# Architectural Framework for Supporting Operating System Survivability *

Xiaowei Jiang and Yan Solihin
North Carolina State University
{xjiang, solihin}@ece.ncsu.edu

## Abstract

*The ever increasing size and complexity of Operating System (OS) kernel code bring an inevitable increase in the number of security vulnerabilities that can be exploited by attackers. A successful security attack on the kernel has a profound impact that may affect all processes running on it. In this paper we propose an architectural framework that provides survivability to the OS kernel, i.e. able to keep normal system operation despite security faults. It consists of three components that work together: (1) security attack detection, (2) security fault isolation, and (3) a recovery mechanism that resumes normal system operation. Through simple but carefully-designed architecture support, we provide OS kernel survivability with low performance overheads (< 5% for kernel intensive benchmarks). When tested with real world security attacks, our survivability mechanism automatically prevents the security faults from corrupting the kernel state or affecting other processes, recovers the kernel state and resumes execution.*

## 1 Introduction

Computer systems are increasingly becoming more interconnected and being used for critical computation tasks. Along with such changes, more security attacks have appeared that exploit software vulnerabilities and design errors. Attacks are targeted towards user applications, as well as the Operating System (OS). The OS *kernel* is a fundamental software layer between the hardware and user applications, which serves to provide machine abstraction and security protection for user applications. As the kernel source code grows in size and complexity, it provides a fertile ground for security vulnerabilities [28]. In addition, the number and variety of kernel modules are growing. It is estimated that kernel modules represent 70% of the Linux kernel code [31]. One primary type of modules is device driver. Quite often these device drivers are programmed with performance and compatibility as the main goals, without sufficient attention to security and reliability [2, 4]. As a result, the security vulnerabilities of the kernel are quite prevalent and many new ones are continuously discovered [5].

A successful kernel security attack has a profound impact. The kernel runs at the highest processor privilege level and can manipulate hardware resources freely. Therefore, a successful kernel attack exposes the entire system to malicious intruders. Even an unsuccessful attack oftentimes crashes the kernel. Kernel crash is highly undesirable because of the damages it causes, such as the termination of all running processes and a long period of unavailability.

Due to vulnerabilities of the OS and the dire consequences of successful security attacks on it, researchers have for a long time studied ways to provide survivability for the OS or the system. Survivability is the ability to operate in the presence of security faults [11]. It seeks to detect the occurrence of an attack, isolate the fault, and recover from it. Hence, a survivability scheme consists of three components that work together: (1) security attack detection mechanism, (2) security fault isolation, and (3) recovery mechanism that resumes normal system operation.

Supporting survivability is challenging. One reason is that it cannot be trivially constructed from combining a security scheme and a recovery scheme, due to several factors. First, many security protection mechanisms are simply incompatible with survivability. Many of them were originally designed for protecting a user application and not the kernel. In a user program, the goal of an attack is typically to alter the program control flow or reveal sensitive data, rather than to crash the program. Indeed, the goal of many security protection mechanisms is to convert control flow hijacking attempts into application crash [8, 18, 32], since application crash is relatively acceptable than hijacking or information loss. Attacks on the kernel often have a goal of crashing it, so if the kernel crashes as a result, the attackers have achieved their objective.

Secondly, although there are abundant recovery proposals for recovering from hardware faults [7, 10, 25, 27], they are not capable of recovering from security faults. While they provide the ability to save and recover state efficiently, the recovery process is simple: rolling back the program execution and identical re-execution of the program. In secu-

*This work was supported in part by NSF Award CCF-0347425. Much of the work was performed when Jiang was a PhD student at NCSU.

rity attacks, such identical re-execution will simply replay the attack. Hence, kernel survivability needs non-identical re-execution, in which the kernel is restored into a "good" state but the fault has to be isolated before re-execution. In addition, traditional fault recovery schemes treat the system as consisting of a single state. However, the kernel state consists of states from multiple user processes, hence these states must be separated from each other.

The contribution of this paper is an efficient architectural framework for supporting OS survivability. Our scheme includes three components: attack detection, fault isolation, and recovery mechanisms. A malicious (or attacked and subverted) application interacts with the kernel via system calls or exception/interrupt handlers. Hence, we structure our scheme around system call/exception handler boundaries. To support recovery, a checkpoint is created prior to the kernel serving a system call or an exception. We distinguish the kernel state of different user processes and the checkpointed state is specific to a user process. If an attack is detected, the kernel state is rolled back to the previously created checkpoint. Since a checkpointed kernel state is user process-specific, the recovery does not affect the progress of other user processes. To avoid the repeat of an attack, after the kernel state is rolled back, the user process involved is suspended, and the kernel is resumed while the system call used for the attack is skipped.

Our survivability scheme also includes a security attack detection mechanism. Attack detection capability determines the level of security coverage the survivability scheme can provide. As many existing attack detection mechanisms can be easily integrated into our survivability scheme, high security coverage comes at the cost of high performance overhead. In this paper, we propose a low-overhead attack detection mechanism as an example to protect against the most prevalent type of kernel attacks (i.e. contiguous overwrite attacks). We uses word-granularity write protection with architectural support. While it detects contiguous overwrite attacks, it is also served as a fault isolation mechanism to isolate the faulty process from corrupting the state of other processes.

Overall, our OS kernel survivability scheme has the following features:

1. *Low performance overhead.* Kernel is performance sensitive. We achieve low performance overhead through efficient architecture support.
2. *Automatic*. The recovery from security attacks should not be conditional upon human intervention because a long period of unavailability may be the goal of an attack. In our design, kernel recovery is automatically triggered when a security attack is detected, and the detection and recovery events are logged and users alerted immediately.
3. *No recompilation*. Because kernel source code is not always available, and hence the ability to recompile kernel cannot be assumed. In our design, checkpoint creation and deletion, protection manipulation, and state rollback are achieved through dynamic instrumentation of the kernel.

