Remote Repair of OS 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

Abstract

Backdoors is a novel architectural approach that enables remote monitoring and recovery/repair of the software state of a 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 useful OS and application state from a failed system.

In this paper, we describe how a Backdoors architecture 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 recovers the affected machine.

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 [10, 2, 14, 11]. This trend reflects a shift from raw performance towards intelligent, self-manageable computer systems, driven by recent industry initiatives [6, 5].

Remote healing [14] 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 computer resources (memory, I/O devices, etc.), i.e., without involving its processor(s) or relying on its OS.

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 perform 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 useful 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 [15], we have shown how Backdoors can be used for dynamic recovery of Internet service sessions from server failures.

In this paper, we address the problem of automated monitoring and 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. Damage to OS state 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 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 remote monitor process can observe the state of another target machine and take repair actions when it detects exceptional

---

*This work is supported in part by the National Science Foundation under NSF CCR-0133366.
To support efficient monitoring, we have designed a novel OS mechanism called Sensor Box (SB), defined as a structured collection of meter variables (sensors) that track the state of OS subsystems in terms of health, liveness, performance, etc. A remote monitor process samples the SB of a target machine, detects exceptional events signaled by sensors in the SB, and dispatches calls to specialized user-defined remote healing modules. The remote healing modules perform problem diagnosis and remote repair actions using a subset of the OS state of the target system. To support diagnosis and repair, our BD implementation externalizes this state as fine-grained regions of the OS memory.

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 healing damaged OS state: 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 recovers the affected machine.

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

2. Backdoor Architecture

Backdoors (BD) [14] is a system architecture for nonintrusive remote healing of computer systems. The BD combines hardware and software mechanisms to enable accurate monitoring and effective healing actions on a remote machine, without using its processors. BD relies on a specialized intelligent NIC used on a computer system to enable external access to its memory, I/O devices, and other resources. BD also requires operating system support to provide remote monitoring and remote healing actions.

In [14] we showed that remote memory communication (RMC) can be used to implement basic BD operations. RMC is a technology originally developed to lower the overhead of communication by reducing OS involvement [4, 1]. RMC defines remote DMA (RDMA) primitives that allow external access to the memory of a system bypassing 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 [3, 7] and are implemented in modern RMC controllers [8, 9].

Figure 1 illustrates the RDMA read/write operations between two machines. At initialization, both endpoints notify their NICs of the memory buffers involved in RDMA using a one-time register operation. The processor of the initiator machine starts the RDMA operation (vertical solid arrow). The solid horizontal arrows show the logical RDMA data path, while the dotted line follows the physical data path. The remote NIC bypasses the host processor on the target to directly perform a remote DMA.

To support remote healing through BD, an OS must provide interfaces to the state of the system, e.g., for detecting bad state and to perform repair actions on the OS or the applications running on it. Such an interface may define OS abstractions specifically designed for external manipulation of OS state or may externalize native OS data structures to be used in diagnosis and repair. In addition, BD must provide safe remote locking operations to ensure mutual exclusion of concurrent access to the state of a live target OS.

The system described in this paper uses a BD along with an OS abstraction (Sensor Box) for monitoring the state of a target system. It also requires a target OS to externalize fine-grained OS data structures for remote read access (used for accurate diagnosis of exceptional events), and to enable write accesses to selected regions of the OS memory for state repair.

3. Design

Figure 2 shows a BD-based remote healing pair consisting of a monitor machine (M) and a target machine (T). A specialized monitor process 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 (labeled as horizontal arrows in Figure 2) are performed through a BD.

To enable low overhead monitoring, the target system 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 externalizes parts of its state for validation of exceptional events and problem diagnosis. It also
defines Repair Hooks (RH) in the OS to allow state repair. Repair hooks can be regions of 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.

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 which 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 which 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 which are incremented at the monitored system, upon detecting the occurrence of an event. The threshold sets a limit on the number of monitored events. 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 retrieve the sensor and compare $V$ and $T$. It detects a violation of the contract if $V$ exceeds $T$ for level and pressure sensors, and if $V$ has not been updated within $T$ time units for progress sensors.

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

```plaintext
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, $C$ is the class of the sensor (progress, level, or pressure sensor), and $T$ 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 access to a part of its OS state. The monitor performs a fine-grained inspection of this state to detect an exceptional event.

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

