Assessing the Attack Resilience Capabilities of a Fortified Primary-Backup System

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 service replication does not constrain that the service be built as a deterministic state machine. It is meant to tolerate crashes, not intrusions. We consider an approach, called FORTRESS, for adding intrusion-resilience capability to a primary-backup server system. It involves using proxies that block clients from directly accessing servers, and periodically randomizing the executables of proxies and servers. We argue that proxies and proactive randomization can offer sound defense against attacks including de-randomization attacks. Using simulations, we then compare the attack resilience that FORTRESS adds to a primary-backup server system with that attainable through state machine replication (SMR) that is fit only for deterministic services. A significant observation is that FORTRESS emerges to be more resilient than an SMR system of four server replicas that are diversely randomized at the start and are subject to proactive recovery throughout.

1 Introduction

The most commonly suggested method for achieving intrusion tolerance is State Machine Replication (SMR). It however requires that the system to be protected execute as a deterministic state machine (DSM). This in turn requires that all sources of non-determinism be identified and resolved. This can be a difficult task in practice as these sources can often be present at several levels, such as, application, programming, middleware and OS levels. Identifying and handling every source of nondeterminism at each level can extract a considerable amount of overhead.

Classical primary-backup replication, PB for short, is most widely used for maintaining service availability despite crashes. Here, one replica, called the primary, does processing and provides state updates to other replicas that act as backups. PB is thus suited to replicating any service without having to deal with sources of non-determinism. On the other hand, it cannot tolerate intrusions.

This led us to examine, in [7], whether a PB system could be made intrusion-resilient without losing its ability to replicate an arbitrary service. More precisely, we explored the possibility of attaining intrusion tolerance without having to comply with the SMR requirement. In [7], we identified an approach, termed as FORTRESS, that can augment intrusion-resilience to *any* server system. In particular, the latter may not even be replicated; if replicated, it can be by PB or SMR. In other words, surviving attacks and service replication can be separated as distinct concerns and SMR is not a pre-condition for attaining intrusion-resilience.

FORTRESS is a union of two ideas. It involves fortifying servers with proxies which block direct server access, and periodically randomizing the executables of proxies as well as servers. Assuming that no server can be compromised until at least one proxy is compromised, the following result was established in [7]: a fortified PB server is at least as attack resilient as a 4-server SMR system; an attacker who cannot intrude more than 1 server in the SMR system cannot compromise the fortified PB server. The result also assumes randomization only at system set-up and proactive recovery as in [6], i.e., no periodic re-randomization. This finding has significant implications: a fortified PB system can have the same degree of resilience as an initially randomized, periodically recovered, 1-tolerant SMR system. We build on this finding in two ways.

Though the use of proxies in server systems is not new, it is examined here in the context of randomization defence and the benefits are highlighted (in section 2). Secondly, we relax a central assumption in [7] and allow servers to be compromised even when they are not directly accessible to an attacker. We then compare, using both analytical models and Monte-Carlo simulations, the resilience of FORTRESS system against simple SMR and PB systems with either proactive randomization or proactive recovery. We consider a realistic range of randomization key entropies and a range of attacker strengths. Section 3 presents the FORTRESS system with its PB servers. Sections 4 and 5 describe the candidate systems, parameters considered and the evaluation methods used in our comparative study whose results are presented in section 6. Section 7 concludes the paper.

2 Background And Related Work

2.1 Attacks and Randomization Defences

Attacks launched for gaining intrusions typically exploit programming and design errors in software running on target systems. (For other forms of 'non-technological' attacks, see [5].) Code injection attacks exploit such an error to achieve (1) injecting malicious code, and (2) changing existing control information (e.g., return address) to have the code executed. If the malicious code is executed, an intrusion occurs, and the attacker gains a greater control over the system leaving the latter *compromised*. The vulnerabilities that the attackers commonly exploit include: unchecked buffer, double-freeing of freed or unallocated heap buffers [2], integer overflow [15], and format-string errors.

To achieve step (2) above, the attacker must know the critical address values; this is easy to determine once the details of the operating system of the target system are figured out, given that the memory layout schemes of all major operating systems are known or can be determined off-line. Moreover, occasional use of an incorrect address value during an attack merely causes crashing of the process serving the attacker who is also a legitimate client. Usually, servers have a forking daemon which forks a new (child) server process if the working one crashes, assuming the causes underlying the crash to be benign. So, an attacker with less than perfect knowledge on memory layout can still succeed so long as the resulting crashes do not alert the administrator.

