Remote Repair of Operating System State Using Backdoors[*]

Aniruddha Bohra[†], Iulian Neamtiu[‡], Pascal Gallard[∓], Florin Sultan[†], and Liviu Iftode[†]

[†] *Department of Computer Science*
Rutgers University,
Piscataway, NJ 08854-8019
{*bohra, sultan, iftode*}*@cs.rutgers.edu*

[‡] *Department of Computer Science*
University of Maryland,
College Park, MD 20742
neamtiu@cs.umd.edu

[∓] *IRISA / INRIA Rennes*
Campus Universitaire de Beaulieu,
35042 RENNES Cedex - France
Pascal.Gallard@irisa.fr

Abstract

Backdoors is a novel system architecture that enables remote monitoring and recovery/repair of the software state of a computer system without using its processors or relying on its OS resources. We have implemented a Backdoors prototype in the FreeBSD kernel using Myrinet NICs for remote access to the target machine. In a previous paper, we have shown how Backdoors can be used for recovery of "good" OS and application state from a failed system on other healthy systems.

In this paper, we describe how Backdoors can be used to detect and repair damage to the OS state of a computer system. We present two case studies of remote repair of an OS subject to resource depletion (fork bomb and memory hog) to the point where it cannot perform useful work and local repair is impossible. We show that our prototype detects OS resource exhaustion efficiently and it successfully repairs the affected system.

1. Introduction

Self-healing and recoverability from events that impair the functionality of a system have become more and more the focus of systems research [7, 1, 11, 8]. This trend reflects a shift from raw performance towards intelligent, self-manageable computer systems, driven by recent industry initiatives [3].

Remote healing [11] is a novel approach to system-level survivability in which monitoring a system for detection of exceptional events (failure, damaged state, attacks, policy violations, etc.) and recovery/repair actions are performed remotely from another system. The physical separation of the ”healing system” from the ”healed system” achieves robust detection and reaction to anomalies. To enable remote healing, we have proposed *Backdoors* (BD), a system architecture that combines hardware, firmware and OS extensions to support automated observation and intervention (through monitoring, diagnosis, state recovery and repair) on a computer system, *i)* even if its capabilities have been severely compromised, and *ii)* without overhead during its normal operation. BD uses specialized, intelligent network interfaces for *nonintrusive* remote access to the resources of a computer (memory, I/O devices, etc.), i.e., without involving its processor(s) or relying on its OS resources.

Existing "healing-from-within" approaches cannot solve problems like depletion of system resources, system-hang failures, or corruption of the software state of an OS subsystem. Such situations render a system unavailable, to the point where it cannot execute any useful work and it becomes impossible to perform monitoring, correctly diagnose the problem and/or execute automated repair actions from within the impaired system. In these cases, where healing-from-within fails, remote healing can still ensure accurate monitoring, correct diagnosis of the problem, and can take corrective actions to bring the system back to normal operation (or as close as possible). A BD-based remote healing system can either extract good state from a failed system and recover it on another healthy machine, or can perform in-place state repair towards restoring the system. In a previous work [12], we showed how Backdoors can be used to *recover* the state of Internet service sessions from failed server nodes and reinstate them on other healthy systems.

In this paper, we address the problem of automated remote monitoring and *in-place repair* of the OS state of a computer system. Damage to OS state may include effects of corruption in OS data structures (e.g., a corrupted file system) but is not limited to these. We regard the state of an OS as damaged when a certain OS subsystem is impaired and cannot perform its normal functions. The damage can have various causes: attack, system misconfiguration, OS bugs triggered under load or other exceptional conditions, resource exhaustion due to a runaway user program or to heavy load on a computer used as a server, etc. We describe

[*] This work is supported in part by the National Science Foundation under NSF CCR-0133366.

a BD-based system that enables automated monitoring, diagnosis and repair actions on a system even when that system is crippled by bad OS state. Using our system, a monitor machine can observe the state of a target machine and take repair actions when it detects exceptional conditions.