We evaluate the performance and efficacy of the proposed scheme on a full system simulator running the Linux OS. We test with a range of real world kernel attacks, and confirm that all the attacks are detected, and the OS survives all the attacks and keeps its normal function intact. Although the hardware support we propose is quite simple, the average and worst-case performance overheads due to our survivability mechanism are low, at only 2.1% and 4.7%, respectively.

The rest of the paper is organized as follows: Section 2 discusses related work, Section 3 describes the attack model and assumptions that we use. Section 4 discusses our kernel survivability scheme in details. Sections 5 describes our evaluation environment while Section 6 discusses evaluation results. Finally, Section 7 summarizes our conclusions.

## 2 Related Work

Many hardware fault tolerance methods have been proposed, such as [7, 10, 25, 27]. However, as fault tolerance techniques simply result in repeat of attacks and do not distinguish multiple kernel states in their recovery process, they are not directly applicable to survivability techniques. Our scheme is different than hardware fault tolerance mechanisms in that we distinguish multiple kernel states and the recovery process prevents security attacks from being replayed. There are also mechanisms to tolerate software faults such as device driver failures [29]. However, they do not protect the kernel itself, and rely on identical re-execution, so the recovery process could repeat security attacks if they cause the failures.

Researchers have also proposed many attack detection and avoidance schemes [12, 14, 18] in the past to protect user-level applications. However, they do not provide the ability to recover from a security attack. Many of them simply convert an attack into a crash, which is incompatible with survivability. There are also studies on fine-grain protection such as Mondrian Memory Protection (MMP) [30]. MMP divides the memory into different protection domains, where permission control bits denote the region that a program or kernel module can operate on. While MMP improves kernel security by providing fault isolation, it is inherently a detection scheme rather than survivability scheme, and does not provide recovery or isolation from security faults. Security faults could still occur within a single protection domain and when they do, the kernel cannot recover from them.

There has long been research in survivability [6, 21, 22]. Unfortunately, current survivability techniques are expensive and/or not sufficiently robust. They primarily rely on a

replica approach, in which multiple non-identical or diverse instances of a system or a program are employed. The drawbacks is that running *N* replicas require *N* times as many resources. Besides being a very expensive approach, a more fundamental weakness is that the efficacy of the approach relies on the assumption that replicas do not share the same vulnerabilities. In practice this assumption can be hard to fully achieved [11]. Independent software teams still tend to make similar wrong assumptions, leading to common bugs, while naturally diverse systems may turn out to rely on the same (vulnerable) libraries.

Finally, Recovery Domains [19] is a recovery-based scheme that can recover the kernel from unexpected runtime error. As a software scheme, it slows down the normal attack-free execution by between 8% to 5.6×. In addition, it is not aware of the dependencies between faulty and non-faulty processes while isolating faults. In contrast, our scheme provides efficient architecture support and only incurs less than 5% performance overhead. While recovering from security faults, our scheme isolates the faulty processes from non-faulty ones.

## 3 Security Model and Kernel Attacks

**Security Model and Assumptions**. The goal of a kernel attack can be to either crash the kernel or to execute malicious code at the kernel's privilege level. We assume that the attacker can achieve these goals by exploiting vulnerabilities in the kernel. Attacks on the kernel can be launched remotely through attacking a vulnerable application. After the attackers gain control of the application, it can be used to attack the kernel. Attacks on the kernel can also be launched locally if attackers already have access to the machine. To cover both scenarios, we assume that the attack can be carried out through vulnerable or malicious applications. We assume that kernel and kernel modules such as device drivers are *vulnerable but not malicious*. Dealing with malicious modules is very difficult because such code already runs in the privileged mode. Fortunately, there are mechanisms to authenticate module signatures to avoid accepting modules developed by unknown parties.

There are various categories of software vulnerabilities that can be exploited by an attacker. In a *buffer overflow* vulnerability, the lack of buffer bound checking allows a long string input to overwrite adjacent bytes beyond the buffer. In an *integer overflow* vulnerability, a signed-integer value is not checked for a negative value that indicates an overflow before it is used. In a *format string* vulnerability, the format string in the `printf` family of functions is taken as an input parameter, which may result in a read from or a write to a location specified in the format string argument. Since buffer overflow occurs due to buffer copying, it typically causes a *contiguous overwrite* of bytes adjacent to the buffer. Integer overflow and format string vulnerabilities can sometimes be exploited to perform a *random overwrite*, i.e. overwrite any location chosen by the attacker.

A detection mechanism that provides comprehensive security coverage towards all categories of attacks comes at the cost of high performance overhead (e.g. an average of 22% slowdown in MMP [30]). We observed that contiguous overwrites attack indeed is the most prevalent kernel attack so far [5]. This is because random overwrite attacks are difficult to exploit in the kernel, due to several factors. First, `printk` family of functions in the kernel are mainly used for kernel debugging purposes. Hence, they tend to have hard-coded format strings rather than ones accepted from user space. This leaves little space for format string vulnerabilities. Second, due to the limited size of kernel stacks, arrays are rarely used in the kernel, leaving few opportunities for integer overflow attacks to perform array indexing.

As a result, in this paper we focus on providing attack detection mechanism for contiguous overwrite attacks. We present an efficient detection that detects contiguous overwrites before the attack contaminates beyond the buffer boundary (Section 4.2). However, if other types of kernel attacks become prevalent in the future, our survivability scheme provides the flexibility to integrate other new detection mechanisms easily.

**Target of Kernel Attacks**. Based on the memory locations that the attack tries to overwrite, we categorize kernel attacks into kernel stack tampering and kernel pool tampering. Each process has a kernel stack (e.g. 8KB in size) for holding local variables and execution context while it is executing in the kernel space. Because local variables in the stack may be adjacent to critical kernel data (e.g. `thread_info` field), a buffer overflow in this stack can cause significant damage by overwriting the critical data.

