# OSIRIS: Efficient and Consistent Recovery of Compartmentalized Operating Systems

Koustubha Bhat* Dirk Vogt* Erik van der Kouwe‡ Ben Gras† Lionel Sambuc†
Andrew S. Tanenbaum‡ Herbert Bos‡ Cristiano Giuffrida‡
Department of Computer Science
Vrije Universiteit Amsterdam, The Netherlands
*{k.bhat, d.vogt}@vu.nl, †{ben, lionel}@minix3.org, ‡{vdkouwe, ast, herbertb, giuffrida}@cs.vu.nl

***Abstract*—Much research has gone into making operating systems more amenable to recovery and more resilient to crashes. Traditional solutions rely on partitioning the operating system (OS) to contain the effects of crashes within compartments and facilitate modular recovery. However, state dependencies among the compartments hinder recovery that is globally consistent. Such recovery typically requires expensive runtime dependency tracking which results in high performance overhead, high complexity and a large Reliable Computing Base (RCB).**

**We propose a lightweight strategy that limits recovery to cases where we can statically and conservatively prove that compartment recovery leads to a globally consistent state—trading recoverable surface for a simpler and smaller RCB with lower performance overhead and maintenance cost. We present OSIRIS, a research OS design prototype that demonstrates efficient and consistent crash recovery. Our evaluation shows that OSIRIS effectively recovers from important classes of real-world software bugs with a modest RCB and low overheads.**

## I. Introduction

All modern operating systems contain bugs [1], [2]. Studies show that all software averages between 1 and 16 bugs per 1,000 lines of code [3], [4] even in tested and deployed software, and dormant software faults persist even in mature code bases [5]. With code bases that often easily exceed a million lines of code, operating systems (OSes) contain thousands of bugs at any time. Some of these bugs are in code that is beyond the control of the operating system developers. For instance, a considerable portion of the privileged code consists of code in kernel extensions such as third-party drivers. A decade ago, Chou et al. [6], [7] identified buggy kernel extensions as an important cause of operating system crashes for both Windows and Linux. Assuming that drivers are mostly stateless and faults are mostly transient, we can handle such faults by isolating the driver code and restarting it in case of a crash [8].

However, a more recent study on the Linux kernel [9] shows that faults in stateful core OS subsystems have started to outrank the buggy drivers in importance, even though the latter are still large in number. Furthermore, these bugs are typically *not* transient [10], [11]. In other words, relying on simple re-execution is no longer a viable solution to recover from such faults. Since faults in a core OS subsystem greatly reduce overall system dependability and existing techniques cannot handle them, we conclude that software faults in the operating system still rank among the greatest challenges to system dependability. In this paper, we seek a method to mitigate their effects.

*Reliability and stateful interaction:* There are many approaches that improve fault tolerance in operating systems. Typically, they protect either applications [12] or specific OS subsystems from the effects of software faults in the operating system [13], [14], [15], [16], but there also have been efforts to provide whole-OS fault tolerance [17], [18], [19], [8].

Extending fault mitigation techniques to an entire OS is a complex problem. Most of the solutions compartmentalize the OS, primarily to prevent the effects of faults in one component from spreading onto other components. Moreover, compartmentalization allows components to be recovered individually. However, stateful runtime interactions among components make *per-component* recovery nontrivial. In a scheme that enforces strict fault isolation between OS components, recovering from crashes resembles a distributed systems recovery problem. In this context, the solutions are generally of three kinds:

1) *Global replication* – which is least suitable for a general-purpose operating system that aims for judicious and efficient usage of system resources;
2) *Dependency tracking* – which does not scale for the high-frequency inter-component interactions found in an operating system setting;
3) *Global checkpointing* – which not only hinders normal execution performance, but also degrades it exponentially when we scale to a larger number of active system components.

Global checkpointing in the context of operating systems [20] offers strong global consistency guarantees, but suffers from the need to synchronize all components, introducing bottlenecks which greatly affect overall system performance. Since OS components interact all the time, checkpoints should be taken at high frequencies—typically orders of magnitude beyond what is possible with today's global checkpointing solutions. Local, per-component checkpointing allows for more concurrency in the system but dependent components still have to coordinate to ensure that their locally checkpointed state remains consistent with that of their counterparts. In case of uncoordinated checkpoints, there is the risk of a *domino effect* [21], where crash recovery leads to unbounded rollback of inter-dependent components. In general, expensive runtime dependency tracking is the price that local checkpointing schemes pay towards guaranteeing globally consistent recovery.

*Safe recovery windows:* In this paper, we introduce a new design called OSIRIS (Operating System with Integrated

Recovery preventing Inconsistent State), which seeks to strike a new balance between performance and globally consistent recoverability of an operating system. Building on efficient in-memory checkpointing, OSIRIS recovers only in cases where we can conservatively infer that performing local recovery will not lead to global state inconsistencies. This eliminates the need for dependency tracking and synchronization, also greatly simplifying the recovery mechanism. Such solution offers better performance than any other existing scheme, at the cost of not being able to recover in every case.

Our solution is the first to achieve globally consistent recovery of stateful core OS components with very low performance overhead. With a runtime performance penalty of just 5.4% (on a microkernel-based OS baseline), OSIRIS brings OS recoverability (limited, but still powerful) within the reach of production systems. Moreover, unlike many existing OS recovery approaches, we do not limit ourselves to drivers and explicitly target the core system services. Given their heavily stateful nature, it is much harder to recover from faults in such components than from faults in drivers. For instance, a system call like `exec` involves the file system, memory manager, cache manager, process manager, etc. Crash recovery in any of these components must keep its state consistent with other components' state and resume execution in a globally consistent way.

Our approach uses knowledge about the nature of inter-component interactions in the system to perform recovery only within dependency-safe *recovery windows*—intervals during which state changes within a component have not affected other components. We use a lightweight in-memory checkpointing system [22] to allow efficient high-frequency creation of per-component checkpoints. We further optimize this approach by disabling our runtime (memory logging) instrumentation whenever recovery to a consistent state is known to be impossible. Our design results in lower runtime overhead and provides a fine-grained trade-off between recoverable surface and performance. Finally, instead of replaying the execution after recovering from a crash, we send an error to the component that sent the request that triggered the crash. This allows us to deal with persistent faults in addition to transient faults.

