Diverse Replication for Single-Machine Byzantine-Fault Tolerance

Byung-Gon Chun[†], Petros Maniatis[⋆], Scott Shenker[†‡]
[†]*ICSI*, [⋆]*Intel Research Berkeley*, [‡]*UC Berkeley*

Abstract

New single-machine environments are emerging from abundant computation available through multiple cores and secure virtualization. In this paper, we describe the research challenges and opportunities around diversified replication as a method to increase the Byzantine-fault tolerance (BFT) of single-machine servers to software attacks or errors. We then discuss the design space of BFT protocols enabled by these new environments.

1 Introduction

Current commodity computing architectures still offer increasing, abundant power, despite the pessimists' periodic proclamations that the underlying technology has reached its limits. Chip makers double the number of standard processing cores per chip with every semiconductor process generation, and some research groups already design programming models and system architectures for tens to thousands of cores per chip [2, 6]. Nevertheless, the dependability of applications running on this abundant power has not necessarily similarly improved. First, as more transistors are crammed into the same chip area, soft errors and undetected hardware defects increase in incidence; furthermore, as cycle-hungry software expands to absorb what hardware evolution provides it, its complexity also increases leading to more bugs, more malicious exploits of those bugs, and more application crashes.

In this paper, we revisit the classic idea of replication-based system reliability: run the same application (say, a database server) in multiple instances on the same powerful server, and perform some form of consensus on the results of each request across replicas, to mask out faults. Multi-processor systems as far back as NonStop [10] and TARGON [14] used this idea to increase reliability at the instruction or kernel level, and distributed replicated services using Paxos [26] or Castro and Liskov's PBFT [16] are increasingly practical [4,18,19,25]. In all cases, some form of voting among replicas helps choose the right result, as long as no more than a maximum fraction of all replicas are faulty at any time. In this position paper, we focus on using replication in single-server systems—that is, collocating multiple instances of the replicated service on the same physical machine—to take advantage of per-machine cycle abundance for Byzantine-fault tolerance to software attacks or errors. Performance is not necessarily the major challenge: running multiple

replicas in their own virtual machines (VMs) within a single multi-core system is already feasible without incurring performance overheads [15]. Our goal is to provide single-server systems with an increase in *reliability* that is commensurate with current trends in computation power.

A fundamental challenge towards our goal is ensuring that replicas of a server fail independently. For instance, if the exact same bugs can be triggered by a malicious exploit in all instances of the code, replication will be an ineffective dependability tool: no amount of voting will be able to mask faults if all replicas exhibit the same fault. Two trends in current research and technology make us hopeful on this front. First, recent hardware and software advances lead to improved isolation among different execution domains: Intel's LaGrande platform, a.k.a. Trusted Execution Technology (TXT) [1] and AMD's Presidio (Secure Virtual Machine—SVM) [5]) can isolate threads, processes, or virtual machines from each other, and similar advances such as SELinux and Singularity [21] improve on software-only isolation. Second, much work has been done to increase the diversity and predictability of runtime systems, both to expose bugs that might cause silent faults, and to prevent adversaries from reliably exploiting software bugs. In particular, recent research has diversified Guest OSes and rewritten binaries via randomizing address space layout, system call interfaces, instruction set numbering, event delivery, etc. [9, 13, 17, 22, 24, 28, 34].

In what follows, we hope to describe the research challenges and opportunities around *diversified replication* as a method to increase the Byzantine-fault tolerance of single-machine servers. We start by treating the execution environment itself, in terms of its isolation and diversity characteristics (Section 2). Then we describe the design choices available in adapting traditional Byzantine-fault tolerant replication to such a single-machine execution environment, presenting some potentially promising design points in the space (Section 3). Finally, we discuss the ecosystem around such a dependability approach, including dealing with core defects, soft errors [11] or hardware failures (Section 4). We conclude with related work and our research agenda.