Address space randomization closes this loop-hole by randomizing the default memory layout. Base addresses of stack and heap in [1] or Global Offset Table in [13] are randomized at runtime, and the random offset or *key* used is securely stored and can be varied at re-boot. Critical addresses therefore take their values at run time from a wide range of possibilities, making it much harder to correctly determine the exact values taken without committing too many mistakes. Address randomization also reduces the success rate of `return-to-libc` attacks.

Other defensive techniques work to make the injection of an executable code impossible/difficult. Notable ones are: W$\oplus$X pages, instruction set randomization and Heap randomization [3]. However, these three techniques are easily bypassed by `return-to-libc` attacks.

As pointed out in [5] and [8], these randomization techniques offer not just efficient defences against intrusions but also cost-effective means of injecting diversity within a replicated system. Deploying varied hardware platforms, different OS and application software with diverse design and implementation is prohibitively expensive. Instead, randomizing identical server replicas using different schemes or different keys of the same scheme, leads to *diverse executables* that have a high degree of failure independence against attacks - a key requirement for intrusion tolerance.

De-randomization attacks are the only known class of attacks to have been tried against randomization defence. They are deployed in [10] and [12] for studying the effectiveness of ASLR and ISR, respectively. These attacks take advantage of the fact that keys cannot be arbitrarily large. In a 32-bit machine using the PaX system [1] only 16 bits of entropy are available, so the random address offset is one of 65536 possibilities.

A de-randomization attack is launched in two phases. The first phase is devoted to determining the current randomization key by probing for every possible key value. If the value being probed is not the key, then the target process will crash; otherwise, it will exhibit a specified behavior.

To complete phase-1, the attacker therefore requires (i) a forking-server that keeps forking a new child whenever the existing one crashes; and, (ii) a way of observing a process crash in the remote target machine. In [10, 12], (i) is assumed and (ii) comes in the form of the `TCP` connection linking the attacker machine to the target: a process crash at the target machine results in the closure of the `TCP` connection that the attacker has with the child server process.

The experiments in [10] and [12] confirm that the randomization key used can be worked out within a finite time.

2.2 Role of Proxies

For a de-randomization attack to succeed, the attacker must reliably observe a process crash on the target machine and a direct `TCP` connection facilitates this in [10] and [12]. Moreover, the attacker should not be identified despite the potentially large number of process crashes he may cause.

The authors of [10] point out two difficulties in suspecting attacks at the server level. The attacker can pace his probes so that the number of crashes he causes in a given period does not exceed the threshold for raising suspicion. Secondly, commercial enterprises deploy several servers for load-balancing; probes launched at different times can be processed by different servers; unless servers exchange observations on process crashes, the attacker evades suspicion when the crashes are distributed across multiple servers.

A de-randomizing attacker is denied his advantages when proxies hide servers from clients; they forward clients' requests to servers and server responses back to clients. When an attacker launches an incorrectly guessed probe, the proxy observes him as having submitted an invalid request; with replicated servers the observation is repeated. Since proxies do not do processing (unlike servers), they can be used for logging their observations on client behavior for longer periods which can be used for identifying sources suspected of launching de-randomization probes.

Proxies can thus reduce the success probability of de-randomization attacks and hence play a major role in our FORTRESS architecture [7]. Using proxies for enhancing intrusion tolerance is not new. Saidane et al [9] use proxies for hiding replication from clients, and also for detecting server intrusions. The performance study in [9] indicates that the overhead due to proxies is minimal when intrusions are not suspected. It does not however measure resilience gains attributable to proxies, which will be the focus here.

2.3 Proactive Obfuscation and Recovery

Re-randomization renders an attacker's success in eliminating certain keys meaningless. It restores resiliency against further attacks. However, it cannot be done online and requires the files to be re-compiled, re-linked and re-loaded. Hence, a re-boot is essential. Periodic re-randomization is commonly referred to as *proactive obfuscation* [4, 8]. In this paper, the re-randomization period will be referred to as the *unit time-step*.