To support remote repair through BD, a target OS provides: *(i)* a *Sensor Box* (SB), a novel OS mechanism defined as a structured collection of meter variables (sensors) that track the state of OS subsystems in terms of health, liveness, performance, etc., *(ii) externalized state* consisting of fine-grained regions of OS memory accessible for remote read, and *(iii) repair hooks*, regions of OS memory that control the execution of the OS and which can be written remotely. A remote monitor process samples the SB of a target machine, detects exceptional events signaled by sensors and dispatches calls to specialized user-defined remote healing modules. The remote healing modules perform fine-grained diagnosis using the externalized state of the target OS and, if a problem is identified, perform repair using the repair hooks.

We have implemented a prototype of the system in the FreeBSD kernel, using Myrinet NICs to implement the BD. We describe two case studies of OS state repair: remote memory reclamation from memory hogging processes, and remote process table repair after a fork bomb. We show that our system detects damage fast and with low overhead, and that it effectively recovers the affected machine.

The remainder of the paper is structured as follows. Section 2 describes the BD idea. Section 3 presents the design of our BD-based remote repair system. Section 4 describes its implementation. Section 5 presents two case studies in using the system. Section 6 discusses security issues in using BD for OS state repair. Section 7 presents an experimental evaluation. Section 8 reviews related work. Section 9 concludes the paper.

2. The Backdoor Architecture

Backdoors (BD) [11] is a system architecture for nonintrusive remote healing of computer systems. BD combines hardware and software mechanisms to enable accurate monitoring and effective healing actions on a remote machine, without using its processors. The core of the architecture is the *backdoor* - a programmable NIC placed on the system I/O bus that connects to backdoor NICs of other computers through a network fabric/backplane. The backdoor NIC can access local system resources (memory, I/O devices, etc.) without involving the system processors and can initiate remote access operations through other backdoor NICs. A BD architecture requires operating system support for remote monitoring and healing actions. To ensure mutual exclusion between remote and local accesses to OS state, the BD provides *remote locking* operations.

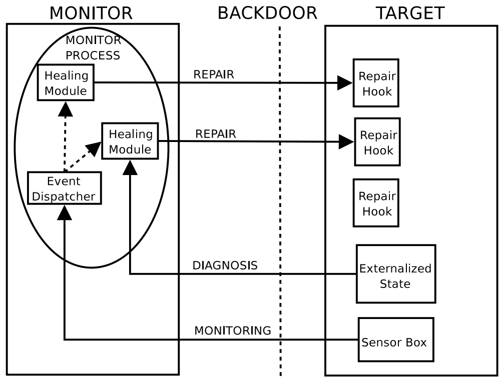

Figure 1. Software architecture for remote repair using Backdoors.

In [11] we showed that remote memory communication (RMC) can be used to implement basic BD operations. RMC defines remote DMA (RDMA) primitives that allow external access to the memory of a system to bypass its processor(s). With RDMA write, a sender can write into a remote memory buffer without remote processor intervention. With RDMA read, a receiver can initiate a transfer from a remote memory buffer without involving the remote OS or processors. Both RDMA primitives are included in industrial standards [2, 4] and are implemented in modern RMC controllers [5].

To support remote repair through BD, an OS must provide access to the state of the system, both for detection of bad state and for performing repair actions on the OS or the applications running on it. The remote access interface may define OS abstractions specifically designed for external observation of the OS state of a target system, or may enable remote read/write memory access to native OS data structures to be used in diagnosis and repair.

The system described in this paper uses a BD along with three OS extensions: a specialized OS abstraction for remote monitoring, externalized OS state for accurate diagnosis, and OS repair hooks for state repair.

3. Remote Repair with Backdoors

Figure 1 shows a BD-based remote healing pair consisting of a monitor machine (M) and a target machine (T). A specialized monitor process running on M observes the state of T and identifies exceptional conditions (monitoring). It then determines the problem that affects the target system (diagnosis) and performs remote repair operations on it (repair). All these operations (shown as labeled horizontal arrows in Figure 1) are performed through a BD.

To enable low overhead monitoring, the target OS allocates a structured region of memory called *Sensor Box* (SB) where monitored entities (OS or application processes) define progress meters or health indicators called sensors. The target OS provides *externalized state* for validation of exceptional events and problem diagnosis by enabling remote read access to fine-grained OS data structures. It also enables remote writes to *repair hooks* (RH) in the OS to allow state repair. Repair hooks can be regions of OS memory that control the execution behavior of the monitored system (e.g., configuration variables), raw OS data structures (e.g., file, process, inode), etc.