2 Challenges in a BFT Server

The execution environment we consider uses BFT replicated state machines within a single server to tolerate software Byzantine faults. Traditionally, such replication
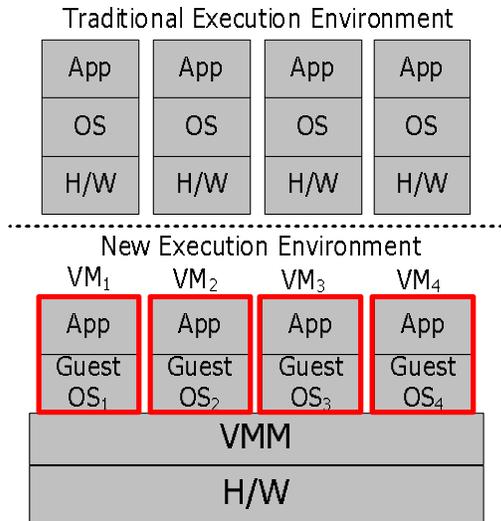
Figure 1: Traditional execution environment (top) vs. new execution environment (bottom). In our new execution environment, each replica runs in a VM running a diversified Guest OS.

is used across *multiple* machines, usually geographically distributed. There, a client issues an authenticated request to the replicas over the network. Those requests are ordered using methods such as agreement or ordered atomic broadcast. Then each replica executes each request in that order on its local copy of the service state. Figure 1(top) illustrates this traditional execution environment. Running multiple such replicas in a single machine is not obviously useful, since replicas compete with each other for resources slowing the whole system down, and they can interfere with each other spreading bugs or malicious faults, losing fault independence which is essential for replication to work. Thanks to multi-core architectures, the performance disincentives are no longer debilitating (especially for CPU-bound services).

To provide fault independence, the main challenges are *isolation* among replicas so that they do not interfere, and replica *diversity* so that they do not suffer from common vulnerabilities. We propose to virtualize replicas, running each replica in a VM with an automatically and systematically "diversified" Guest OS, replication software, and application replica (see Figure 1(bottom) for an illustration). Here, our trusted computing base (TCB) is the VMM and the hardware. Compared to making the entire OS and application stack reliable through sound software practices, making the VMM reliable enough to be justifiably inside the TCB is much more plausible: the Xen[1] VMM consists of tens of thousands of lines of code, but Microsoft Vista consists of 50 million lines of code even excluding the target application service.[2]

Next, we describe in detail what isolation features are required for protecting domains (VMs) and how Guest OSes can be diversified automatically.

2.1 Isolation

To isolate protected domains (VMs), the VMM should provide isolation of computation, memory, and disk. In addition, it should provide a protected communication mechanism between VMs for efficient communication.

The VMM schedules CPU resources fairly among multiple VMs by partitioning CPU time. A VMM should ensure availability and liveness by not starving VMs. The VMM must protect physical memory pages of a VM from inappropriate accesses of other VMs. The VMM achieves this by partitioning the memory space, and controlling the paging mechanism by managing the register that contains the base address of the page directory. In addition, the VMM should shepherd DMA-capable devices.[3] If a page is protected, the page is indicated in a secure table maintaining protected pages, a page fault occurs in a DMA, and the VMM disallows the DMA access to the page. Hardware trusted execution and protection such as Intel TXT [1] and AMD SVM [5] support a way to implement memory protection from DMA. For example, TXT supports the noDMA table that maintains this information at a chipset component.

The VMM also isolates virtual disks used by VMs by partitioning physical disk(s) into non-overlapping disk space. All disk I/Os are mediated by the VMM, thus a VM cannot directly access the virtual disk of other VMs. Finally, the VMM should ensure that communication between two VMs cannot be tampered with by another VM. No direct communication occurs among VMs, but the VMM mediates all communication among VMs.

There could be several ways of providing this isolation. The minimal mechanism that provides such isolation is a fundamental research question, especially in view of existing and future trusted hardware extensions for protected execution.

2.2 Diversity

Providing isolation is not enough to defend against software attacks. If replicas can be exploited simultaneously due to their common vulnerabilities, the replicated system cannot meet the bounded-fault assumption on which replication guarantees are founded. To avoid such correlated faults, replicas need to be diversified. Replicas are *diverse* if they maintain the same application semantics (i.e., they produce the same output given the same input) even though their implementation details (i.e., actual instructions executed) may be different.[4] It is hard to compromise diverse replicas simultaneously via bug exploits or other such software attacks, since those attacks typically rely on specific memory layouts or instruction sequences.