The probability that a de-randomization attack succeeds on a given machine within a unit time-step depends on the number of probes (ω) that an attacker can complete within the unit time-step and the number of keys (χ) available.

Roeder and Schneider [8] comprehensively investigate applying proactive obfuscation to a state machine replicated (SMR) server system. They identify the necessary mechanisms and assumptions, build two prototypes and measure throughput and latency. The main challenge lies in accomplishing proactive obfuscation without stopping the SMR system itself, and it has been addressed as described below.

For the SMR system to be f–tolerant, it must have $n > 3f$ server replicas. At specific instances, a batch of at most f replicas (logically) exit the SMR system to be re-booted and re-randomized, and re-join the system after having restored the service state *and* before the next batch is to exit. There are thus at least $\lceil \frac{n}{f} \rceil$ state restorations per unit time-step. Each one succeeds because $n - f > 2f$ and the re-joining replicas have at least $(f + 1)$ correct working replicas to supply the correct service state.

Incorporating proactive obfuscation therefore requires strict synchrony assumptions for timely exchange and processing of state messages; also requires synchronized clocks, secure components and timely links for timely start and completion of batched re-randomization. Note that these requirements are much stronger than the weakest environments in which SMR can be managed.

Proactive obfuscation is a more robust version of *proactive recovery* [6] where the same software is re-installed during periodic re-boot. A limited form of randomization is possible in [11] where a replica chooses, during re-boot, an executable from a small set of candidates.

3 Fortifying a Server System

In [7], we presented proactive fortification as a means of incorporating attack resilience to *any* given server system; the latter may or may not be replicated for fault-tolerance; if replicated, any replication strategy can be used. The architecture is termed as FORTRESS and prescribes fortifying a server system of n_s servers using n_p redundant proxies. As mentioned in 2.2, proxies act as intermediaries between clients and the server system.

FORTRESS also prescribes that the n_s server nodes are uniquely indexed and the indices are known to servers and proxies. Client can know proxies' addresses and public keys, servers' indices (not addresses) and public-keys, the type of replication, and the degree of fault-tolerance if replication is by SMR. This is facilitated through a trusted *name-server* (NS) that is read-only for clients. (see [7] for details.) Servers accept messages only from proxies and NS.

When the server tier is a fault-tolerant system, $n_s > 1$. Of interest here is that tier being a primary-backup replicated system. Note that such a server system can tolerate only node crashes, relies on failure independence and is not inherently equipped to resist intrusive attacks. Below, the interactions between clients and proxies, and between proxies and primary-backup server are described.

Clients send their requests to all proxies and each proxy submits the request to each server. The primary processes each unique request and sends its response to all backup servers. Each server signs the response together with its index and returns the signed response to every proxy. A proxy over-signs any one of the authentic responses and forwards the doubly-signed response to the client. A client accepts a response as valid if it has two authentic signatures - one from the proxy that sent the response and the other from one of the servers. Note that proxies do not interact with each other during client-server interactions.

FORTRESS prescription includes that the proxies and server nodes are periodically re-randomized such that the same key is used for servers which is distinct from the different keys used for randomizing proxies. Thus, at any time, $(n_p + 1)$ randomly-selected keys will be in use.

Servers are randomized identically so that regular state updates of backup servers by the primary can be done warranting no changes to an existing primary-backup system. If randomized differently, state *representations* within each server can be different (even though states are no different); this means that marshalling and unmarshalling functions would be needed to convert a server's state representation to/from an abstract representation that is the same for all servers. Since proxies do not interact with each other, randomizing them differently incurs no overhead.

The fortified system fails against an attacker if he compromises one of the server replicas; we would expect the attacker to target the primary server, which is readily identifiable as it is the one that executes requests from clients.

It is meaningful for an attacker to target only the primary server for two reasons: (i) the primary server is readily identifiable as it has to send regular state updates to others; (ii) a successful attack on the primary server compromises S_1.

The fortified system is said to be compromised when an attacker compromises either the primary or all n_p proxies.

As explained in 2.2, the presence of proxies makes it harder for successful de-randomization attacks to be launched when no proxy is compromised. An attacker's strategy may be to compromise one proxy and then use that proxy to launch attacks on a server over a direct `TCP` connection (as in [10, 12]). We note that proxies do not do any processing, so compromising a proxy is harder than compromising a server that is directly accessed by the attacker.