*Contributions:* First, we describe an operating system recovery method that determines whether rolling back only component-local state can restore the OS to a globally consistent state.

Second, we show how OSIRIS occupies a meaningful new point in the design space of dependable operating systems by introducing a new trade-off that solves the performance, maintenance, and complexity drawbacks of existing solutions at the cost of reduced recovery surface. With a performance overhead of just 5.4%, we show that we can achieve an average recovery surface of 68.4% in critical OS components with the total size of Reliable Computing Base (RCB) constituting only 12.5% of the code base.

Third, we explain how our approach deals with persistent software faults in core system services whereas most prior efforts are limited to transient faults.

Fourth, we minimize checkpointing instrumentation overhead by disabling memory logging when we cannot recover, providing a fine-grained trade-off between performance and recovery surface.

Fifth, we show that our prototype implementation performs meaningful recovery from real-world faults by means of large-scale fault injection experiments. We compare our results with a baseline recovery strategy and demonstrate a significant improvement. We also show that the performance and memory overheads are within practical limits.

*Roadmap:* The remainder of this paper is laid out as follows. We provide a background on recovery techniques and the challenges of stateful recovery in Section II. We provide an overview of OSIRIS in Section III and detail its design elements in Section IV. We provide implementation details in Section V. We evaluate our prototype in Section VI and discuss the limitations and future work in Section VII. We survey related work in Section VIII and conclude in Section IX.

## II. Background and Problem Statement

Software recoverability has been studied in various contexts. They range from preventing loss of unsaved application context due to operating system crashes, insulating operating systems from faulty drivers and hardware peripherals, and improving availability of server applications to dealing with fatal faults in high-performance computing and large-scale distributed systems. Recovery solutions typically consist of the following elements: fault isolation, a way to statefully restart components, and dependency monitoring. We will discuss each in turn. Afterwards, we pose our problem statement and the fault model that we assume in this work.

### A. Fault isolation

Modular software design, along with enforcing component boundaries allows a system to be compartmentalized, reducing the scope of recovery to just the affected compartments. Many fault-isolation techniques exist. Software-only techniques use static analysis [15], [23], or dynamic object tracking [24], while hardware-assisted methods may build on hypervisor-supported service domains [17] and virtual-memory-based isolation [18], [25]. In addition, there are solutions that superimpose higher-level semantics over static program analysis to isolate fault regions [19], [26], [27]. Regardless of the techniques applied, compartmentalization of software into smaller fault domains represents a first step towards achieving practical software recoverability.

### B. Stateful restart

A crashed component needs revival to resume normal system execution. In the simplest of cases, components are mostly stateless and can be revived by simply restarting them—much like the device drivers in MINIX 3 [25]. However, in case of stateful components and inter-component interactions, revival must ensure that the affected components are restored to a sane state. Kadav et al. [15] repurpose power management code in peripheral devices to take device checkpoints in addition to saving device driver state. Similarly, researchers have proposed runtime object tracking [14] and request-oriented undo logging [22] to protect drivers and even the Linux kernel [19]. In a distributed system setting, Cruz [28] explored a coordinated checkpoint-restart scheme using process and network state migration techniques.

### C. Dependency tracking

While fault isolation and stateful restart are mostly solved problems by themselves, monitoring state dependencies is not. We first describe why it is necessary to monitor state dependencies and then explain why current solutions are not adequate for use in production environments.

If a request sent from component A to component B triggers state changes in component B, it creates a *dependency* between the states of the two components A and B. If one is rolled back to the last checkpoint while the other retains its state, this may lead to two kinds of inconsistencies: 1) If component A crashes before component B completes, state changes in component B may get *orphaned* because A is no longer interested in the results. Moreover, any inter-component interactions performed by B to fulfill the request also get orphaned recursively. 2) If component B crashes while handling the request, component A may get no reply or an incorrect reply and component A would not know in which state component B ends up.

We must deal with both the cases in order to avoid inconsistent crash recovery. Existing solutions track state dependencies to solve these problems, but doing so complicates recovery and results in increased software complexity and performance overhead. For example, the hierarchical recovery domains in Akeso [19] incur slowdowns between 1.08x to 5.6x even in a virtualized environment. The device driver tracking in Nooks [24], which allows the system to recover from crashes only in kernel extensions, incurs up to 60% runtime overhead depending on the workload. Shadow drivers [14], which build on top of Nooks, report an increase in CPU utilization by up to 30% for a benchmark that performs a significant number of driver-kernel interactions. In addition, dependency tracking complicates the recovery mechanism, possibly even to the extent of making it one of the more complex elements in the system. In contrast, the approach taken by ASSURE [26] and REASSURE [27], which stops all the execution threads in a checkpointing interval, is much simpler, but may not be applicable in a broader context and is prone to deadlocks. We conclude that runtime dependency tracking adds considerable performance overhead and complexity to a system, making operating system recovery impractical in production systems.

### D. Problem statement and goals

As discussed, dependency tracking and recursive recovery greatly increase complexity and cause substantial performance overhead, making them unsuitable for production systems. In this paper, we aim to show that, by adjusting expectations, it is possible to achieve a practical recovery solution with much less complexity and much better performance.

Our goals for consistent recovery in OSIRIS are twofold: (1) ensure global state consistent recovery, (2) keep the crash recovery infrastructure simple to minimize the risk of introducing faults in the RCB. In the remainder of this paper, we present our recovery solution, which takes into account the possibility of system state being spread across several fault domains and yet avoids performing complex and expensive runtime dependency tracking.

### E. Fault model

In this section, we describe our fault model and discuss its consequences.

Many previous efforts assume that faults are transient [25], [29], [30], which means that simply restarting a component and restoring its original state is sufficient. Given the transient nature of the fault, attempting the same operation again is likely to succeed. While it is true that hardware faults are often transient (e.g., bit flips in memory) and software faults may also be transient (e.g., a race condition that only occurs for particular scheduling decisions), many common software faults are persistent, where the fault can be a function of the inputs and the existing state. In the case of persistent faults, recovering a component and replaying the same inputs will inevitably trigger the same fault again. Our fault model considers both transient and persistent software faults.