Traditional approaches to achieving diversity include N-version programming [8] or using N different implementations of the same specification (e.g., by different vendors) [30–32]. N-version programming takes a design diversity approach: multiple teams produce different implementations of the same specification. N-version programming is rarely used in practice due to its high development and maintenance costs. Instead, opportunistically using existing independent implementations of standard services or interfaces has been more productive in practice. HACQIT [30] uses Apache running on Linux and IIS running on Microsoft Windows. HRDB [32] uses different DBMS implementations that support SQL. BASE [31] supports different NFS implementations through abstraction. Such implementation diversity is applicable only when different implementations that produce identical outputs exist and often when the service has a standardized high-level abstraction (e.g., SQL).

To create diversified replicas, we take a more broadly applicable approach, using OS and binary *randomization*. In particular, we propose to combine multiple existing randomization techniques to create diversified Guest OSes and/or binaries to run within VMs. Such randomization was used before to isolate bugs or to forestall particular intrusions via a specific attack vector. We use only techniques that preserve the same application semantics, so that it is safe to use voting over the replicas' results.

Combining diversification techniques is not straightforward. Some techniques can be used continuously, while some may conflict with each other, or in combination lead to unacceptably high overheads. We discuss some instances and lay out the challenges of creating diverse replicas.

Randomizing the location of stack or heap memory [13,22,34] has been explored to defend against buffer overflow attacks. Recent OSes such as Microsoft Vista and Mac OS X Leopard employ address space layout randomization (ASLR) as well. Randomizing interface mappings has also been studied: system call interface randomization changes mappings between system call numbers and code [17], while instruction set randomization changes mappings between opcodes and instructions. In both cases, an exploit meant to piggy-back executable code in a buffer overflow cannot know which system call number or even instruction opcode to use. Recovering from bugs in Rx [28] also relies on various randomizations: memory management randomizations including delayed recycling of freed buffers, padding allocated memory blocks, allocating memory in an alternate location, and zero-filling newly allocated memory buffers; and asynchronous event delivery including scheduling of events, signal delivery, and message reordering.

With these techniques, we can create diversified Guest OSes by combining randomization techniques with the following design goals: 1) maximizing the diversity of the replicas, 2) avoiding conflicting diversity techniques, and 3) controlling the overhead of diversity techniques.

There are a few research challenges in this scheme due to the use of randomization. First, among the sets of diversified Guest OSes, how does one choose those with the highest diversity? Naïvely using all randomization techniques together may be ineffective (e.g., delayed freeing combined with relocation defaults to one or the other according to their order of application). Even worse, some techniques in combination may result in conflicts or incur overheads beyond what applications can tolerate. For example, using two different stack randomization techniques at the same time may incur incorrect stack frames. The interplay of different types of techniques must be carefully mapped.

Creating diverse replicas using randomization also offers an opportunity to quantify the overall system probabilistically. We conjecture that increasing the number of replicas is likely to increase the safety of the system exponentially. To answer these questions, we would like to research a formal way to measure diversity and run real experiments with various attacks to quantify diversity.

Second, by using randomization techniques we may transform Byzantine faults to crash faults (e.g., because of failed exploits accessing disallowed memory regions). Exploits that were successful for the original undiversified code may translate to exploits that are still successful (i.e., cause safety violations), cause crashes, or are masked by our system. We need to examine how different exploits translate to different outcomes in our environment; e.g., this environment might have more crash faults than the original code. Having crash faults is clearly better than having faults violating safety, and we speculate that restarting VMs running replicas with crash faults in our secure virtualization environment can fix this problem without side effects. This is related to the Nysiad approach [23] of making crash-fault tolerant distributed applications to ones that are Byzantine-fault tolerant.

3 New BFT Opportunities

In this section, we discuss the spectrum of BFT replicated algorithms enabled by our new execution environment. In particular, the environment allows us to explore three axes of designing BFT systems: the trust characteristics of a coordinator process, the synchrony assumptions of communication among replicas, and the level of replication transparency towards clients.

The first design axis concerns whether the facility that orders client requests is trusted or not. In traditional replicated state machines, a replica acting as a *coordi-*

nator or *primary* typically assigns sequence numbers to requests. In a single-machine environment, this task can be taken on by a distinguished coordinator component. Since this functionality is simple, the coordinator may be straightforward to implement and formally prove correct, therefore justifying its inclusion in the TCB. On the other hand, not trusting the coordinator keeps the TCB leaner, but results in more complex replication protocols.