Similar to the user heap, the kernel dynamically allocates and frees data structures (e.g. *inode*) in the *memory pool* through the use of *buddy system*. Allocation units (i.e. *slab*) in the memory pool are laid out contiguously without protection in between. This leads to a vulnerability because many slabs contain buffers that receive input from the user process, and a buffer overflow in one slab can corrupt neighboring slabs that may contain critical data. Note that some of the kernel space resides in the virtual memory area (VMA). Since VMA is not used often for storing critical kernel data (mostly device driver data etc.), and also it allows a simple yet effective security protection using nonaccessible pages to surround its buffers, our scheme only provides protection towards kernel stacks and memory pool.

**Mechanism of Kernel Attacks**. A kernel attack starts from the attacker gaining control of a user application. The attacker then continues to attack the kernel through the interaction of the kernel and the application. In general, the kernel attack consists of two steps. The first step (*trap-to-kernel*) seeks to elevate the execution privilege level. This

step can be carried out through a system call, or an exception handler. In the call chain of system call routines and exception handlers, user arguments are passed along the way. The attacker attempts to attack the kernel by passing invalid or malicious arguments. Once a trap to the kernel occurs, the kernel function executes with the arguments passed by the user process. The second step (*vulnerability exploitation*) then exploits vulnerabilities or bugs in the kernel code. For example, if the kernel has a buffer overflow vulnerability, a lack of bound check may cause an overflow to bytes adjacent to a buffer in the kernel stack or pool.

```
static int elf_core_dump(long signr,
  struct pt_regs *regs, struct file * file){
  ...
  int i, len;
  len = current->mm->arg_end -
        current->mm->arg_start;
  if (len >= ELF_PRARGSZ)
    len = ELF_PRARGSZ - 1;
  copy_from_user(&psinfo.pr_psargs,
        current->mm->arg_start, len);
  ...
}
```

**Figure 1. Linux `elf_core_dump` vulnerability**

In order to give a more concrete illustration, we detail an example real-world attack [3]. Figure 1 shows the code for kernel function `elf_core_dump`, which is triggered when an executable crashes, and which dumps the process' memory image to the disk. The code copies arguments from the crashed process' stack to a kernel slab. The next line of code has a vulnerability in that while `len` is checked against its upper-bound value, it is not checked against its lower-bound value. Hence, through integer overflow, an attacker can set a negative value for `len`, and that value would then be passed to the buffer copying function `copy_from_user`, which overflows the buffer and results in a contiguous overwrite attack that overwrites the critical data it is adjacent to.

# 4 Our Solution

We now describe the components of survivability scheme in the following order: recovery, detection, and fault isolation.

## 4.1 Recovery Mechanism

To design a checkpointing and recovery mechanism, we address the questions of *when* to create a new checkpoint and *what state* a checkpoint should consist of, *when* a checkpoint can be deleted, *where* the checkpoints are stored, and *how* to perform checkpoint operations efficiently.

**Checkpoint Creation**. The checkpoint should be made when a "good" state is available, preferably just before an attack may occur. Our analysis in Section 3 indicates that entry points for attacks are system call or exception handlers. Consequently, we create a new checkpoint at the entry point to each system call and exception handler function.

**Checkpoint Components**. We should checkpoint the state that is the target of modifications in an attack. One challenge is that unlike hardware fault tolerance mechanisms, we must distinguish the kernel state of different user processes, and the checkpoint should only capture the kernel state of the user process making the system call or exception. The kernel state of a user process includes a kernel stack, and parts of the kernel pool that are written due to the system call or exception handling. Hence, our checkpoint consists of three components: kernel stack, slabs that reside in the kernel pool, and the process' execution contexts that include register values and `pid` of the process. For all these components, we need to know their ranges of addresses (*base address* and *offset*) in order to copy them into the checkpoint.

For the kernel stack, the offset is always fixed because the size of kernel stack is static (e.g. 8KB). Since the kernel stack is allocated when the process is created, its base address is always known prior to the checkpoint creation, so its address range to be checkpointed is determined statically. However, slabs are allocated dynamically, and which slab would be overwritten by system call handling is not known until a kernel function that will overwrite them is called. Hence, we must incrementally checkpoint each slab as it is discovered. To achieve this, we monitor the address ranges in the kernel pool written during system call or exception handling just before they are overwritten. One challenge is that there is a very high number of kernel functions that attempt to write to the kernel pool memory. Fortunately, we found that all of them rely on a set of common kernel library functions, such as `strncpy_from_user`. A segment of this function is shown in Figure 2. To figure out the buffer parameters passed to this function, we analyze buffer management functions and insert a breakpoint at line `[**]`. When the breakpoint is encountered in execution, we can discover the base address and offset of the buffer by looking up the values in register `edi` and `ecx`, respectively.

```
    pc:0xc0212fbd  entrance of strncpy_from_user
    ...
[**]pc:0xc0213002  lodsb
    pc:0xc0213003  stosb
    ...
    pc:0xc0213008  dec ecx
    pc:0xc0213009  jne 0xc0213002
```

**Figure 2. Code from `strncpy_from_user`, showing the breakpoint that triggers checkpoint creation (line `[**]`).**

These functions cannot be patched easily to make them secure, as they have no way to know whether the buffer address parameter they receive is valid or not. So it is up to the caller functions to supply valid address ranges to the buffer management functions. Unfortunately, there are many such callers, and some of them lack the necessary checking that make sure that they pass valid buffer ranges to the buffer management functions, such as the example illustrated in Figure 1. However, with our checkpointing scheme, even if buffer ranges passed to such functions are invalid, we always create the checkpoint of that invalid address range to undo subsequent overwrites to it.

