

Acknowledgment

The authors would like to thank the UK EPSRC for funding this research through Dylan Clarke's PhD scholarship, and the School of Computing Science, Newcastle University for funding Dylan Clarke's attendance at SRDS 2012 to present this paper.

References

[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] Kenneth P. Birman and Fred B. Schneider. The monoculture risk put into context. *IEEE Security and Privacy*, 7(1):14–17, 2009.

[3] 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.

[4] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[5] Shuo Chen, Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Security vulnerabilities: From analysis to detection and masking techniques. In *Proceedings of the IEEE*, pages 407–418, 2006.

[6] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[7] 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.

[8] 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.

[9] Pramod Koppol, Kedar Namjoshi, Thanos Stathopoulos, and Gordon Wilfong. The inherent difficulty of timely primary-backup replication. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '11, pages 349–350, 2011.

[10] Sape Mullender, editor. *Distributed systems 2nd Edition*. ACM, New York, NY, USA, 1993.

[11] P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded corba applications. In *Proceedings of the 18th SRDS*, pages 263 –273, 1999.

[12] P. Pal, P. Rubel, M. Atighetchi, F. Webber, W. H. Sanders, M. Seri, H. Ramasamy, J. Lyons, T. Courtney, A. Agbaria, M. Cukier, J. Gossett, and I. Keidar. An architecture for adaptive intrusion-tolerant applications. *Software: Practice and Experience*, 36(11-12):1331–1354, 2006.

[13] Tom Roeder and Fred B. Schneider. Proactive obfuscation. *ACM Trans. Comput. Syst.*, 28:4:1–4:54, July 2010.

[14] F. Sabahi and A. Movaghar. Intrusion detection: A survey. In *Systems and Networks Communications, 2008. ICSNC '08. 3rd International Conference on*, oct. 2008.

[15] F. B. Schneider. Beyond traces and independence. *Dependable and Historic Computing, LNCS*, 6875:479–485, 2011.

[16] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[17] Marco Serafini, Peter Bokor, Dan Dobre, Matthias Majuntke, and Neeraj Suri. Scrooge: Reducing the costs of fast byzantine replication in presence of unresponsive replicas. *Dependable Systems and Networks, International Conference on*, pages 353–362, 2010.

[18] 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.

[19] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, 2010.

[20] 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.

[21] PAX team. PAX documentation on ASLR. http://pax.grsecurity.net/docs/aslr.txt.

[22] Hengming Zou and Farnam Jahanian. A real-time primary-backup replication service. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):533–548, June 1999.