The monitor process consists of an event dispatcher (ED) and one or more healing modules (HMs). The ED retrieves the SB (periodically or on demand) from the target and performs an efficient detection of exceptional events. On detection, it invokes the appropriate healing module to handle the event. The healing module validates the event and identifies the problem, possibly using extra information from the externalized state of the target. If it determines that healing is necessary, it then carries out the repair action through the RHs. The repair action may involve remote modification of target OS data structures, manipulation of control variables, shutting down the target system, or even overwriting a clean system image on the target system. It is important to note that the BD architecture only defines the basic infrastructure, on top of which system specific monitoring, detection and healing modules can be implemented.

3.1. Monitoring with Sensor Box

A *Sensor Box* (SB) is a structured collection of records called *sensors*, allocated in the OS memory of the target system. The monitor retrieves the SB to create a local view that reflects the state of the monitored system.

A sensor is a tuple $< ID, C, T, V >$, where ID is a globally unique identifier, C is a class of sensors it belongs to, T is a threshold that depends on the class of sensors, and V is a scalar (the actual sensor). A monitored entity (e.g., OS subsystem) updates the sensor value V and defines the initial value of T.

We define three classes of sensors based on their functionality and detection properties:
(i) Progress sensors: These are monotonically increasing counters that indicate the "liveness" of the system. The monitored entity defines a deadline for updates. Failure to update the counter value within the deadline indicates an exceptional event. Examples of progress sensors are: number of interrupts raised by a hardware clock with the clock time period as the deadline, number of context switches in the system with the time quantum as the deadline, etc.
(ii) Level sensors: These are counters that account for resource utilization in the system. If the sensor value exceeds the threshold, an exceptional event is detected by the monitor process. Examples of level sensors are: number of processes in a system with a limit on the maximum number of processes, number of wired pages in system memory with a limit on the maximum number of such pages, etc.
(iii) Pressure Sensors: These are counters that are incremented at the monitored system upon occurrence of certain events. The threshold sets a limit on the number of event occurrences. The monitor process detects an exceptional condition if the number of times the event occurs exceeds the threshold. Examples of pressure sensors are: the system could not allocate memory, the file descriptor table in the system is full, etc.

Upon creating a sensor, a monitored entity specifies the global identifier ID, the type of the sensor C, and a threshold T. The monitored entity must cooperate with a monitor by modifying V for its associated sensor(s). This establishes a *contract* between monitor and monitored. The monitored commits itself to updating V, according to the type of the sensor, by increasing its value at intervals smaller than T (progress sensors), by tracking the value of the measured quantity (level sensors), or by incrementing its value if an exceptional condition is detected locally by the monitored entity (pressure sensors). The monitor commits itself to retrieving the sensor and comparing V and T. It detects a violation of the contract if V has not been updated within T time units for progress sensors, and if V exceeds T for level and pressure sensors.

The SB is accessed from the target (locally), and by the monitor (remotely) using a simple interface:

```
sensor = new_sensor(ID, C, T)
set_sensor_value(sensor, value)
sb_view = fetch_sb(nodeID)
```

where ID is the globally unique identifier for the sensor, $\mathcal{C}$ is the class of the sensor (progress, level, or pressure sensor), and $\mathcal{T}$ is the threshold value. On the target system, `new_sensor()` creates a new sensor and `set_sensor_value()` is used to update its counter value. On the monitor, `fetch_sb()` creates a local copy `sb_view` of the SB of a monitored node identified by `nodeID`.

3.2. Diagnosis and Repair

The SB provides an efficient and lightweight, but coarse-grained mechanism for detection of exceptional conditions. To decide whether healing is required and to accurately diagnose the problem may require more fine-grained knowledge of the target system state. To achieve this, a monitored system enables remote read access to a part of its OS state. The monitor performs a fine-grained inspection of the externalized state to detect anomalies.