Fail-stop faults cause the affected component to crash immediately, unlike fail-silent errors which corrupt its state without an immediate crash. Fail-stop faults are common. Typical examples of fail-stop errors are NULL-pointer dereferences and divisions by zero. Moreover, defensive coding practices such as the use of assertions to verify invariants transform many potential fail-silent faults into fail-stop faults. In addition, compilers are increasingly equipped with options that convert traditionally fail-silent faults into fail-stop faults. Typical examples are bounds checking and the wide range of sanitization and stack protector options in popular compilers like `gcc` and `clang`. Finally, hung-component faults can be detected, and thus transformed into fail-stop faults, by sending regular heartbeat messages and killing them if they do not respond in time [25]. Our fault model assumes only fail-stop faults and relies on fault detection mechanisms (such as the ones above) to handle many classes of fail-silent faults in a "fail-stop fashion".

Furthermore, our fault model assumes only one failure (i.e., crash) at a time. We note this is a relatively minor restriction considering the other assumptions we make. First, since we assume fail-stop faults, the failing component itself is unable to execute any more code between the failure and its recovery, so a second failure cannot happen there. Second, since our recovery code is trusted (in the small RCB), no failure can happen in the recovery code itself. The only possibility of additional failures during recovery is thus elsewhere in the system. To minimize this possibility, we temporarily disallow system call processing, stalling the userland until recovery operations complete.

Finally, an important assumption is that faults may also occur in core OS components. This is unlike much prior work that only deals with faults that occur in drivers [24], [16], [25], [14]. Failures in core OS components are considerably harder to deal with than driver failures, because the system requires availability of the core OS components at all times. Moreover, while drivers typically have relatively simple and highly standardized communication protocols [24], the core system components are much more tightly coupled and need to perform complex interactions to execute a range of cross-cutting system calls. Akeso [19] and CuriOS [18] also recover from faults in the OS, but their strategies impose nontrivial runtime performance and design constraints.

## III. Overview

In this section, we first give a high-level overview of OSIRIS' system architecture and then provide an example explaining our recovery methodology.

### A. System architecture

OSIRIS builds on top of a compartmentalized operating system. Its fault-isolated components interact through message passing. We put in place *Side Effect Engraved Passages (SEEPs)*, wrapping each of the channels transporting these messages. A SEEP keeps side-effects information, especially about whether a state dependency arises out of that interaction or not. SEEPs allow OSIRIS to restrict component recovery to cases where state changes since the last checkpoint are not visible outside the crashed component. In the other cases, the system aborts and does not attempt recovery.

We perform recovery by rolling back the crashed component state to its last *checkpoint*. The components making up OSIRIS are *event-driven*, allowing them to create a checkpoint every time they receive a new request message (i.e., event). Because checkpoints need to be created at high frequencies, we do not copy the entire state but rather opt for *incremental checkpoints* by maintaining an *undo log* [22]. The undo log stores all the memory writes since the last checkpoint and allows them to be rolled back at recovery time. We use a compiler pass to insert necessary instrumentation to maintain the log. Since our fault model focuses on fail-stop faults, we can assume the last checkpoint to be a valid component state. This means that only one checkpoint needs to be maintained at a time. After rolling back to the last checkpoint, we send an error reply to the component that sent the failure-triggering request, instead of simply replaying its execution. This is to also recover from persistent faults, as dictated by our fault model.

### B. Ensuring safe recovery

To ensure that we attempt recovery only when the system is known to reach a consistent state, we first consider what it means for our system to be in an inconsistent state. The system is in an inconsistent state whenever at least one component believes another component to be in a state it is not in. Our system is compartmentalized into components that can only interact through SEEPs. This means state changes after the most recent outgoing SEEP message are not known to the rest of the system. Therefore there can be no inconsistencies if we roll back a component to a checkpoint that was made after the last outgoing SEEP message in that component. While this approach is more conservative than strictly necessary, it prevents the need to do dependency checking and recursive rollbacks. When a component crashes, we only need to determine whether the last checkpoint was made after the last SEEP.

A SEEP can be *state-modifying* or *non-state-modifying*, indicating whether a request passing through it affects the state at the receiving side or not, respectively. Because the receiving end of a non-state-modifying SEEP does not update its own state, it does not become aware of changes in the sender's state. As such, we can safely ignore these for recovery purposes and only consider the last encountered state-modifying SEEP. Building on these intuitions, OSIRIS only performs recovery operations when the crashed component has sent no state-modifying outgoing message since the last checkpoint, masking the failure to any other component in the system and resulting in a globally consistent state after recovery by construction.

### C. Recovery example

Considering our OSIRIS design, let us exemplify what happens when a fault occurs. Consider, for example, a shell issues a `fork()` system call to create a new process to run a user command. The shell process sends a message to the responsible OS component and waits for a reply. When the component responsible for creating new processes receives the message, OSIRIS creates a checkpoint which marks the start of a new *recovery window* for the component. Within the recovery window, our instrumentation maintains the checkpoint-associated undo log for recovery purposes. The recovery window remains open as long as we can conservatively prove that the component has not altered states of any other components.

Now let us consider what happens in case of a crash. For instance, assume that, while handling the `fork()` request and before communicating with other components, the code dereferences a NULL pointer that causes the *Process Manager* (PM) to crash. In response to the crash, the OS kernel notifies the *Recovery Server*, a key OS component in OSIRIS. The Recovery Server has a spare fresh copy of all recoverable components. It transfers state from the crashed PM to the fresh copy and then rolls back the operations listed in the undo log to restore the state that existed when the `fork()` call was received. It then replaces the crashed component with the recovered PM component. The Recovery Server also sends an `E_CRASH` error code to the requester—in this case, the shell. The shell can handle it just like other unexpected failures, such as resource limits preventing `fork()`. Most well-written programs routinely deal with such error codes and take the most appropriate action. In this case, the shell would simply abort the execution of the command and inform the user that something went wrong. At this point, the failure has been cleanly handled and the system is once again in a stable and consistent state.

So far, our example illustrates an instance of successful recovery. However, it is not always possible to perform a successful recovery. Our system closes the recovery window whenever the component sends a message through a state-modifying SEEP. If a crash occurs after the recovery window has closed, the system knows that recovery may not yield a consistent state and performs a controlled shutdown to prevent a potentially inconsistent state (with unpredictable consequences). Hence, as an optimization, after the recovery window has closed, the system stops updating the undo log, reducing the performance overhead. As a result, our approach achieves good performance and never performs unsafe recovery by design under the fail-stop fault model.