```c
hm = new_hm(hm_handler)
add_sensor(hm, sensor)
```

The ED maintains the association between an HM and sensors. The ED performs detection of exceptional conditions using the SB. It retrieves the SB from the monitored node (periodically, or on demand) to create a local SB view, and uses the SB view to identify exceptional conditions indicated by the sensors.

An HM is a user defined plugin which performs fine-grained detection, diagnosis, and executes 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.

The ED invokes an HM on detecting an exceptional condition to perform fine-grained detection and accurate diagnosis. To achieve this, the HM may retrieve additional 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. The actual specification of a hook depends on the domain of its intended action. For example, a file system repair hook may enable remote write 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), etc.

4. Implementation

We have implemented a BD prototype in the FreeBSD 4.8 kernel, using Myrinet NICs [9] with Myrinet GM 2.0 for remote access. For remote monitoring and state extraction, we used in-kernel RDMA read operations between the machines involved in monitoring and repair. Remote access is enabled by registering 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, and the healing modules are user defined plug-ins for the event dispatcher. Healing modules are implemented as dynamically loadable libraries.

The SB interface is implemented as a pseudo-device accessed both from the kernel (at the monitored node, for progress reporting) and from user space (at the monitor, for sampling the remote SB).

5. Case Studies

We illustrate the remote repair mechanism in scenarios where traditional techniques fail to prevent the system from becoming unavailable due to resource exhaustion, and cannot repair the system. We describe two such situations, 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 on the system. The fork bomb starves all processes indirectly, as the scheduler has to traverse a large list of processes, to identify and update their priorities. No user or other 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.

On a heavily loaded system, a forkbomb may never reach the system limit, 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. In this scenario, the system is inaccessible through the traditional channels (console, or network) as new processes (at least the shell) are required to execute any repair task. The forkbomb also prevents the existing processes, e.g. daemons, from repairing the system by starving them of CPU cycles.

However, the BD based system remotely detects and eradicates a forkbomb. It uses two level sensors: (i)NPROCS, the number of processes in the system, (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 HM (Proc_Repair_Healer) with the event dispatcher, and associates the three sensors with it.
The ED does the coarse-grained detection using individual sensors, and passes control to the Proc_Repair_Healer. The Proc_Repair_Healer retrieves the remote process table and creates its local view.

**Diagnosis**

Proc_Repair_Healer uses three basic policies to identify a forkbomb on the monitored machine: (i) POLICY_TOO_MANY_PROCS, (ii) POLICY_RATE_TOO_HIGH, and (iii) POLICY_TREE_TOO_DEEP. These can be combined to define more complex policies.

- 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 recursively.
- 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 RMC write) in the process signal mask.

### 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 process 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 as useful processes with a large memory footprint would be hampered. However, 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 to choose the process with the largest memory footprint, and to kill 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 prevent any useful execution on the system. Moreover, this policy can eliminate a useful process on the system if it has the largest memory footprint.

Although the system is alive, it is unusable, as no new processes can be created. Any local repair is also impossible if it requires allocation of memory as the resource has been exhausted. This prevents the administrator from carrying out any additional repair on the system. In contrast, our BD based system identified the memory hog, and repaired the system.

**Monitoring and Detection**

To repair such a system with remote healing, a pressure sensor (OOM_KILLER_RUNNING) is used. The target node increments the sensor value when the out-of-memory handler starts execution. The monitored externalizes its process table, and defines an RH with the signal masks of all processes.

The monitor process registers an HM (Mem_Reclaim_Healer) with the Event Dispatcher, and associates the sensor with it. The ED does the coarse-grained detection using the individual sensors, and passes control to the Mem_Reclaim_Healer. The Mem_Reclaim_Healer retrieves the process table and creates its local view.

**Diagnosis**

Mem_Reclaim_Healer uses three policies 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 the local OOM handler policies.
- (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 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 saved. The processes 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 RMC write) in the process signal mask.
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. Because repairing may require writes to critical data structures in regions of OS memory, an active attacker that takes control of a monitor machine could remotely write to the target system and irreversibly compromise it. The rudimentary access control already supported at 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 prevent attacks by taking advantage of the narrow interface at the BD. The BD can be made immune to attack by implementing protected access control and validation mechanisms in the BD network interface firmware, at the level of primitive operations (remote write 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 the guard 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 of the BD to attacks. Repair operations from multiple monitors on the same target system will be subject to low-level protected validation mechanisms implemented in the BD guard as described above, 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. 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 value is correct or not, but can be trusted to propagate correct values when all or a majority of monitors are correct, according a preset access policy.

### 7. Evaluation

The goal of our evaluation is three fold. First, 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 like an application server. Second, we show that we can detect and repair such cases using BD. Third, we show that the monitoring and repair in our system are efficient.

The experimental setup consists of DELL PowerEdge 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 [9].

#### 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 failure 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 node. Table 1 shows the variation of the average cost of repair with the number of processes on the target node. 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 test 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. A variation of this forkbomb continuously reads from a pseudo device (/dev/zero in FreeBSD) and writes to a NULL device (/dev/null in FreeBSD). This prevents the scheduler from lowering the priority of the forkbomb and of its children.

The memory hog exhausts the system memory and swap space. It has a controller process, and a variable number of child processes. The controller creates the children which allocate memory in a loop. If a child is killed by the system, the controller spawns a new process.

To illustrate the effect of the above two programs, we run a simple process that executes in a loop, measuring the time it takes for each iteration. The user time of this process is constant through all experiments. However, when the forkbomb or the memory hog are executing, the program takes longer to get its share, increasing the wall clock time measured on the system. The ideal time in Figures 3 and 4 is the time the program takes to execute when there is no other load on the system (5s).

Figure 3 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 4 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, and all processes with the same userid are killed. The system is 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 failed to access the system for repair, and we had to reboot to regain control over the machine. With remote repair, the system correctly identified the culprit and recovered in a very short time.

8 Related Work

Self-monitoring was used in [12] 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 [2]. 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 [2] can also solve its vulnerability to system faults and resource exhaustion by monitoring resource constraints and performing repairs remotely from another system.

Defensive programming [11] 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 soft-
ware 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 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 [13] 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 recovers the affected machine.

References