**Checkpoint Management**. The next challenge that needs

to be addressed is *how many* checkpoints need to be kept and for *how long*. This depends on how much time the attack detection mechanism needs to detect an attack after the first kernel state modification is attempted by the attacker. To make the challenge reasonable, we need to pick detection mechanisms that give reasonable properties. Our proposed attack detection mechanism (Section 4.2) provides two essential guarantees. First, an overwrite that begins in a given kernel stack or slab is detected as soon as it extends beyond that stack or slab. Second, overwrites that occur solely within a kernel stack or a kernel slab are detected before the system call or exception handler returns control to the user process. These two guarantees allow us to limit the time and life-span of our checkpoints. The maximum number of checkpoints that need to be kept is bounded by the maximum call chain depth when servicing a system call or exception, and the checkpoints can be organized in a *checkpoint stack*. When a system call or exception handler is invoked, we push a new kernel stack checkpoint onto the checkpoint stack, and each time subsequent kernel functions attempt to write to a buffer in the kernel pool, the buffer is also pushed to the checkpoint stack. Just before the kernel handler returns to user space, an additional scan of the kernel stacks and pools involved in the checkpoint can be performed to detect an attack. If an attack is detected, we pop a checkpoint from the stack and restore its state, and repeat this pop-and-restore until the stack is empty. If no attack is detected, the checkpoint stack is simply de-allocated.

We note that the checkpoint stack is essential for ensuring survivability. Hence, it needs to be stored in memory securely. For this, a portion in the VMA of the kernel is allocated for storing the checkpoint stack. To protect this checkpoint stack from attacks or unintentional tampering, we employ two protection mechanisms. First, the checkpoint stack area is surrounded by non-accessible pages, so that any intentional or accidental buffer overflow from outside the checkpoint stack area would trigger an exception before it can write to the checkpoint stack. In addition, we write-protect the pages that keep the checkpoint stack. Only when the stack is modified this write protection is removed, and after the modification the write protection is restored.

**Dynamic Instrumentation**. We mention in Section 1 that one of the design criteria is that we should avoid OS source code modifications and recompilation. Checkpoint creation, checkpoint stack management, checkpoint deletion, and breakpoints must be weaved dynamically to an existing kernel image. To achieve that, we use dynamic interception through Kernel Dynamic Probes (Kprobes), a mechanism to register kernel execution breakpoints.

Our breakpoint handlers consist of several types. The first type is *prefix handlers* which are inserted into the system calls, exception handlers and other kernel functions that can serve as the entry points to the kernel, and are executed prior to the functions themselves. Prefix handlers are used for triggering checkpoint creations. The second type is a *suffix handlers* which are inserted into the end of system calls, exception handlers and other kernel functions that can serve as the entry points to the kernel, and are executed after the execution of the functions themselves, but before the functions return. Suffix handlers are used for performing checkpoint stack de-allocation. We also use *exception handlers* that are invoked when a detected attack raises an exception, and are used for restoring the kernel state. Finally, *PC handlers* are invoked when instructions with the specified PC are invoked, and are used for identifying buffer ranges in the kernel pool to be checkpointed.

**Architecture Support for Checkpoint Creation**. A software-only checkpoint uses loads and stores to copy the kernel state into the checkpoint area. It may have a non-negligible performance impact because checkpoint creation cannot be overlapped with regular computation (Figure 3(a)). We explore several architectural mechanisms for checkpointing to determine if they would be useful.

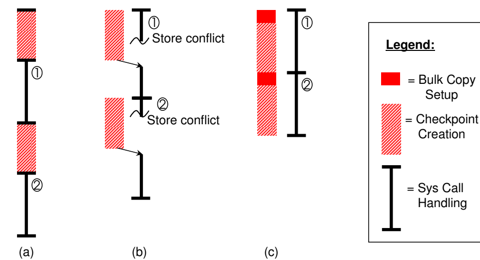


**Figure 3. Checkpoint Creation: software (a), CPTCR instruction (b), and Bulk Copying (c).**

Our first attempt is to add a `CPTCR` instruction which saves the processor context and a specified range of memory addresses. This approach saves instruction fetch bandwidth, but can be internally unrolled into loads and stores. To overlap checkpoint creation with regular computation, we add a *checkpoint address range register* (CARR), which records the range of addresses (base address and length) from which checkpoints are being created. When a checkpoint creation is initiated, its address range is recorded in CARR and the hardware begins to copy data in the background. Meanwhile, the processor can continue executing instructions. However, if the processor writes to an address that conflicts with CARR, the processor stalls until the checkpoint is completed (Figure 3(b)). Once the checkpoint creation is over, CARR is marked as invalid, and the stalled processor is allowed to resume execution. Although this scheme is promising, in practice we found that there is only a very limited overlap between checkpoint creation and regular kernel execution. In addition, a large number of memory locations which are brought into the cache regardless of the reuse pattern, which results in a waste of precious memory bandwidth [15, 16, 17, 23, 26].

To avoid this cache pollution and improve overlap, we

use the bulk copying support proposed in [33] (Figure 3(c)). Checkpointing region is split into pages and sent to the copy engine in the memory controller. A *pending copy table* in the processor is used to keep copying region and detect conflicts. The source and destination address ranges are specified to the memory controller, so it can copy from the source range to the destination range without involving the processor or polluting the cache. At the start of checkpoint creation, the system enters a *bulk copy setup* phase, in which the source and destination address ranges are written to an on-chip Copy Control Unit (CCU) for virtual-to-physical address translation. The cache controller writes back all cached lines in that address range. During this setup time, a conflicting store from the processor would result in stalling the processor. After the write-backs are completed, the system enters a *bulk copy* phase. In this phase, the copy engine located in the memory controller performs the actual copying. The processor is allowed to resume execution. However, if cache lines in the source address range are modified, which is indicated by a match to the pending copy table, the modified lines are not allowed to be written back, to ensure the integrity of the checkpoint.

## 4.2 Security Attack Detection Mechanism

One important component of our survivability scheme is the attack detection mechanism. Our scheme naturally accommodates many existing attack detection mechanisms. However, to restrain the performance overhead associated, we present a detection mechanism that deals with the most prevalent attack, i.e. contiguous overwrite attacks.