## IV. Design

In this section we discuss the design of OSIRIS. We first describe the underlying programming model. Next, we explain how our system decides whether it can safely perform a component-local recovery and how it performs recovery operations. Then, we describe our incremental checkpointing optimization that reduces runtime overhead whenever safe

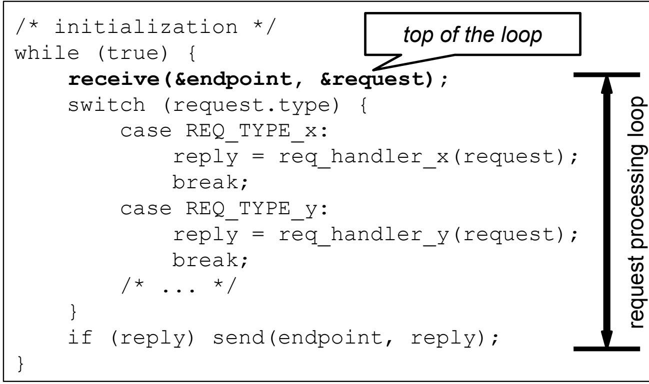


Fig. 1. OSIRIS' event-driven programming model.

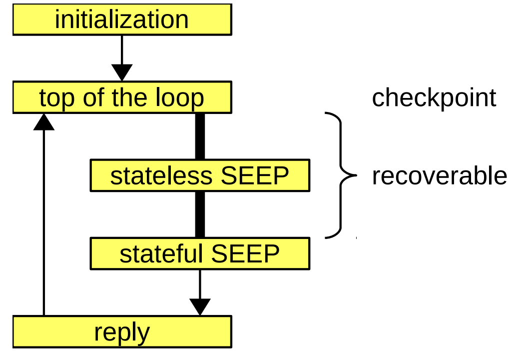


Fig. 2. Example showing the recovery window; undo log instrumentation is enabled where the line is thick.

recovery is known to be impossible. Finally, we explain how our approach deals with multithreading.

### A. *Event-driven programming model*

Our OSIRIS design applies to compartmentalized operating systems which apply component isolation and use a message-passing interface for communication. In OSIRIS, core OS components (or servers) follow an *event-driven* programming model as shown in Figure 1. An event-driven model significantly simplifies state management for recovery purposes [31]. After initialization, the OS components run indefinitely in a request processing loop. They block to receive incoming messages at the top-of-the-loop, including events such as system calls initiated by user programs as well as requests from other OS components and responses to previously issued asynchronous requests. The corresponding request handler then processes the received message based on its type. Finally, a component typically sends a reply except in cases where the sender must block (for example when no data is available to satisfy a `read()` call) or if the incoming message itself was an asynchronous reply. In those cases, the server postpones the reply until it receives the response it needs to complete its work. This model imposes clear boundaries on state changes and binds them to specific request messages, which, as mentioned earlier, simplifies state management and therefore recovery.

### B. *When to perform recovery*

Inter-component communication may lead to state dependencies between the communicating servers. Such dependencies threaten global consistency after crash recovery. Our recovery methodology is centered around identifying a recovery window within the request processing loop in each OS component. A recovery window starts at the top-of-the-loop and spans those instructions that we may roll back to the top-of-the-loop (checkpoint) without affecting the consistency of the overall system. In other words, the recovery window *closes* at the point where an irrecoverable operation is encountered. Conservatively speaking, any operation that affects the global state of the system is a potential threat to consistent recovery.

In OSIRIS, we make communication interfaces among the system components *side effect* aware. As mentioned in Section III, all messages are exchanged through Side Effect Engraved Passages (*SEEPs*). SEEP is aware of the consequences of the messages it carries. This allows us to classify every message in the system based on whether or not it affects the state of the recipient and whether or not it is possible to send an error reply back after crash recovery. We use this information to define system-wide *recovery policies*. Recovery policies control which classes of SEEPs are allowed within a recovery window, hence controlling the span of recovery windows in each OS component. The first SEEP communication that the policy does not allow closes the active recovery window in an OS component. We define two simple recovery policies to demonstrate our design: (i) *pessimistic recovery*, where sending out *any* message closes the recovery window, and (ii) *enhanced recovery* (default), where we use SEEP to identify which interactions actually create dependencies.

To implement SEEP (and enhanced recovery), we rely on a compiler pass to instrument all the outbound communication call sites with metadata indicating their potential side effects.

Figure 2 shows an example of how a recovery window would appear when messages are sent while processing a request. We create a checkpoint at the `receive()` call at the top of the request processing loop, marking the start of the recovery window. The SEEP the first message is sent through is non-state-modifying, so it does not affect the recovery window in our default configuration. The second SEEP is state-modifying and causes the recovery window to be closed. The system can discard the checkpoint (i.e., undo log data) at that point as the system will not attempt to restore it.

### C. *How to perform recovery*

At its core, OSIRIS' recovery mechanism relies on memory checkpointing [22], [32]. It creates a new in-memory checkpoint at the start of every recovery window. From that point onward, compiler instrumentation keeps track of an *undo log*. Specifically, an LLVM [33] instrumentation pass places hooks in the system components to keep track of every memory write operation. It instruments every `store` instruction of the LLVM intermediate representation with a call to the checkpoint-recovery library which adds an entry to the undo log. Each entry consists of the address and the original value that was overwritten. At any point within the recovery window it is possible to restore the checkpoint by rolling back the entries in the undo log. We base OSIRIS' approach on our prior work on lightweight memory checkpointing [22], but with an

important optimization (see Section IV-D). Furthermore, we use software fault isolation (SFI) to protect the checkpointing library from any inadvertent corruption.

The benefit of using write logging rather than storing a copy of the original state is that it is efficient for creating checkpoints at a high frequency. This makes it a good fit for our system, as operating system components typically deal with many incoming messages while doing a relatively small amount of work for each message. The latter property bounds the number of per-message memory writes to a very limited number in practice, favoring a simple undo log organization over more sophisticated memory shadowing schemes [22].

One OS component, known as the *Recovery Server* (RS), is responsible for detecting and reacting to crashes in the system. RS receives a notification whenever a server crashes and periodically sends heartbeat messages to detect hung servers. In addition, it initiates the recovery procedure whenever it detects a crash or hang. Recovery in our design is structured in three phases: *restart*, *rollback*, and *reconciliation*.