The second axis concerns the question of synchrony assumptions within the single machine. If one were to assume memory and I/O buses to be fault-free and all replicas equally fast regardless of diversification, then all communication among VMs can be thought of as synchronous. This simplifies the replication protocols, since no ambiguity exists among "slow" and maliciously "mute" replicas that try to stall the progress of the system. On the other hand, buses are not always fault-free, different diversification techniques or random seeds might result in vastly different execution times of the same code in different replicas at different times, making it difficult to maintain this synchrony assumption. Replication protocols for asynchronous environments are significantly more complex as a result. Bridging the gap between the two, one might imagine enforcing synchrony via the VMM by bounding execution time at replicas and restarting execution after a certain timeout, or resorting to eventual or virtual synchrony designs.

The final axis concerns the transparency of replication to clients. In traditional replicated systems, clients do interact with multiple replicas. In our setting, the coordinator can collect replies from multiple replicas inside a machine and interact with clients directly, offering the illusion of an unreplicated service, which is simpler and requires only a single communication session between the client and the server. In contrast, if replication is exposed to the client, individual replicas at the server communicate with the client directly. At the expense of greater bandwidth requirements and greater protocol complexity, exposure of replication removes the aggregation task (vote tallying, formation of a single response message) from the coordinator's functionality.

In the design space we have described, there are many design combinations. For example, a design point that is closest to traditional agreement-based BFT protocols is built using an untrusted coordinator, asynchrony, and exposed replication. Here, we take two other unconventional design points from the design space and explain the resulting specific protocols in detail to illustrate that BFT can be achieved in a simpler way in our new execution environment.

To explain protocols concisely, we use the authentication notation of Yin et al. [35], according to which we denote by $\langle X \rangle_{S,D,k}$ an authentication certificate that any node in a set D can regard as proof that k distinct nodes
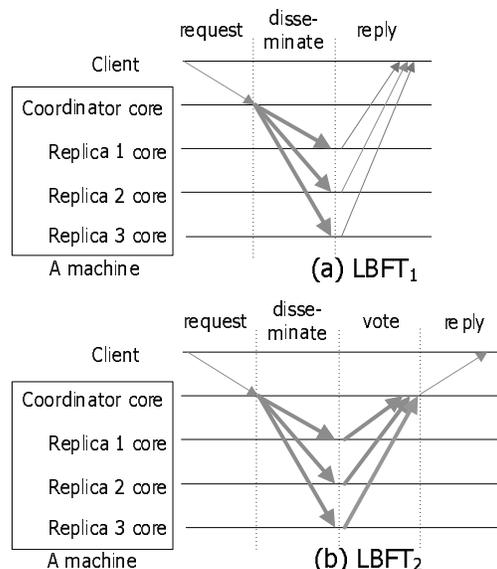

Figure 2: Lightweight BFT (LBFT) protocol instances. Thicker lines indicate communication inside a machine.

in S said X. For example, a traditional digital signature from p that is verifiable by the entire replica population R would be $\langle X \rangle_{\{p\},R,1}$, and a message authentication code (MAC) from p to q would be $\langle X \rangle_{\{p\},\{q\},1}$. If X is not signed, there is no subscript in the notation.

LBFT$_1$: In LBFT$_1$, we assume a trusted coordinator, asynchrony, and non-transparent replication. We run one coordinator and a set R of $2f + 1$ execution replicas to provide both safety and liveness with up to f Byzantine faults. Figure 2(a) shows LBFT$_1$, our simple protocol that uses the trusted coordinator. When p, the coordinator, receives $req = \langle \text{REQUEST}, o, t, c \rangle_{\{c\},R,1}$ from client c where o is the operation requested and t is the timestamp, it multicasts to R a $\langle \text{DIST}, n, req \rangle$ message where n is the assigned sequence number. Note that there is no authenticator in this message. Each replica executes the request, and sends a reply message to the client. When the client receives $f + 1$ valid matching reply messages forming the reply certificate $\langle \text{REPLY}, n, t, c, r \rangle_{R,\{c\},f+1}$, it accepts the result r.

It is worthwhile noting the distinct characteristics of this protocol. The coordinator does not deal with any cryptographic operation, but only replicas and clients perform the operations: there is no cryptographic operation between the coordinator and clients and between the coordinator and replicas. In this protocol, the client knows that it interacts with multiple replicas and performs majority voting from the received reply messages.