To guarantee an overwrite is detected as soon as it extends beyond the stack or slab boundary and is detected before system call or exception handling returns, we place write-protected delimiters around a kernel state to prevent an overwrite from pasting the delimiters. To minimize the storage overhead of storing delimiters and to avoid complicating the kernel memory allocators, we use word-size delimiters around each kernel slab, and word-alignment of slabs. This requires each word to have its own write protection bits. This protection would ensure that a buffer overflow beyond a kernel slab would promptly trigger an exception. To detect attacks on the stack, we simply make return addresses and other critical thread information non-writeable. To add these delimiters without modifying the kernel source code, we insert prefix and suffix breakpoints to kernel slab allocation routine (`kmalloc`). The prefix handler intercepts the requested allocation size, while the suffix modifies the pointer returned by the allocation routine to adjust for the alignment and delimiter.

In the main memory, we associate one write-protection bit per kernel state word and the bits are linearly stored in a bit vector format in the main memory. The protection bit area is allocated at the virtual memory area and since its security is critical to the correctness of our attack detection, so it is surrounded by write-protected pages, and the protection area itself has temporal write protection that is only removed when it is updated.

Stores made by the processor now need to be checked against the protection bits in order to determine if they are valid or not. Reading the protection bits in the memory for each store is obviously unacceptable. We observe that at any given time, the number of write-protected words is quite small, indicating that a small cache may be sufficient to keep most of the protection information for these words. As a result, we simply cache the address of each individual word that is write-protected in a small *protection cache*, as shown in Figure 4. A protection cache is basically a set-associative tag array for write-protected words. If there are more write-protected words than the associativity of a set, the *overflow bit* is set and the information is stored into the protection bit vector in the main memory. The protection cache is indexed by XORing all sections of the address bits of a store. The purpose of XOR-based [20] indexing is to make sure that all sets are uniformly utilized. The 30-bit tags in the indexed set are then compared to the word tag (part of the store address). A match indicates that a store to a write-protected word is attempted, hence an exception is triggered. However, when the overflow bit is one, a mismatch is quite costly as it means that the store address must be checked against the protection bit vector information in the main memory.

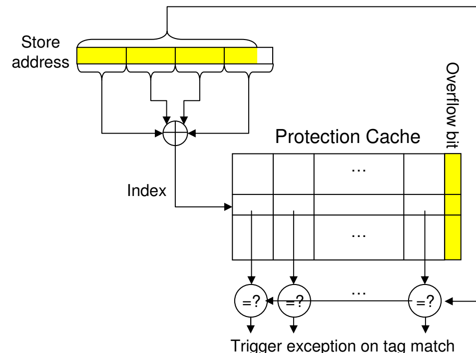


**Figure 4. Protection Cache Structure.**

To reduce the cost of a mismatch, we rely on two overflow mechanisms, as shown in Figure 5. The first overflow mechanism (Figure 5(b)) utilizes a bloom filter [9, 13] for each cache set to summarize protection information that overflows the protection cache. The protection cache's associativity is halved in order to make space for the bloom filter. On a mismatch at an overflowed cache set, the store address is matched against the bloom filter for that set. Since a bloom filter has no false negatives, a mismatch means that the store is not to an address that is write-protected. However, a match in the bloom filter may be a false positive, so the address needs to be checked against the protection bit vector in the memory.

The second mechanism keeps the protection cache.

However, on overflow, several cache lines in the set are grouped together and transformed into a bloom filter (Figure 5 (c)). On another overflow, we have two choices: store the new address in the existing bloom filter, or group other cache lines in the set to make a new bloom filter for the new overflow. On the surface, aggressively turning cache lines into bloom filters seem unnecessary. However, further mathematical analysis reveals that the combined false positive rate is minimized when we use the aggressive approach to ensure that the number of addresses kept by all bloom filters are roughly the same (detailed analysis is omitted due to space limitation.). We find that when the addresses are mapped uniformly to all bloom filters, the false positive rate increases much more slowly than when most of them are mapped to just one bloom filter. Hence, we conclude that the protection cache that converts to multiple Bloom Filter is the superior approach. When an address' write protection is removed, its address can be removed from the bloom filter without going to the main memory.

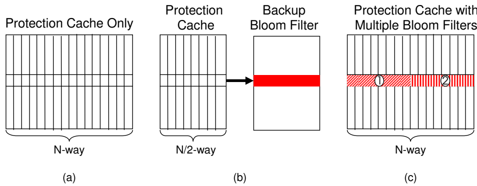


**Figure 5. Protection Cache Overflow Mechanisms: no overflow (a), Bloom Filter (b), and multiple Bloom Filters (c).**

### 4.3 Fault Isolation

As long as the detection mechanism ensures that kernel state of different user processes are isolated against each other, we can always recover the kernel state of each individual user process from security attacks. When a security attack is detected, an exception is raised and our exception handler is invoked. The handler rolls back the kernel state to the clean state prior to the system call or exception handler that causes the attack. Then the handler calls a kernel function `kill_proc` that suspends the user process by sending a `SIGSTOP` signal to the process, and invokes scheduler to continue normal operation by scheduling another process. The handler also logs the fact that an attack detection and kernel recovery have occurred. Optionally, the handler can be made to dump the memory image of the process for further analysis, and alarm users about the events.