In the *restart phase*, RS transparently replaces a deceased component with a freshly forked *clone* component. However, if the target component is one of the core system servers such as PM, VM, or even RS itself, `fork()` and other fundamental system functions would not work properly. Hence, for core system servers, RS replaces the deceased component with a clone prepared ahead of time. RS starts the clone and, in its initialization code, the clone determines that it is in recovery mode. After receiving a special *capability* from RS, kernel support allows the clone to copy over the data sections from the crashed component's memory image. The clone has now the same state as the original server when it crashed.

In the *rollback phase*, the initialization code in the clone rolls back the local state using the undo log just transferred from the crashed component. This restores the last checkpoint taken at the start of the current recovery window, i.e., the top-of-the-loop. At this point, the state is identical to the state at the time when the crash-triggering request was received. Under a fail-stop fault model, this is also the last known good state.

Finally, we enter the *reconciliation phase*. Now that the local component state has been restored, we still need to ensure that the global state is consistent. The *recovery action* to take at this point is specified by the recovery model and depends on the state of the recovery window at the time of the crash. Although this model is extensible, we use two strategies in our OSIRIS prototype. If the recovery window was open at the time of the crash and the last received message was a request to which we can reply, we perform *error virtualization* by sending an error reply to the requesting process. We then discard the original message. Doing so makes the state globally consistent and allows OSIRIS to seamlessly handle persistent faults. In the other cases, we know that we cannot reach a consistent state and we then perform a controlled shutdown of the system. Although this means giving up on a possibly successful recovery, it allows us to guarantee a consistent state, eliminating the unpredictable consequences of an unstable system.

### D. Optimization

To support our checkpointing algorithm we need to update the undo log for each memory write, which introduces runtime overhead. However, as it is impossible to recover when execution goes past the recovery window, the checkpoint is not useful anymore. We reduce overhead by updating the undo log only when the recovery window is open. For this purpose, we rely on another LLVM-based compiler pass to create two clones of every function in the component and replace the original function with one that conditionally selects one of the two clones based on whether the recovery window is open or not. To ensure that recovery window status is checked every time it enters the top-of-the-loop, we perform loop extraction and wrap it in a new function prior to function cloning. Then, our checkpointing instrumentation pass adds write logging only to the cloned version that is used when the recovery window is open. Finally, we force LLVM to inline the cloned functions to avoid introducing function call overhead.

This optimization reduces the runtime performance overhead by 11%. Furthermore, it provides an explicit trade-off between recovery surface and performance overhead. Decreasing the recovery surface (e.g., by switching to *pessimistic recovery*) would lead to less frequent execution of instrumented function clones and thus better performance (and vice versa).

### E. Multithreading

It is possible to have multiple message loops run simultaneously in the same component using multithreading, which can possibly increase efficiency and/or lower complexity in certain components. Our design supports multithreading under the condition that the state is managed by the server itself by using a cooperative thread library. The recovery window is open when a thread becomes active (i.e., starts processing a message) and forcefully closed when the same thread becomes inactive (i.e., explicitly yields to other threads). Restoring the state from a crashed server also restores the state of the inactive threads. The active thread, on the other hand, needs special handling. When we take a checkpoint, we call the thread library that forces the context to be saved. When we restore a checkpoint, the thread library thinks the crashed thread is still running while the server is in fact starting in the main thread. For this reason, we call a function to fix the current thread variable and add the crashed thread back to the run queue. After these steps, the thread library is back to a consistent state and the server can run again.

## V. Implementation

We implemented our OSIRIS prototype on top of the `MINIX 3` microkernel-based operating system architecture [8]. The core operating system consists of a number of OS user-space processes (referred to as *system servers*) isolated from each other using the MMU and a small microkernel that performs privileged low-level operations (e.g., scheduling and message passing). The underlying design adheres to the *principle of least authority* (POLA), which minimizes the consequences of faults through a combination of memory isolation and by restricting each component (i.e., process) to only those operations that are necessary for it to do its job. This compartmentalized design serves as a basis for fault isolation, which is a key prerequisite for OSIRIS' crash recovery strategy (with no uncontrolled fault propagation across OS components).

The system servers at the core of our OSIRIS prototype are: the *Process Manager* (PM), which manages processes and

signals, the *Virtual Memory Manager* (VM), which manages virtual memory, the *Virtual Filesystem Server* (VFS), which provides a virtual file system interface, the *Data Store* (DS), which provides a persistent key-value store service, and the *Recovery Server* (RS), which detects and restores crashed OS components in the system. The VFS server in our prototype is multithreaded to prevent slow disk operations from effectively blocking the system. Our prototype allows all these core system components (including RS itself) to be recovered in case of crashes using our design.

In addition, OSIRIS includes a set of LLVM link-time instrumentation passes and static libraries together adding up to 6,496 LOC[1]. The Recovery Server implements the restart phase described in Section IV-C. The static libraries, in turn, implement our checkpointing, rollback, and reconciliation mechanisms. Compile-time settings allow SEEPs to be mapped to their corresponding reconciliation mechanisms, which, as a result, define the recovery policies supported by our system. To enable OSIRIS' recovery functionalities, we link every system component against our static libraries and instrument (and optimize) each component using our LLVM link-time passes.

### A. Reliable Computing Base

The *Reliable Computing Base* (RCB) [34] consists of the parts of the system that we need to trust to be free of faults. The RCB in OSIRIS includes mechanisms that implement:

1) Checkpointing – Maintaining a simple per-request undo log.
2) Restartability – Maintaining clones of OS components, transferring state, and replacing crashed components.
3) Recovery window management – Tracking whether the per-component recovery window is open or not.
4) Initialization – Calling a component-specific function to initialize the local state before entering the request processing loop.
5) Message passing substrate - The underlying microkernel in our prototype.

OSIRIS has a total of 237,270 LOC. The RCB adds up to 29,732 LOC which is only 12.5% of the entire code base.

## VI. Evaluation

We evaluate our system in terms of recovery coverage (Section VI-A), survivability (Section VI-B), performance (Section VI-C), and service disruption guarantees (Section VI-E).