The specialized monitor process consists of an event dispatcher (ED) and one or more user-defined healing modules (HMs). The monitor associates an HM with one or more sensors using the following interface:

```
hm = new_hm(hm_handler)
register_sensor(hm, sensor)
```

The ED performs detection of exceptional conditions using the SB. It retrieves the SB of the monitored node (periodically, or on demand) to create a local SB view, which it uses to identify exceptional conditions, and dispatches calls to HMs associated with the sensors involved.

An HM is a user defined plug-in that performs fine-grained detection, diagnosis and repair actions on the target system state. HMs are opaque to the ED, which only maintains the association between sensors and HMs. There may be multiple sensors associated with one HM, and one sensor may be associated with multiple HMs.

Upon detecting an exceptional event signaled by a sensor, the ED invokes an HM to perform fine-grained detection and diagnosis. For this purpose, the HM may retrieve externalized state from the monitored system to validate the event and to diagnose the problem before invoking the repair action.

After diagnosis, an HM identifies the corrective measures needed to bring the damaged OS back to normal operation and performs remote repair actions on it using one or more repair hooks (RHs). RHs are defined by the target system OS as regions of OS memory to which remote write access is enabled. The actual specification of a hook depends on the domain of its intended action. For example, a file system repair hook may enable access to in-memory superblock and inode blocks, a process table repair hook may enable access to fields in the process structure that control behavior of a process (priority, signal handling), a system corruption repair hook may enable overwriting the system image to bring the system back to a trusted clean state, etc.

4. Implementation

We have implemented a BD prototype in the FreeBSD 4.8 kernel, using Myrinet programmable NICs [6]. For remote monitoring and repair, we modified the Myrinet GM 2.0 library to provide in-kernel remote read/write operations between monitor and target machines. Remote access is enabled by registering the kernel memory with the NIC and dynamically updating virtual-to-physical mappings when needed.

To ensure consistent remote access to in-kernel data structures, we implemented a *remote OS locking* mechanism that blocks execution of system calls and interrupt handlers on the target machine. If the target machine is stuck in a critical section while holding the OS lock, the monitor waits for a timeout and then acquires it by brute force.

We have implemented the SB mechanism in the OS kernel. The event dispatcher is implemented as a user-space daemon. The healing modules are user defined plug-ins for the event dispatcher, implemented as dynamically loadable libraries. The SB interface is implemented as a pseudo-device accessed both from the kernel (at the target, for progress reporting) and from user space (at the monitor, for sampling the remote SB).

5. Case Studies

We illustrate the remote repair mechanism with two OS resource exhaustion scenarios in which traditional techniques *(i)* fail to prevent a system from becoming unavailable due to resource exhaustion, and *(ii)* cannot repair the system. We describe each scenario, show why the traditional mechanisms fail, and describe a remote healing solution.

5.1. ForkBomb: Process Table Repair

A forkbomb is a process that recursively spawns new processes, without doing useful work, until the resources on the system are exhausted. A forkbomb hogs the CPU of the system and does not allow other processes to execute. It also causes the process table in the OS to fill up, preventing new processes from being created. In addition, a forkbomb indirectly starves all processes, as the scheduler has to traverse a large list of processes to identify and update their priorities (no user or system activity is possible when the scheduler is running).

In our test system (FreeBSD), the OS protects against the forkbomb or any such runaway process by limiting *(i)* the maximum number of processes per user, and *(ii)* the maximum rate of process creation. When any of these limits is exceeded, the user is “locked out” of the system by killing all her processes, and not allowing her to create new processes. Other operating systems have similar protection mechanisms.

Such simple mechanisms provide only limited and easy to defeat protection. On a heavily loaded system, a forkbomb would exhaust the available CPU cycles before the system limits are reached, so the built-in OS protection will not work. The system is not dead, as all its hardware components and the OS are functioning correctly, but cannot do any useful processing. Obviously, the system is also inaccessible for repair through the console or the network since new processes (at least the shell) are required to execute any repair task. The forkbomb also prevents any existing watchdog processes, e.g., daemons, from repairing the system by starving them of CPU cycles.

In contrast, in a BD-based system we can efficiently detect and eradicate a forkbomb. To achieve this, we use two level sensors: *(i)* NPROCS, the number of processes in the system, and *(ii)* MAXPROCRATE_PERUID, the maximum rate at which a user spawns processes. The monitored system enables remote access to its process table as externalized state, and defines RHs with the signal masks of all existing processes.