In the recovery process, processes that have dependencies on the faulty one must be identified, since continuing their execution may result in unexpected consequences. Process dependencies can be built *explicitly* through interprocess communication (IPC) or *implicitly* via data dependencies. Our fault isolation mechanism identifies and isolates processes that have either type of dependencies on the faulty process. To identify explicit process dependencies, we instrument system calls and kernel functions that handle IPC creation (e.g. `sys_pipe`). The `pid`s of dependent processes are kept along within the checkpoint stacks. Once the kernel recovery begins, they are looked up and suspended as well. To identify implicit dependencies, one way is to track processes' data dependencies along with their execution. However, this way the incurred performance overhead can be unbounded. We found that implicit process dependencies only affect dependent process in the presence of *kernel preemptions*. If a process never gets preempted in serving its system call, even it is detected as faulty and is rolled back, its dependent processes still perceive a consistent kernel state after the recovery process. Hence, rather than tracking fine-grain data dependencies, we instrument OS scheduler to track the kernel preemptions. If an attack is detected, processes that preempt the faulty one are also rolled back and re-executed. Checkpoints maintained for processes that preempt others must be kept until the system call of the preempted process returns.

## 5 Evaluation Methodology

**Machine Parameters.** We use a full system simulator based on Simics [24] to model an out-of-order 4GHz x86 processor with issue width of 4. For L1 instruction and data caches, we assume 16KB write-back caches with 64-byte line size, and an access latency of 2 cycles. The L2 cache is 8-way and has 1MB capacity, 64-byte line size, and an access latency of 8 cycles. The memory access latency is 300 cycles (or 75ns), and we model a split transaction bus with 6.4 GB/s peak bandwidth. For the protection caching support, we use a 3.75KB structure that is either configured entirely as a protection cache, as half a protection cache and half a bloom filter, or as a protection cache that can transform into multiple bloom filters (Section 4.2). The protection cache has 16 ways and each line stores a 30-bit tag. Each bloom filter entry is 120-bit wide and has six keys.

**Benchmarks.** We use all 16 C/C++ benchmarks from the SPEC2000 benchmark suite [1] with reference inputs. Since out-of-order processor simulation in Simics can be over ten times slower than an in-order processor simulation, we first run all benchmarks with in-order processor model, and then select seven of them that show the highest checkpoint creation frequencies for further simulation with the out-of-order processor model. To stress the performance of our scheme, we add four *kernel intensive* benchmarks, including *Apache Server* with *ab* workload, a network benchmarking tool *iperf*, and two unix tools: *find* and *du*. For *find*, we use the command '`find /usr -type f -exec od {} \;`', which searches the `/usr` directory and dumps the file content. For `du`, we use the command '`du -h /usr`' to summarize the disk usage of each file in `/usr` directory. For SPEC2000, *find* and *iperf*, we simulate 1B instructions after skipping the initialization phase. For *du* and *apache*, we simulate all dynamic instructions.

**Attacks.** To validate the functionality of our survivability scheme, we use two real-world kernel vulnerability exploits. The first one is the *elf_core_dump* attack described in Section 3 which attempts to crash the kernel. The second one is a *block* device driver attack described in [4]. To add variation, we write a device driver program that is vulnerable to buffer overflow and the number of bytes overflew can be varied, and it attempts to crash the kernel. We call the last attack *drivercrash* attack.

**Operating System.** We choose Fedora Core with the Linux kernel v2.6.11. Although checkpoint creation, detection, recovery, and fault isolation are all automatic, currently breakpoint handlers are still programmed and inserted to kernel manually. Hence, we limit the kernel routines that are instrumented. To approximate realistic performance overheads, we instrument the top twenty most frequently invoked ones. They contribute to more than 90% of all system call and exception handler instances and hence provides a good coverage. In addition, we instrument eight other kernel functions which have been reported as vulnerable.

# 6 Evaluation

## 6.1 Survivability Functionality Validation

First, we validate the three attacks (*elf_core_dump*, *block*, and *drivercrash*) on an unmodified FC Linux. Both *elf_core_dump* and *drivercrash* crash the kernel, while *block* successfully escalates to kernel level privilege. This is a significant problem, because many production systems that use FC Linux are completely open to these attacks. We then boot the same Linux kernel on Simics, repeat the attacks, and observe the same outcome.

We then integrate our survivability scheme into the simulated machine. We repeat all attacks, with various overflow amounts for the *drivercrash* attack. In all these attacks, illegal overwrites are detected immediately when they exceed the bounds and write into write-protected words. Automatic recovery occur in that case and the faulty process is correctly suspended, while the kernel successfully survives the attacks and retains its normal functioning. After each recovery, we continue the simulation for eight more simulated machine hours to see if there is any sign of system instability, and we found none.

## 6.2 Benchmark Characterization

**Checkpoint creation frequency.** To determine how frequently benchmarks would cause checkpoint creations, we measure the *number of checkpoint creations per billion instructions* (NCPBI). NCPBIs of all the SPEC benchmarks and kernel intensive (*KI*) benchmarks are shown in Figure 6(a), with their separate averages. NCPBIs for SPEC2000 benchmarks have an average of 151, which corresponds to one checkpoint creation of every 6.5 million instructions. In contrast, all kernel intensive benchmarks show much higher NCPBIs compared, ranging from a minimum 8158 (find) to a maximum 21632 (du), with an average of 16516, which corresponds to one checkpoint creation for every 60 thousand instructions. Thus, due to frequent system calls, kernel intensive benchmarks create checkpoints roughly two orders of magnitudes more frequent compared to SPEC2000 benchmarks. We pay particular attention to benchmarks with high NCPBIs because they stress the performance of our survivability scheme.

**Figure 6. Number of checkpoint creations per per billion instructions (NCPBI) (a), and Average checkpoint size in KB (b).**

**Checkpoint sizes.** Figure 6(b) shows the average size of checkpoints. Since each kernel stack checkpoint is always 8KB, the variation comes from the sizes of checkpoints from the kernel pool. The figure shows that a checkpoint size is typically a few kilobytes. Combined with a checkpoint interval of every 60k instructions, kernel intensive benchmarks would incur a large checkpoint footprint if these checkpoints are cached when they are created. For the rest of the evaluation, we focus on benchmarks that stress our hardware mechanisms, namely all kernel intensive benchmarks and ones with the highest NCPBIs: gap, gcc, gzip, mcf,perlbmk, twolf, and vortex.