For our experiments, we use two different workloads. For our performance evaluation, we rely on Unixbench [35], which is specifically designed and widely used to measure OS performance. As a workload for recovery and survivability tests, we use a homegrown set of 89 programs in total, written to maximize code coverage in the system servers. In this section, we refer to this set of programs as *prototype test suite* (included in `MINIX 3` [36]).

We use four recovery policies to evaluate OSIRIS. In addition to the pessimistic and enhanced recovery policies described in Section VI, we define two more policies as a baseline for comparison purposes:

1) *Stateless restart*. This serves as a baseline to compare against existing "microreboot systems" operating stateless recovery.
2) *Naive recovery*. This serves as a baseline to compare against best-effort recovery strategies with no special handling.

[1]Source lines of code generated using David A. Wheeler's 'SLOCCount'

| **Server** | **Recovery coverage (%)** | |
|---|---|---|
| | *Pessimistic* | *Enhanced* |
| PM | 54.9 | 61.7 |
| VFS | 72.3 | 72.3 |
| VM | 64.6 | 64.6 |
| DS | 47.1 | 92.8 |
| RS | 49.4 | 50.5 |
| *Weighted average* | 57.7 | 68.4 |

TABLE I. Percentage of time spent inside the recovery window for each server (mean weighted by time spent running server)

### A. Recovery coverage

To measure the opportunity for recovery under our chosen recovery models, we measure the cumulative execution time each server spends inside and outside the recovery window while executing the prototype test suite. We count the number of basic blocks covered during the execution in each of the five servers and compute the recovery coverage as the fraction of number of basic blocks executed inside recovery windows out of the total number of basic blocks executed in the servers. This provides an indication of how often the system remains recoverable. Table I presents the results for our pessimistic and enhanced recovery policies. As shown in the table, the execution spends a mean of 57.7% and 68.4% of the execution time across all the servers inside recovery windows, respectively.

As shown in the table, DS has the lowest recovery coverage in pessimistic mode and the highest in enhanced mode. This indicates the presence of a SEEP fairly early in DS' request processing loop—which is non-state-modifying as marked in enhanced mode. DS is a relatively simple server, which rarely issues state-modifying calls to the rest of the system. Hence, it is almost always recoverable. Since enhanced mode allows SEEPs that perform read-only interactions with other components to keep recovery windows open, the increase in recovery coverage for PM can be explained by the many read-mostly system calls it implements. This property applies to many other OS components (indeed OSes are known to exhibit read-mostly behavior in typical workloads) and overall our system can be recovered 68.4% of the time. This means OSIRIS can guarantee safe recovery in the majority of the cases.

### B. Survivability

To demonstrate improved survivability of the system in the presence of faults, we run large-scale fault injection experiments. We conduct fault injection experiments by booting our prototype inside a virtual machine and executing our prototype test suite. We use a modified QEMU which allows us to log the status of the system and outcomes of the tests in a way that is

| **Recovery mode** | *Pass* | *Fail* | *Shutdown* | *Crash* |
|---|---|---|---|---|
| Stateless | 19.6% | 0.0% | 0.0% | 80.4% |
| Naive | 20.6% | 2.4% | 0.0% | 77.0% |
| Pessimistic | 18.5% | 0.0% | 81.3% | 0.2% |
| Enhanced | 25.6% | 6.5% | 66.1% | 1.9% |

TABLE II. SURVIVABILITY UNDER RANDOM FAULT INJECTION OF FAIL-STOP FAILURE-MODE FAULTS.

| **Recovery mode** | *Pass* | *Fail* | *Shutdown* | *Crash* |
|---|---|---|---|---|
| Stateless | 47.8% | 10.5% | 0.0% | 41.7% |
| Naive | 48.5% | 11.9% | 0.0% | 39.6% |
| Pessimistic | 47.3% | 10.5% | 38.2% | 4.0% |
| Enhanced | 50.4% | 12.0% | 32.9% | 4.8% |

TABLE III. SURVIVABILITY UNDER RANDOM FAULT INJECTION OF FULL EDFI FAULTS.

| **Benchmark** | *Linux* | | *OSIRIS* | | Slowdown (x) |
|---|---|---|---|---|---|
| dhry2reg | 1,707.8 | (4.2) | 357.7 | (1.1) | 4.77 |
| whetstone-double | 464.1 | (0.9) | 200.4 | (0.1) | 2.32 |
| execl | 1,006.4 | (3.8) | 1,171.0 | (3.9) | 0.86 |
| fstime | 2,975.8 | (3.9) | 1,106.0 | (1.9) | 2.69 |
| fsbuffer | 320.7 | (0.5) | 1,299.0 | (229.1) | 0.25 |
| fsdisk | 1,398.9 | (30.4) | 106.8 | (0.4) | 13.09 |
| pipe | 1,143.3 | (39.8) | 65.2 | (0.1) | 17.54 |
| context1 | 1,590.2 | (7.8) | 260.3 | (0.5) | 6.11 |
| spawn | 1,204.5 | (3.4) | 36.5 | (0.3) | 33.00 |
| syscall | 122.5 | (0.2) | 46.3 | (1.8) | 2.65 |
| shell1 | 430.1 | (4.2) | 385.2 | (102.0) | 1.12 |
| shell8 | 1,605.3 | (10.3) | 45.9 | (0.1) | 35.01 |
| *geomean* | 873.5 | | 207.9 | | 4.20 |

TABLE IV. BASELINE PERFORMANCE COMPARED TO LINUX (MEDIAN UNIXBENCH SCORES, HIGHER IS BETTER, STD.DEV. IN PARENTHESES).

not affected by the injected faults. We use EDFI [37] to inject the faults. We perform a separate profiling run to determine which fault candidates actually get triggered by our prototype test suite to exclude those that are triggered during boot time or are not triggered at all. Boot-time errors are not a good measure for survivability and they are unrealistic because such faults would be removed in the testing phase, while untriggered faults would inflate the statistics with runs in which no recovery is needed. The end result of each run is a log that we use to classify the run based on whether the system crashed, whether the tests succeeded, and what recovery decisions were taken.

We performed the experiments in eight different settings: all combinations of two different fault models and four different recovery models. The first fault model consists only of fail-stop errors (dereferencing a NULL pointer). It allows us to determine how effective our recovery mechanism is in the fail-stop situation for which our system is designed. The second fault model uses the full set of realistic software faults available in EDFI, which shows to what extent the current implementation of our approach also generalizes to other types of common faults. To ensure comparability between recovery strategies, we select faults to inject once for both fault models and apply the same faults to each of the recovery models.