Monitoring and detection. The monitor process registers a Proc_Repair_Healer healing module with the event dispatcher and associates the three sensors with it. The ED performs the coarse-grained detection using the sensors and passes control to the Proc_Repair_Healer when a forkbomb is suspected. The Proc_Repair_Healer HM retrieves the remote process table and creates its local view.

Diagnosis. Proc_Repair_Healer uses three basic policies (which can be combined to define more complex policies) to identify a forkbomb on the monitored machine:

POLICY_TOO_MANY_PROCS: The HM traverses the process table to identify the user with the largest number of processes as the culprit.

POLICY_RATE_TOO_HIGH: The HM traverses the process table to identify the user who creates processes at a rate higher than the threshold as the culprit. This policy prevents a malicious user from avoiding the hard limits imposed on the number of processes per user by creating short lived processes at a high rate.

POLICY_TREE_TOO_DEEP: The HM traverses the process table to identify the user whose process tree depth exceeds a threshold. The child of a process at depth n is defined to have depth $n + 1$. A deep process tree is generated when a process recursively spawns child processes. With this policy, the user with maximum process tree depth is identified as the culprit.

Repair. If a culprit is identified, Proc_Repair_Healer traverses the remote process table and posts a non-maskable signal (SIGKILL) to terminate all processes owned by the user found to be the culprit. The signal is posted by setting a flag (using remote write) in the process signal mask repair hook.

5.2. MemoryHog: Memory System Repair

A process or a group of processes that allocate large amounts of memory may cause the system to exhaust its memory. The virtual memory abstraction allows each processes to allocate the maximum addressable memory. Under memory pressure, unused memory is moved to a backing store for anonymous memory called swap space. We define the usable memory on the system as the sum of the physical memory size and the swap space size.

In our test system (FreeBSD), the OS limits the maximum amount of memory allocated per process. This limit cannot be too low since useful processes with a large memory footprint would be hampered. As a result, the maximum usable memory in the system can be exhausted with a small number of processes that allocate the maximum allowed memory without freeing it.

When the entire usable memory is exhausted, the OS has no alternative but to reclaim memory from processes. It calls an out-of-memory handler that chooses the process with the largest memory footprint and kills it. Unfortunately, such a brute force policy does not prevent a process that creates several child processes, each of which continuously allocate memory, from exhausting system memory. (In one experiment, with as few as 30 such processes, we were able to block any useful execution on the system.) Moreover, this random policy can choose as victim a useful process if it happens to have the largest memory footprint.

Despite built-in protection, a system running a memory hog becomes unusable since no new processes can be created. Local repair is also impossible if it requires allocation of memory - the resource that has been exhausted. In contrast, a BD-based system can accurately identify the memory hog and perform state repair.

Monitoring and detection. To detect a memory pressure situation remotely, we use a pressure sensor OOM_KILLER_RUNNING. The monitored system increments the sensor value when the out-of-memory handler starts execution. The monitored OS externalizes its process table and defines an RH with the signal masks of all processes.

The monitor process registers a Mem_Reclaim_Healer healing module with the ED and associates the sensor with it. The ED performs the coarse-grained detection using the sensor and passes control to the Mem_Reclaim_Healer if it detects memory pressure. The Mem_Reclaim_Healer retrieves the process table and creates its local view.

Diagnosis. Mem_Reclaim_Healer uses three policies (or a combination thereof) to identify a memory hog on the monitored machine:
(i) POLICY_MAX_RSS: The HM traverses the process table to identify the process with the largest memory footprint as the culprit. This policy is identical to that used by a local out-of-memory handler.
(ii) POLICY_MAX_RSSUID: The HM traverses the process list and classifies all processes according to the userid. The user whose processes have allocated the maximum amount of memory is identified. With this policy, all processes owned by that user are chosen as culprits.
(iii) POLICY_MAX_RSSUID_SAVEUID: This policy is identical to POLICY_MAX_RSSUID, but the administrator can configure a list of users whose processes are critical to the system execution and must be spared. The pro-

cesses belonging to the users in this list are filtered out from the process table before applying POLICY_MAX_RSSUID.