## 6.3 Performance Overhead

Having validated the functionality of the survivability scheme, we now turn to the performance overheads of our survivability scheme. Since the performance overheads of our scheme come from checkpoint management (mostly from checkpoint creation and very little from checkpoint stack de-allocation) and attack detection, we first show the checkpoint management and attack detection overheads separately, then show the performance overheads of combining both of them to fully support survivability.

**Checkpoint Creation Evaluation**. Figure 7 compares the execution time of NULL breakpoint handlers (*Kprobes*), software-only checkpoint creation (*SW*), and bulk copying hardware support (*BC*). *Kprobes* represent a kernel instrumented with all breakpoint handlers that we use for other

cases, but we remove the code in each handler's body, which essentially measures the performance overheads of only the dynamic instrumentation. For each benchmark, all configurations simulate a fixed number of *user* instructions plus instructions from the breakpoint handlers and checkpoint management. Unrelated kernel instructions are simulated but not counted in deciding when to stop the simulation.

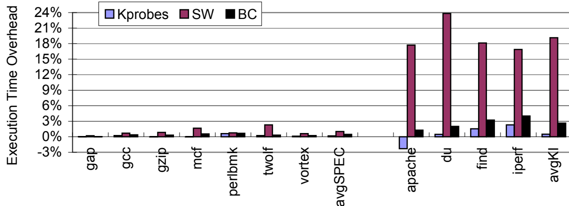


**Figure 7. Execution time overhead (IPC).**

In Figure 7, Kprobes shows largely negligible overheads compared to an un-instrumented (and unprotected) kernel. However, due to the more frequent system calls, it is clear that kernel intensive benchmarks suffer larger instrumentation overheads, with an average of 0.5% vs. 0.2% for SPEC benchmarks. The negative performance overhead of -2.3% in apache is mainly caused by the variation in context switching effect with other processes (such as OS daemons) that also run and skew our measurement slightly. With software-only checkpoint creation, the figure shows much smaller overheads in SPEC benchmarks (average of 1.0%) than in kernel intensive benchmarks (average of 19.1%). Because the software mechanism uses a loop of loads and stores to do the copying for checkpoint creation, checkpoint creation is not overlapped with regular computation. These checkpoints will not be accessed again unless there is an attack that requires kernel state rollback. Finally, our bulk copying hardware mechanism avoids cache pollution and overlaps most of checkpoint creation with regular computation. As a result, it achieves very small overheads, with an average of 1.5% for SPEC benchmarks and only 2.6% for kernel intensive benchmarks. The worst case overhead is also low (4% for iperf).

**Attack Detection Evaluation**. We now show the evaluation results for our attack detection mechanism. For comparison, we also implement Mondrian Memory Protection (MMP) to compare with our protection cache scheme. Figure 8 shows the percentage increase of memory references (or L2 cache misses) due to various attack detection mechanisms.

The *Mondrian* bars show that using MMP results in a significant increase of memory references compared to an unprotected system, especially for kernel intensive benchmarks. This is due to the insufficient spatial locality that results in a high Protection Look-aside Buffer (PLB) miss rate. Note that without bloom filters, each PLB miss must result in memory access to fill the PLB with the protection information. Furthermore, looking up protection information in memory sometimes requires multiple memory accesses to traverse the multi-level protection table.

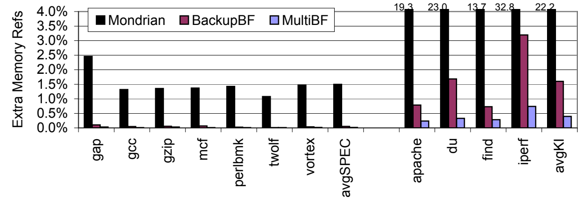


**Figure 8. Increase in memory references.**

Our protection cache with backup bloom filter (*BackupBF*) significantly reduces the extra memory references to an average of 0.05% for SPEC benchmarks and 1.6% for kernel intensive benchmarks. This is because in contrast to regular caching which causes memory access on a cache miss, with a bloom filter memory access is incurred only if there is an address that matches the bloom filter. Since bloom filter false positive rates can be made much lower than protection cache miss rates, it reduces the number of memory accesses. Finally, we show our protection cache that converts to multiple bloom filters per set on overflows (*MultiBF*). In fact, MultiBF always incurs smaller false positive rates than BackupBF. The reduced false positive rates result in a lower number of memory accesses, on average by only 0.02% for SPEC benchmarks and 0.4% for kernel intensive benchmarks. Even the worst case (iperf) shows less than 1% increase in memory accesses.

Compared our schemes to MMP, our scheme works better because of two factors. First, MMP relies on spatial locality of the protection information and stores both write-protected and non write-protected word information on chip in its PLB. In contrast, our protection cache only stores the write-protected word information. Since at any given time there is a lot less write-protected words than non write-protected words, our protection cache is more efficient. In addition, bloom filters are able to summarize protection information in the memory quite well, so they reduce the frequency of checking the protection bit vector in memory. Finally, we use counting bloom filters rather than plain bloom filters, which enable on-chip protection information to be modified (address deleted or added) without involving the main memory most of the time.

For our detection schemes, we assume 1-cycle access latency for the 3.75KB protection cache. The base case is an unprotected system. *BackupBF* shows an average performance overhead of 1% for SPEC benchmarks and 5.3% for kernel intensive benchmarks, with a worst case of 10.2% for iperf. These high overheads despite small increases in the number of memory references are due to the long latency to read, locate, modify, and write protection bit vectors in the main memory. As expected, the multiple bloom filter scheme (*MultlBF*) outperforms BackupBF scheme in all cases, and reduce the overheads to 0.8% for SPEC and 1.9% for kernel intensive benchmarks.