4 Models of System and Attack

The systems considered for comparison are characterized here by the number of tiers they have and whether or not re-randomization is used. The number of tiers can be either 2 or 1 when the server tier is and is not fortified with a proxy tier, respectively. While the server tier of a 2-tier system implements only primary-backup replication, the 1-tier server system can implement either one of the replication types. Thus, three system classes are defined.

Definition 1 *S_0 is the 1-tier server system implementing state machine replication.*

S_0 consists of 4 differently randomized nodes implementing a service built as a DSM. Clients interact with these nodes directly. The nodes execute an order protocol to decide on the order for processing requests; correct nodes generate identical responses for each request. S_0 is compromised as soon as more than one node is compromised.

Definition 2 *S_1 is the 1-tier server system implementing primary-backup replication.*

S_1 consists of 3 nodes. Clients interact with all of these nodes directly. The primary node processes client requests and passes the state updates and the results to the backup nodes. Should the primary node crash, it is detected and one of the backup servers becomes the new primary.

As compromising the primary server is all that is required for compromising S_1, differently randomizing servers does not add any extra resilience, but complicates the state updating of backups by the primary. Therefore, it will be assumed that the servers of S_1, as in FORTRESS servers and unlike in S_0, are randomized identically.

Given that the servers of S_1 are randomized identically and the primary server is the only obvious target to attack, it makes sense to regard compromising any one of them the same as compromising the primary and hence S_1 itself.

Definition 3 *S_2 is the 2-tier system where the server tier implements primary-backup replication.*

S_2 complies with the FORTRESS architecture with $n_s = n_p = 3$. As in S_1, we assume that compromising any server replica is the same as compromising the primary. Given that an attacker launches attacks simultaneously on all proxies and servers, the compromise of S_2 can happen in three ways. Either the attacker manages to compromise a server without having compromised any proxy, compromises a proxy and uses it as a launch pad from which to compromise a server, or compromises all proxies.

4.1 Modeling Obfuscation

Proactive obfuscation (PO), is modeled with two parameters: re-randomization period P, and the available diversity (χ) determined by the entropy of randomization key. We take P to be one unit time-step. Thus, in every unit time step, every node is re-randomized if PO has been assumed.

The efficacy of PO depends on χ, the number of possible randomization keys. In practice, the randomization key entropy appears to be 16 bits or 32 bits, and hence we consider the case $\chi = 2^{16}$ in this paper for evaluation.

We assume that re-randomization is completed instantly at the end of each unit time-step. This simplifying assumption is justified as we focus on the assessment of attack resiliency (and not on latency or throughput). So, proactive obfuscation is modeled as randomly selecting fresh keys at the end of each unit time step. Note that there is a non-zero probability that a chosen key has been selected in earlier time-steps. This probability gets smaller as χ gets larger.

Start-up only obfuscation represents the case where nodes are only randomized initially. Subsequently, nodes are simply recovered at the end of each unit time step. We call this scheme *start-up only obfuscation* (SO). It can be seen as proactive recovery [6] after an one-off initial randomization (see also section 2.3). With SO, an attacker trying to guess the randomization key is like sampling without replacement, and the probability of making a correct guess increases as more samples are completed.

However, with periodic re-randomization, i.e., with PO where $P = 1$, guessing the randomization key in use is like sampling with replacement. That is, the probability of successfully guessing the key during a unit time-step is independent of guesses made during earlier time-steps.

4.2 Types of Attacks

Attacks can be of two classes, depending on whether the attacker interacts directly with the target. If that is the case, it is called a *direct* attack; if the attacker cannot directly interact with the target and dispatches the attack to the target through another machine, then the attack is called *indirect*.

Attacks on servers in S_0 and S_1 systems (also in [10, 12]) are direct attacks. In S_2, direct attacks target proxies while the indirect ones target servers and are dispatched through proxies; an attacker can simultaneously launch direct attacks against proxies and indirect ones against servers.

Once a direct/indirect attack against a given node succeeds, we assume that the attacker compromises that node and continues to control it until re-randomization is applied.

Definition 4 *The probability that a direct attack on a proxy or server node launched in the i^{th} time-step succeeds is α_i.*

The probability α_i depends on three key variables: χ (the number of keys available for randomization), ω (the number of probes an attacker can launch in a unit time-step), and whether randomization is PO or SO.