By trusting the coordinator, LBFT$_1$ reduces the complexity for implementing BFT protocols. It needs only a single phase request dissemination from the coordinator to the replicas with $O(N)$ messages. It does not need

to deal with a faulty coordinator (which would required complex *view change* algorithms). In addition, it needs $2f + 1$ replicas instead of $3f + 1$ replicas. The downside of the trusted coordinator is that our TCB grows slightly. **LBFT$_2$:** By changing LBFT$_1$ slightly, we can create a protocol LBFT$_2$ that performs transparent replication. LBFT$_2$ (shown in Figure 2(b)) provides clients with an interface that is similar to interacting with a single server.

In LBFT$_2$, when p, the coordinator, receives a request message $\langle \text{REQUEST}, o, t, c \rangle_{\{c\},\{p\},1}$, it checks whether the message authenticator is correct. If the authenticator is correct, it disseminates the message as in LBFT$_1$. Otherwise, it drops the message. When each replica executes the request, it sends its reply $\langle \text{REPLY}, n, t, c, r \rangle$ back to p. Note that there is no authenticator in the message. When p collects $f + 1$ valid matching reply messages (i.e, performs majority voting), it forwards to the client a reply message $\langle \text{REPLY}, n, t, c, r \rangle_{\{p\},\{c\},1}$.

Unlike LBFT$_1$, this protocol incurs more cryptographic overhead at the coordinator, which must verify request authentication and collate/sign responses to clients. However, this protocol simplifies the interface to clients and reduces wide-area bandwidth overheads. Clearly many trade-offs exist in the customization of such replication protocols and different choices make sense for any given operating environment.

4 Discussion

Core failures: Core failures such as manufacturing defects and soft errors are becoming a concern as more and more transistors are put into a processor without a corresponding reliability increase for each individual transistor. We can mask these core failures by modifying the VMM running our architecture. First, the VMM should provide the capability to pin a VM at a specific core. If a VM is allowed to run at any core (typical in the symmetric multi-processing (SMP) model), the a defective core can affect all VMs, thus violating our fault assumptions. Second, the VMM itself should be able to handle core failures. This can be achieved by replicating VMM state, or by running the VMM at a more reliable core if such core heterogeneity is supported by the platform.

Machine crashes: Our BFT system runs in a single machine. If the machine crashes (e.g., hardware and power failure), our system is not available. To handle this type of failures, we can run our BFT system in multiple machines and coordinate the BFT groups. For example, an extended system coordinates BFT groups (one group per machine) using a benign fault tolerant replicated state machine algorithm like primary-backup replication and Paxos; this is a hybrid of benign and Byzantine fault tolerant protocols. There are other ways to organize these groups that give different fault tolerance properties and protocol complexity.

Adaptive replication: This replication utilizes idle cores opportunistically for improved fault tolerance. Instead of fixing the number of replicas, we can adaptively change the number of replicas based on load to keep performance the same. Adaptive replication trades off fault tolerance for performance.

Confidentiality: Our BFT system increases integrity and availability. However, the system can weaken confidentiality since compromising any one replica can leak sensitive information. This problem can be addressed by using threshold signatures [29] or privacy firewalls [35]. For example, to obtain sensitive information in threshold signature schemes, the attacker should compromise the *majority* of replicas.

5 Related Work

We briefly discuss other related work which is not presented in previous sections.

Diversity has also been explored for detecting *specific* malicious behavior. TightLip [36] runs a shadow process to detect the use of sensitive information. N-Variant Systems [20] aim to create general, *deterministic* variants with disjoint exploitation sets. The authors showed address space partitioning that detects attacks using absolute addresses and instruction set tagging. We might be able to borrow techniques from either camp to diversify virtualized applications on the fly.

Processor-level replication that addresses benign faults has been widely studied. All instructions of the application are replicated and checked through hardware redundancy [12, 33] or time redundancy [7]. These instruction-level replication schemes are suitable for masking hardware faults, but not for defending against software attacks. Executing the same instruction stream having common vulnerabilities multiple times does not help to mitigate Byzantine faults.

Replication systems such as MARE (Multiple Almost-Redundant Executions) [37] focus on reducing replication costs for software and configuration testing and checking. Such a partial-replication system executes a single instruction stream most of the time, and only runs redundant streams of instructions for some parts of a program. This approach could be used to provide the adaptive replication mechanism we envision in Section 4. Replicant [27] creates replicated processes with OS support and focuses on handling non-determinism caused by multiple threads running in a process.