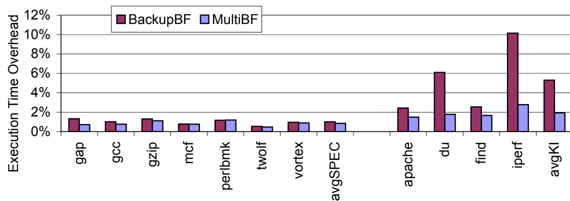


**Figure 9. Performance overhead of attack detection scheme.**

**Overall Performance Overheads of Survivability**. Figure 10 shows the total execution time overheads of the full survivability scheme, which includes bulk copying hardware support for checkpoint creation, and protection cache with multiple bloom filters for attack detection. Because the overheads of checkpoint creation and attack detection mechanisms are both low, the combined scheme still achieve low overheads. The average overheads are 1.3% for SPEC and 3.7% for kernel intensive benchmarks, with a worst case overhead of 4.7% for iperf. iperf spends 99% of its execution time in the kernel mode due to its very frequent system calls, hence we believe that very few applications would be as kernel intensive as iperf, so the 4.7% worst-case overhead probably applies to a wide range of applications as well. Additionally, the rollback handler will not incur any performance overheads unless an attack is detected, which is a rare occurrence.

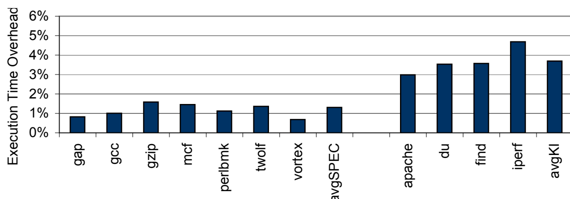


**Figure 10. Overall performance overhead.**

# 7 Conclusions and Future Work

In this paper, we presented the need for ensuring OS survivability. We presented a recovery-based survivability scheme that includes various mechanisms that work together: attack detection, recovery, and fault isolation. We have shown that through simple but carefully-designed architecture support, the performance overhead of our scheme can be kept at under 5% even for kernel intensive benchmarks. We validated the functionality of the survivability scheme through various real world attacks. Besides, our scheme achieves OS kernel survivability with a relatively simple architecture support.

As future work, we would like to improve our work by supporting more sophisticated situations such as interrupt handling and I/O operations. We will also explore applying our survivability scheme to other OSes.

# References


[1] SPEC CPU2000 Benchmarks. *http://www.spec.org/*, 2000.

[2] Linux e1000 Ethernet Card Driver Vulnerability. *http://www.securityfocus.com/bid/10352*, 2004.

[3] Linux Kernel ELF Core Dump Privilege Elevation. *http://www.securityfocus.com*, 2005.

[4] Linux Raw Device Block Device Vulnerabilities. *http://secunia.com/advisories/15392/*, 2005.

[5] Cyber Security Bulletin. *http://www.us-cert.gov/*, 2009.

[6] E. Berger et al. DieHard: Probabilistic Memory Safety for Unsafe Languages. *in PLDI*, 2006.

[7] P. Bernstein. Sequoia: a Fault-tolerant Tightly Coupled Multiprocessor for Transaction Processing. *IEEE Computer*, 1988.

[8] S. Bhatkar et al. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. *in USENIX*, 2003.

[9] B. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 1970.

[10] G. Bronevetsky et al. Application-level Checkpointing for Shared Memory Programs. *in ASPLOS*, 2004.

[11] C. Cowan. Survivability: Synergizing Security and Reliability. *Advances in Computers*, 2004.

[12] C. Cowan et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. *in USENIX*, 1998.

[13] L. Fan et al. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. *in SIGCOMM*, 1998.

[14] M. Frantzen et al. StackGhost: Hardware facilitated stack protection. *in USENIX*, 2001.

[15] X. Jiang et al. Architecture Support for Improving Bulk Memory Copying and Initialization Performance. *in PACT*, 2009.

[16] X. Jiang et al. CHOP: Adaptive Filter-based DRAM Caching for CMP Server Platforms. *in HPCA*, 2010.

[17] X. Jiang et al. CHOP: Integrating DRAM Caches for CMP Server Platforms. *in IEEE Micro Top Picks 2010*, 2010.

[18] M. Kharbutli et al. Comprehensively and Efficiently Protecting the Heap. *in ASPLOS*, 2006.

[19] A. Lenharth et al. Recovery Domains: An Organizing Principle for Recoverable Operating Systems. *in ASPLOS*, 2009.

[20] G. Liao et al. A New IP Lookup Cache for High Performance IP Routers. *in 47th DAC*, 2010.

[21] H. Lipson et al. Survivability: A New Technical and Business Perspective on Security. *New Security Paradigm Workshop*, 1999.

[22] A. Liu et al. Diverse Firewall Design. *in DSN*, 2004.

[23] F. Liu et al. Understanding How Off-chip Memory Bandwidth Partitioning in Chip-Multiprocessors Affects System Performance. *in HPCA*, 2010.

[24] P. Magnusson et al. Simics: A Full System Simulation Platform. *Computer*, 2002.

[25] M. Prvulovic et al. ReVive: Cost-effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. *in ISCA*, 2002.

[26] B. Rogers et al. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. *in ISCA*, 2009.

[27] D. Sorin et al. SafetyNet:Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. *in ISCA*, 2002.

[28] M. Swift et al. Improving the Reliability of Commodity Operating Systems. *in SOSP*, 2003.

[29] M. Swift et al. Recovering Device Drivers. *in OSDI*, 2004.

[30] E. Witchel et al. Mondrian Memory Protection. *in ASPLOS*, 2002.

[31] H. Xu et al. Detecting Exploit Code Execution in Loadable Kernel Modules. *in ACSAC*, 2004.

[32] J. Xu et al. Transparent Runtime Randomization for Security. *in Reliable Distributed Systems*, 2003.

[33] L. Zhao et al. Hardware Support for Bulk Data Movement in Server Platforms. *in ICCD*, 2005.