Tables II and III show the performance of our recovery system under fault injection for the fail-stop and full EDFI fault models respectively. We injected a total of 757 fail-stop faults and 992 full EDFI faults, each in a separate run. This covers all the appropriate fault injection locations based on our criterion that boot time and unreached faults are to be excluded. We classify the outcomes of the runs in one of four groups: “pass” means that the test suite has completed and all tests passed, “fail” means that the test suite has completed but one or more tests failed, “shutdown” means a non-recoverable fault was detected and a controlled shutdown was performed, “crash” means the system suffered from an uncontrolled crash or hang. Since our aim is to measure survivability, the goal is to keep the system running even if there is some degradation of service (visible as failed tests). Hence, we prefer to have as many completed (passed or failed) runs as possible. As for the remainder, a controlled shutdown is much preferred over a crash, which may indicate random behavior and corruption.

With fail-stop errors (the fault model for which our solution was designed), the enhanced recovery mode manages to provide significantly better survivability than all the other approaches, especially when considering the “fail” case where the test fails but the system remains stable. The pessimistic approach has somewhat lower survivability than the other approaches, which is to be expected as it sometimes shuts down in cases where recovery may work out even though it cannot be proven to be safe. Both the pessimistic and enhanced approaches are very effective in reducing the number of crashes. We must note that crashes cannot be fully eliminated as it is impossible to recover from faults injected into the code involved in the recovery itself (such faults violate the single fault assumption in our fault model). We conclude that for faults within our fault model, our enhanced recovery method offers superior survivability while still avoiding recovery in cases where it cannot be proven to be safe.

Even when injecting all faults from the EDFI model, violating our fail-stop assumption, our enhanced recovery method offers the best result in terms of both survivability and is very effective at avoiding crashes. The higher number of crashes in this case is to be expected as we can no longer assume our checkpoint to be in a known-good state due to the possibility of silent state corruption. The fact that our approach still performs well shows its robustness in the face of violations of the fault model and its ability to handle realistic software faults.

### C. Performance overhead

To evaluate the performance of our system we used the Unixbench benchmark [35]. We ran the benchmark 11 times on a 4-core 2.3 GHz AMD Phenom processor with 6 GB of RAM. Table IV compares the median Unixbench score of the baseline system (without recovery) against Linux. Our prototype is significantly slower than Linux in the Unixbench benchmark. This can be explained by the overhead incurred by context-switching between OS components due to the microkernel design and the fact that Linux is a much more mature and optimized system with performance as one of its major goals. Our focus however remains on recoverability in compartmentalized systems rather than on microkernel system performance, a subject extensively explored in prior work [38], [39], [40].

To evaluate the overhead incurred by our recovery solution, we compare the baseline against our unoptimized recovery instrumentation, optimized for the pessimistic recovery policy, and optimized for the enhanced recovery policy. The relative slowdowns are listed in Table V. The results show that our optimization of disabling undo log updates outside the recovery window greatly pays off. Overall, in comparison

| Benchmark | *Without opt.* | | *Pessimistic* | | *Enhanced* | |
|---|---|---|---|---|---|---|
| dhry2reg | 1.001 | (0.003) | 0.996 | (0.004) | 0.991 | (0.004) |
| whetstone-double | 1.002 | (0.001) | 1.001 | (0.001) | 1.003 | (0.001) |
| execl | 1.326 | (0.007) | 0.750 | (0.003) | 0.762 | (0.004) |
| fstime | 1.321 | (0.003) | 0.749 | (0.002) | 0.762 | (0.002) |
| fsbuffer | 2.317 | (0.879) | 1.175 | (0.207) | 1.194 | (0.211) |
| fsdisk | 1.165 | (0.008) | 1.168 | (0.006) | 1.179 | (0.007) |
| pipe | 1.158 | (0.007) | 1.158 | (0.004) | 1.169 | (0.006) |
| context1 | 1.137 | (0.007) | 1.146 | (0.003) | 1.156 | (0.003) |
| spawn | 1.228 | (0.010) | 1.213 | (0.009) | 1.253 | (0.010) |
| syscall | 1.173 | (0.047) | 1.164 | (0.050) | 1.164 | (0.046) |
| shell1 | 1.110 | (0.368) | 0.942 | (0.248) | 0.928 | (0.245) |
| shell8 | 1.256 | (0.004) | 1.261 | (0.004) | 1.266 | (0.005) |
| *geomean* | 1.235 | | 1.046 | | 1.054 | |

TABLE V. SLOWDOWN RATIO (MEDIAN SLOWDOWN RATIO, LOWER IS BETTER, STD.DEV. IN PARENTHESES).

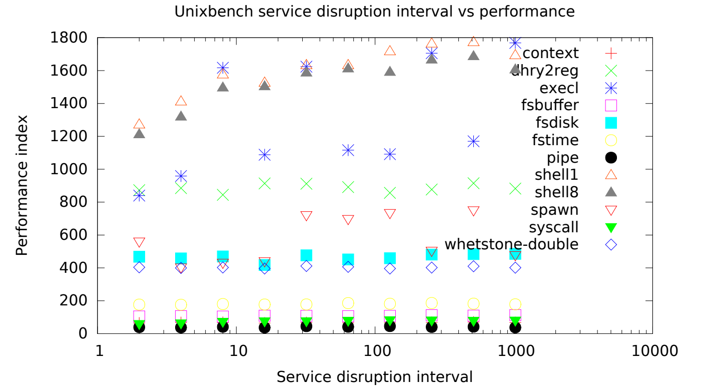


Fig. 3. Unixbench scores as a function of service disruption interval.

to that without optimization, the performance overhead has decreased from about 23% to only 5%. As a consequence, the system incurs only a modest performance overhead of around 5% for both recovery modes. The pessimistic mode incurs lower overhead than the enhanced mode since recovery windows remain open for shorter periods of time. We observe a performance improvement in our `execl` and `fstime` tests due to their multi-process interactions. In such tests, our instrumentation seems to positively affect scheduling decisions and improve overall performance [41].