Repair. To reclaim memory, Mem_Reclaim_Healer posts a non-maskable signal (SIGKILL) to terminate all the culprit processes. The signal is posted by setting a flag (using remote write) in the process signal mask repair hook.

6. Security

Using Backdoors in remote repair comes at the risk of enabling access from monitor(s) to the OS memory of a target system, which makes the BD a potential access point for mounting attacks. An attacker that takes control of a monitor machine could exploit the BD interface to read sensitive information or to remotely write to the target system and irreversibly compromise it. The rudimentary access control provided by the BD by selective access to parts of the in-memory state of the target OS may not be sufficient.

To address this problem, we plan to secure the BD through a two-level solution: *(i)* access control through trusted hardware and firmware, and *(ii)* monitor replication.

The access control level will take advantage of the narrow interface offered by BD to implement protected access control and validation mechanisms in the backdoor NIC firmware, at the level of primitive operations (e.g., memory accesses). The idea is to use the intelligent NIC as a secure co-processor which executes trusted code that cannot be tampered with, and can even store secrets in a memory which is not accessible from the host system. To implement trusted and protected low-level access control, a *Backdoor Guard* can be run as part of the firmware that implements low-level access operations (remote read/write). Placing the guard in firmware and disabling access to its implementation after system initialization (even for legitimate users of the system) makes it tamper-free.

The second level of protection against attack is the functional level, by logical replication of the monitor side of the BD across different machines. In a practical system, to achieve fault tolerance of the monitoring function, the monitor side of a BD will have to be replicated on different machines. This redundancy can be exploited to also achieve resilience to attacks. Operations on the same target system issued from multiple monitors will be subject to low-level protected validation mechanisms implemented in the BD guard, before being applied to the target machine.

Remote write operations can be validated by the guard on the target side of a BD through a *delayed write agreement* protocol implemented in the BD firmware. In this protocol, a target-side guard will enforce consistency rules, e.g., write operations used to repair a system by modifying a region of its memory must produce the same values when performed from multiple monitors. The target BD guard (trusted, not malicious, not faulty) will monitor incoming writes for a particular memory location and only perform the write if values agree. Similarly, remote reads will only be allowed if they match in address and length. This simple all-or-none scheme specifically relies on the asymmetry of the parties involved (the possibly malicious monitors and the low-level "guard" entity on the target side of the BD). The guard is weaker, in the sense that it does not have enough intelligence to decide whether a particular operation is correct or not, but can be trusted to propagate correct operations when all or a majority of monitors are correct, according to a preset access policy.

No. of processes	Time (ms)
100	140
500	263
1000	350
1500	426

Table 1. Variation of the repair time with the number of processes in the system.

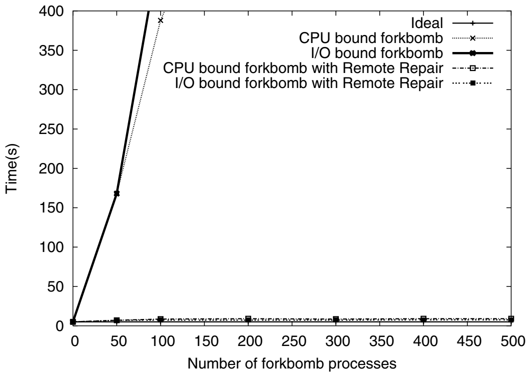

Figure 2. Variation of the execution time of a test program with number of forkbomb processes.

7. Evaluation

The goal of our evaluation is threefold. First, we show that the monitoring and repair in our system are efficient. Second, we show that a computer system can be brought down using the programs described in Section 5. We also show that the system cannot be repaired locally, i.e., either the system is unresponsive, or it terminates essential processes. Third, we show that we can detect and repair such cases using BD.

The experimental setup consists of DELL PowerEdge

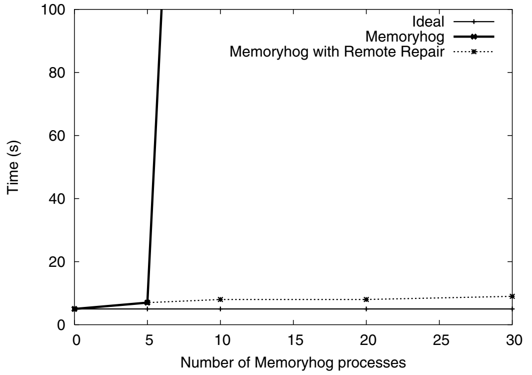

Figure 3. Variation of the execution time of a test program with number of memory hogs.