Indirect attacks are launched against the servers of a 2-tier S_2 system through the proxies.

We have noted earlier (in 2.2) that when a proxy handles an incorrectly guessed probe, it observes the client having submitted an invalid request; by logging these observations for several time-steps, and analyzing the frequency of such occurrences from a given source, it can fairly accurately deduce whether the client is acting with malicious purposes.

Given this possibility, the attacker is forced to opt for a smaller ω to evade detection; this means that the presence of proxies effectively reduces ω of an attacker. That is, for a given attacker, his indirect attacks tend to have lower probability of success compared with his direct attacks. This is modeled using the *indirect attack coefficient*, denoted as κ.

Definition 5 *The probability that an indirect attack launched in the i^{th} time-step succeeds is $\kappa \times \alpha_i$, and κ, $0 \leq \kappa \leq 1$ is the indirect attack coefficient.*

Note that κ is independent of the number of proxies deployed. Also α_i is independent of i for PO systems; $\alpha = \alpha_i, \forall i \geq 1$. For SO systems, $\alpha = \alpha_1$ and we derive α_i, $i \geq 2$, from α_{i-1} assuming that χ is large compared to ω. Note that α_i decreases as i increases in the SO case; attacks on such a system are like sampling without replacement.

Definition 6 *The probability that a direct attack succeeds on a node during a unit time-step that begins immediately after that node has been (re-)randomized, is denoted as α.*

5 Evaluation Methods

The evaluations presented in this paper assume that time progresses in unit time steps, starting at time T_0 when all nodes are assumed to be correct.

Definition 7 *The expected lifetime (EL) of a system is the expected number of whole unit time steps that elapse until the system is compromised. We say that system A* outlives *system B if EL of A is larger than EL of B. It is denoted as $A \rightarrow B$. $A \xrightarrow{c} B$ implies that $A \rightarrow B$ holds under condition c.*

Finally, we use either Absorbing Markov Chain methods (where state spaces are sufficiently small) or Monte-Carlo simulations to determine the EL until system compromise.

In our assessment, we consider the range of α values from 0.00001 to 0.01 as a realistic range. This range covers a variety of α values where at least one of the systems has large enough EL to be a useful real system.

6 Results

The expected lifetimes (EL) for various systems are presented in figures 1 and 2, showing four key trends.

S_1SO outlives S_0SO: This result is significant as SMR with proactive recovery is widely perceived to be an important method for achieving intrusion-tolerance [6], [11]. The explanations are 2-fold.

First, randomization is done only at start-up; so, every unsuccessful de-randomization probe *continues* to eliminate one key from χ possible ones. In S_0, each of the four nodes is randomized once using a distinct key. The probability that the i^{th} probe, $i \geq 1$ launched uncovers any one of these 4 keys, assuming none has been uncovered so far, is $\frac{4}{\chi - i}$. Similarly, the probability that the i^{th} probe $i \geq 1$ uncovers the second key, assuming that only one key has been uncovered so far, is $\frac{3}{\chi - i}$. Compromise of S_0 happens when the latter event occurs. In S_1, all nodes are identically randomized using a single, randomly chosen key; once that key used is uncovered, S_1 is compromised. Hence, the probability that the i^{th} probe compromises S_1 is $\frac{1}{\chi - i}$, which is at least three times smaller. Thus, when probes are crafted as service requests and when all other factors remain the same, S_1 is likely to be compromised later than S_0.

Secondly, real diversity is not being considered, only artificial diversity through randomization. With the former, a vulnerability is unlikely in all replicas; whereas, here, replicas may share the same vulnerable bug; if so, only the defense they have against that bug being exploited are diverse.

S_2PO and S_1PO outlive all SO Systems: This suggests that proactive obfuscation is a more effective technique for intrusion tolerance than either SMR or proactive recovery.