6 Conclusion

In this paper, we argue for a new single-machine execution environment for BFT which is emerging from abundant computation through many cores, virtualization, and trusted execution and protection. To avoid common exploits, we propose to run diversified Guest OSes

in VMs by combining several OS and binary randomization techniques. In this new execution environment, we discuss new BFT protocol design opportunities, e.g., factoring out serialization from traditional BFT protocols and pushing it to our TCB, or designing protocols on synchrony assumptions. As future work, we plan to develop prototypes of systems from our design space and evaluate the diversity, availability, and performance of the systems.

References

[1] Intel trusted execution technology. http://www.intel.com/technology/security.

[2] Tera-scale computing research program. http://techresearch.intel.com/articles/Tera-Scale/1421.htm.

[3] Xen. http://xen.org.

[4] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP*, 2005.

[5] Advanced Micro Devices. AMD64 architecture programmer's manual: Volume 2: System programming. 2005.

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, UC Berkeley, 2006.

[7] T. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Symposium on Microarchitecture*, 2001.

[8] A. Avizienis and L. Chen. On the implementation of n-version programming for softwae fault-tolerance during program execution. In *International Computer Software and Applications Conference*, 1977.

[9] E. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, and D. Zovi. Intrusion detection: Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM CCS*, 2003.

[10] J. F. Bartlett. A NonStop kernel. In *SOSP*, 1981.

[11] R. C. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. In *IEEE Reliability Physics Tutorial Notes, Reliability Fundamentals*, 2002.

[12] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop advanced architecture. In *DSN*, 2005.

[13] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security*, 2003.

[14] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Trans. Comput. Syst.*, 7(1), 1989.

[15] R. E. Carpenter. Comparing Multi-Core Processors for Server Virtualization. *IT@Intel*, Aug. 2007. http://www.intel.com/it/pdf/multicore_virtualization.pdf.

[16] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4), 2002.

[17] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. In *Technical Report CMU-CS-02-197, CMU*, 2002.

[18] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *SOSP*, 2007.

[19] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, 2006.

[20] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *USENIX Security*, 2006.

[21] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys*, 2006.

[22] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *HotOS*, 1997.

[23] C. Ho, R. van Renesse, M. Bickford, and D. Dolev. Nysiad: Practical Protocol Transformation to Tolerate Byzantine Failures. In *NSDI*, 2008.

[24] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM CCS*, 2003.

[25] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, 2007.

[26] L. Lamport. The part-time parliament. In *ACM TOCS*, 1998.

[27] J. Pool, I. S. K. Wong, and D. Lie. Relaxed determinism: Making redundant execution on multiprocessors practical. In *HotOS*, 2007.

[28] F. Qin, J. Tucek, Jagadeesan, Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies – a safe method for surviving software failures. In *SOSP*, 2005.

[29] T. Rabin. A simplified approach to threshold and proactive RSA. In *CRYPTO*, 1998.

[30] J. Reynolds, J. Just, E. Lawson, L. Clough, R. Maglich, and K. Levitt. The design and implementation of an intrusion tolerant system. In *Foundations of Intrusion Tolerant Systems*, 2003.

[31] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *SOSP*, 2001.

[32] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine Faults in Database Systems using Commit Barrier Scheduling. In *SOSP*, 2007.

[33] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. In *DSN*, 2001.

[34] J. Xu, Z. Kalbarczyk, and R. Iyer. Transparent runtime randomization for security. In *SRDS*, 2003.

[35] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *SOSP*, 2003.

[36] A. R. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping applications from spilling the beans. In *NSDI*, 2007.

[37] Y. Zhou, D. Marinov, W. Sanders, C. Zilles, M. d'Amorim, S. Lauterburg, R. M. Lefever, and J. Tucek. Delta execution for software reliability. In *HotDep*, 2007.

Notes

[1]Xen [3] runs a special management VM (called Dom0) that hosts device drivers. To improve the fault tolerance of Dom0, Dom0 can also be replicated.

[2]Note that other possible architectures exist, e.g., using an unvirtualized secure operating system such as SELinux or Singularity, and running process-level diversification to the application replicas within regular isolated processes. Here we concentrate on a virtualization-based approach, for concreteness.

[3]DMA-capable devices are hardware, but they are typically configured by software.

[4]Of course, this works only if the replicas implement the *correct* application semantics; there is no amount of diversification that can make an application that implements the wrong protocol appear to implement the right protocol.