2600 2.4 GHz, 1 GB RAM dual-processors interconnected by 1 Gb/s Ethernet. Server nodes run FreeBSD 4.8 incorporating our BD prototype. The BD is implemented with Myrinet LanaiX NICs with a 133MHz PCI-X interface [6].

Monitoring Overhead. On a monitor node, the overhead includes *(i)* monitoring cost (reading the local view of the monitored SB, comparing sensor values, etc.), and *(ii)* cost of transferring the remote SB from the monitored node. To determine it, we measured the CPU usage of a monitor process while varying the sampling rate of a remote SB with 100 sensors. In the worst case (sampling the SB in an infinite loop), the CPU usage was 46%. Sampling every 10 ms (the lowest granularity of a timer), the CPU usage is about 5%, while at 100 ms it drops under 1%. This shows that fast detection can be performed with low overhead on a monitor node.

Diagnosis and Repair Cost. Diagnosis and repair involve traversing the remote process table and building a local view, identifying the culprit, and killing all its processes. The cost of repair therefore depends on the number of processes present on the target machine.

Table 1 shows the variation of the average cost of repair with the number of processes on the target system. While the repair cost grows with the number of processes, in the worst case, it takes less than half a second to execute. This shows that repair (an exceptional action) is fast, and also that it should not impose too much overhead on the monitor system.

Repair Effectiveness.

To illustrate the two case studies of Section 5 and to show that remote repair works while local repair is practically impossible, we developed two programs: a forkbomb and a memory hog.

The forkbomb creates processes that execute in a tight loop until the CPU cycles on the system are exhausted. An I/O bound version of the forkbomb program continuously reads from a pseudo-device (/dev/zero) and writes to a null device (/dev/null). This prevents the scheduler from lowering the priority of the forkbomb and of its children.

The memory hog allocates memory until it exhausts system memory and swap space. It is structured as a controller process that maintains a number of children processes, each of which allocates memory in a loop. If a child is killed by the system, the controller spawns a new process.

To illustrate the effects of the forkbomb and memory hog, we run a simple "useful" test program that executes in a loop, alone and concurrently with the forkbomb or the memory hog program. We measure the wall clock time the useful test program takes for each iteration. The results are plotted in Figures 2 and 3.

When there is no other load on the system, the time is constantly 5 seconds (the ideal time shown in Figures 2 and 3 as an horizontal line). When the forkbomb or the memory hog are executing, the useful process takes longer to receive its CPU share and the running time increases.

Figure 2 shows the variation of the wall clock time for the test program with the number of processes created by the forkbomb when *(i)* a CPU bound forkbomb executes, and *(ii)* when an I/O bound forkbomb executes. We see that the execution time grows unbounded for the forkbomb cases without repair, while it stays close to the ideal value when remote repair is performed.

Figure 3 shows the variation of the wall clock time for the test program with the number of processes created by the memory hog. Our system has 1GB of RAM and 2GB of swap space. Our test OS (FreeBSD) limits the maximum memory allocated by a process to 512MB, therefore up to 5 processes the system is well behaved and repair is not triggered. Once the memory is exhausted, the system becomes unavailable and execution time without remote repair explodes. With remote repair, the memory hog is identified, all processes with the same userid are killed and the system recovered with minimal disruption.

With around 400 processes created by the forkbomb, or with a pool of 30 memory hog processes, the test program did not complete for more than 30 minutes. In fact, we were unable to access the affected machine through the console to attempt any manual repair and we had to reboot in order to regain control over it. With remote repair, our system correctly identified the problem in all runs and was able to quickly recover the impaired machine.

8. Related Work

In [11], the remote healing concept and the Backdoors architectural vision were introduced. In [12], we described a system that uses BD to detect the failure of a computer and to recover its functionality by extracting good OS and application state from the failed system and reinstating it on another healthy machine. The system was used to dynamically recover live service sessions from server node failures in a cluster-based Internet service. This paper describes a BD architecture for automated remote monitoring, detection and *in-place repair* of the damaged OS state of a computer.