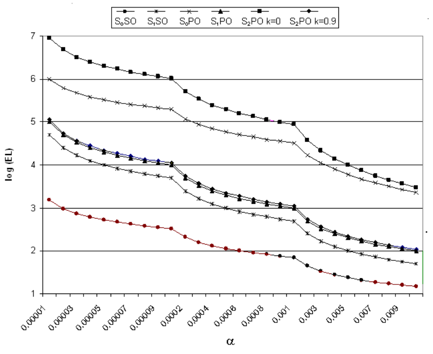

Figure 1. Expected Lifetime Comparison

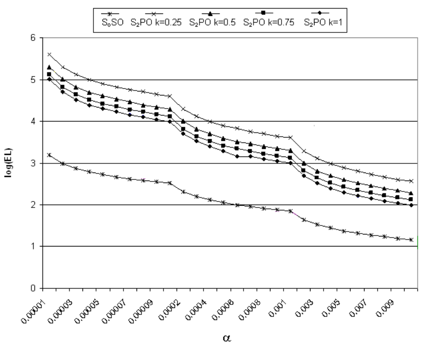

Figure 2. Expected Lifetimes of the S_2PO Systems as κ varies (logarithmic scale)

$\mathbf{S_2PO}$ outlives $\mathbf{S_1PO}$ when $\kappa \leq \mathbf{0.9}$: As long as proxies causes difficulty to an attacker in launching successful indirect attacks, the FORTRESS system is superior to implementing proactive obfuscation directly on a PB system.

$\mathbf{S_0PO}$ outlives $\mathbf{S_2PO}$ except when $\kappa = \mathbf{0}$: This suggests that unless proxies can completely prevent indirect attacks, an SMR system using proactive obfuscation is the most effective of all systems considered.

In summary,

$$S_0PO \xrightarrow{\kappa > 0} S_2PO \xrightarrow{\kappa \leq 0.9} S_1PO \rightarrow S_1SO \rightarrow S_0SO$$

7 Conclusions

The resilience assessment carried out here helps us make a design choice between SMR and FORTRESS approaches. If compliance to deterministic state machine (DSM) is easily achievable or already available, then SMR with proactive obfuscation is recommended. The least effective way forward appears to be SMR with proactive recovery, given that the infrastructure support for recovery is substantial and can be extended for obfuscation (see [8], [11] and [6]). If DSM compliance is costly or not feasible, then primary backup replication with FORTRESS is readily recommended. Detailed comparison of FORTRESS with SMR that is firewalled for confidentiality reasons [14] will be a future work.

References

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

[2] Anonymous. Once upon a free(). *Phrack Magazine*, 57(9), August 2001.

[3] E. D. Berger and B. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *In Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 158–168. ACM Press, 2006.

[4] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 105–120, Berkeley, CA, USA, 2003. USENIX Association.

[5] K. P. Birman and F. B. Schneider. The Monoculture Risk Put into Context. *IEEE Security & Privacy*, 7(1):14–17, 2009.

[6] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4):398–461, 2002.

[7] P. Ezhilchelvan, D. Clarke, I. Mitrani, and S. Shrivastava. Proactive Fortification of Fault-Tolerant Services. In *Proceedings of the 13th International Conference On Principle Of DIstributed Systems, LNCS 5923*, pages 330–344. Springer Verlag, 2009.

[8] T. Roeder and F. B. Schneider. Proactive obfuscation. Technical report, Cornell University, March, 2009.

[9] A. Saidane, V. Nicomette, and Y. Deswarte. The design of a generic intrusion-tolerant architecture for web servers. *IEEE Transactions on Dependable and Secure Computing*, 6(1):45–58, 2009.

[10] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proc. of the 11th ACM conference on Computer and Communications Security*, pages 298–307. ACM, 2004.

[11] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo. Resilient intrusion tolerance through proactive and reactive recovery. In *Proc. 13th Pacific Rim International Symposium on Dependable Computing PRDC 2007*, pages 373–380, 17–19 Dec. 2007.

[12] A. Sovarel, D. Evans, and N. Paul. Where's the feeb?: The effectiveness of instruction set randomization. In *Proceedings of the 14th USENIX security symposium*, pages 145–160. Usenix Association, 2005.

[13] J. Xu, Z. Kalbarczyk, and R. Iyer. Transparent runtime randomization for security. In *Proceedings of SRDS*, pages 260–269, Oct. 2003.

[14] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *SOSP '03*, pages 253–267, New York, NY, USA, 2003. ACM.

[15] M. Zalewski. Remote vulnerability in SSH daemon crc32 compensation attack detector. *RAZOR Bindview Advisory CAN-2001-0144*, 2001.