Self-monitoring was used in [9] for adapting OS behavior with the goal of increasing performance. Our system also relies on introspection by the monitored system (through the SB mechanism), but uses external, nonintrusive observation of the SB and of other OS state to detect and diagnose exceptional events in the monitored system.

Language support for automatic error detection and repair of data structures is explored in [1]. A BD-based architecture can be integrated with and leverage such support to define diagnosis/repair algorithms on OS data structures. Using a BD-based architecture in conjunction with a system like [1] can also solve its vulnerability to system faults and resource exhaustion by monitoring resource constraints and performing repairs remotely from another system.

Defensive programming [8] is a technique in which compiler-assisted program annotation is used to insert introspecting and reactive code (sensors and actuators) into application/OS code, with the goal of detecting and alleviating DoS attacks. Similarly to BD, defensive programming requires cooperation from system and/or application code through the use of minimal APIs for augmenting software functionality. Defensive programming could be used (at least in principle) to detect and react to violations of constraints on system resource usage. However, in the case of OS resources, this would require extensive changes to be integrated with the OS kernel code, including complex detection and repair policies to be statically built into the OS. Moreover, collocation of such code with the system it protects would incur CPU overhead to execute the monitoring and detection algorithms. In contrast, our system provides a simple monitoring abstraction along with support for flexible detection/repair mechanisms to be easily implemented, tested and deployed from a different system, without using resources of the target system.

K42 [10] is an OS with built-in support for component hot-swapping. While in principle hot-swapping can fix certain cases of damaged OS state (e.g., when the cause is an OS bug, the faulty OS module can be dynamically replaced with a correct one), it cannot address the more frequent situations when the trigger lies in user space (e.g., faulty or malicious user programs). Moreover, such systems are built from scratch to provide hot-swapping support. In contrast, our system uses a slightly modified general-purpose OS to provide generic support for monitoring, diagnosis and recovery of a computer system from damage to its OS state.

9. Conclusions

We have presented a prototype system that supports remote monitoring, detection and repair of damage to the OS state of a computer system. The prototype is based on Backdoors, a remote healing architecture that enables remote access to the software state of a system without using its processors or relying on its OS resources.

We have described enabling mechanisms and showed how Backdoors can be used to detect, diagnose and repair damage to OS state. We have described two case studies of remote healing of damaged OS state using our prototype: remote memory reclamation from memory hogging processes, and remote process table repair after a fork bomb. We show that our system detects OS damage fast and with low overhead, and that it effectively recovers the affected machine.

References

[1] B. Demsky and M. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proc. 1st Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

[2] D. Dunning et al. The Virtual Interface Architecture. *IEEE Micro*, 1998.

[3] IBM Autonomic Computing. http://www-3.ibm.com/autonomic/index.shtml.

[4] The Infiniband Trade Association. http://www.infinibandta.org, August 2000.

[5] Mellanox, Inc. http://www.mellanox.com.

[6] Myricom: Creators of Myrinet. http://www.myri.com.

[7] D. Patterson et al. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, Mar. 2002.

[8] X. Qie, R. Pang, and L. Peterson. Defensive Programming: Using an Annotation Toolkit to Build Dos-Resistant Software. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[9] M. Seltzer and C. Small. Self-Monitoring and Self-Adapting Operating Systems. In *Proc. of the Sixth Workshop on Hot Topics in Operating Systems*, May 1997.

[10] C. A. N. Soules, J. Appavoo, K. Hui, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, R. W. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System Support for Online Reconfiguration. In *Proc. USENIX Annual Technical Conference*, June 2003.

[11] F. Sultan, A. Bohra, I. Neamtiu, and L. Iftode. Nonintrusive Remote Healing Using Backdoors. In *Proc. 1st Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

[12] F. Sultan, A. Bohra, Y. Pan, S. Smaldone, P. Gallard, and L. Iftode. Nonintrusive Failure Detection and Recovery for Internet Services Using Backdoors. Technical Report DCS-TR-524, Rutgers University, Dec. 2003.