### D. Memory overhead

OSIRIS increases memory usage due to two main factors: maintaining the undo log and keeping spare copies (clones) of servers in memory for recovery purposes. We present the resulting memory overhead in Table VI. The table shows average runtime memory usage per component observed across the baseline and instrumented variants of our prototype. We measure the physical memory usage in a quiescent state for each component. We then add the maximum undo log size reported during the per-component execution of Unixbench. We observe a total memory overhead of 50 MB, which represents a 6-fold memory usage increase for those OS servers. This is, however, mostly the overhead needed for VM alone. Maintaining a VM clone for recoverability purposes requires a lot of memory pre-allocation so that the new VM does not depend on the defunct VM for memory allocation during recovery. We note that even with this memory overhead, the system servers consume only a small part of memory in comparison to that typically used by user applications.

### E. Service disruption

We show that OSIRIS guarantees continuity of execution with acceptable service disruption in the face of even high-intensity and consistent inflow of crashes. In contrast to the randomized fault injection campaign presented in Subsection VI-B, we designed this experiment to specifically show performance and continuity of a long-running benchmark (Unixbench) under a challenging (but synthetic) fault load. To ensure consistent recoverability and run the benchmark to completion, we only injected faults within the recovery window. In particular, we injected fail-stop faults in the PM server at regular time intervals. We selected PM, as it is a heavily exercised operating system component during the benchmark. We repeated this experiment several times, each time doubling the influx of faults per interval. For each run, we verified that the benchmark completed without any functional service degradation and measured the resulting benchmark performance. Figure 3 presents our results.

As shown in the figure, some of the performance tests are not at all affected by the injected faults. However, others undergo significant slowdown, most noticeably `shell1`, `shell8`, and `execl`. This is explained by a heavy dependence on PM for these tests. Correspondingly, the slowdown is absent for tests with no dependence on PM, such as `dhry2reg`, `whetstone-double`, `fsdisk`, and `fsbuffer`. We conclude that the performance degradation induced by periodic crash recovery operations is workload-dependent and, despite the performance impact in some cases, OSIRIS can effectively guarantee survivability and continuity of execution even in extreme high-frequency failure scenarios.

## VII. LIMITATIONS AND FUTURE WORK

*Controlled shutdown:* Fault injection results show that our approach substantially reduces the number of crashes. We perform a controlled shutdown whenever we cannot guarantee that a recovery attempt leads to a consistent global state. In this case, state is still lost. However, the system is still consistent, which is often more important. In future work, we could extend our approach to provide opportunities for user applications to save their state before restarting the system—a strategy similar to what Otherworld [12] supports after a system crash using annotations.

*Extensibility:* Although our prototype implements a small set of classes of SEEPs and recovery actions, our framework is extensible to allow alternatives tailored towards specific use cases. For example, certain state changes in the system could be limited to updating requester-specific information in several compartments. Killing the requester process could automatically clean up such state changes. In future work, we could define a new class of SEEPs to identify such interactions in the system and a corresponding new reconciliation action to kill the requester process.

*Composable recovery policies:* The ability to classify the nature of inter-compartment communication by their respective SEEP types allows co-existence of multiple composable recovery policies. For example, a recovery window may

| Server | *Base memory usage (kB)* | *+clone (kB)* | *+undo log (kB)* | *Total Overhead (kB)* |
|---|---|---|---|---|
| PM | 628 | 944 | 1 | 945 |
| VFS | 1,252 | 1,600 | 13 | 1,613 |
| VM | 4,532 | 18,032 | 24,576 | 42,608 |
| DS | 248 | 488 | 1 | 489 |
| RS | 1,696 | 5,004 | 1 | 5,005 |
| *total* | 8,356 | 26,068 | 24,592 | 50,660 |

TABLE VI. PER-COMPONENT MEMORY OVERHEAD.

gradually switch from strict recovery policies to more lenient recovery policies, depending on the series of SEEP interactions encountered on a given execution path. OSIRIS could then perform tailored recovery actions accordingly.

*Generality of the framework:* Our recovery framework is generic and can be retrofitted to other systems. Our LLVM-based instrumentation and recovery libraries can easily be reused in other settings. For example, the recovery framework is independent of inter-compartmental communication protocols used in the target system. This design makes our system more adaptable and lowers the cost of maintenance. However, the target system should be composed of modular and restartable components. Modular server applications, multi-tier distributed systems and applications are promising candidates.

*Fail-silent software faults:* Recovery in OSIRIS aims to protect the system from crashes due to software bugs in untested parts of the system. However, it cannot provide the same guarantees in case of bugs that cause silent data corruption. In this case, we cannot determine whether the last checkpoint contains a safe state to recover to. Moreover, since our current failure detection strategy is based on the occurrence of a crash or hang in the affected compartment, we cannot protect against non-fatal manifestations of bugs. Nonetheless, with better fault detection mechanisms in place (e.g., memory safety solutions), we can approximate a fail-stop model for a broader class of real-world faults.

*Recovery surface:* SEEP is a foundational element of our approach. While it is possible to tailor the SEEP mechanism for another system, a target system design that yields a high frequency of inter-compartmental communications with global side effects, is likely to result in a small recovery surface. However, given that such a design is unfavorable even for performance-sensitive systems, we believe this limitation does not greatly affect the general applicability of our approach.

*Performance:* As we have shown in Section VI, our recovery mechanism incurs very low overhead on our prototype. However, our system is just a research prototype and not a mature system. It is possible that relative overheads would work out differently on, say, a more efficient baseline system. Moreover, compartmentalization is a prerequisite for OSIRIS and, as it requires the overhead of switching between compartments, such systems are unlikely to reach the level of performance of a system such as Linux. That said, compartmentalization offers major reliability advantages to compensate for the loss in performance even without recovery. Nevertheless, to improve performance and balance the performance-reliability trade-off, one could, in principle, retrofit our compartment-based design to high-performance monolithic OS architectures, for example, using virtualization-based isolation as in VirtuOS [17].

## VIII. RELATED WORK

The ability to shield systems from software faults has been the subject of many prior research efforts. Techniques include software rejuvenation, checkpoint-restart, and component dependency tracking. These techniques have been applied for recovering from faults in device drivers, operating systems, server and multi-tier applications, and distributed systems. In this section, we discuss prior research work in each of these areas and how it relates to ours.

### A. Recovering by reboot/restart

Restartability is a method to recover from failures or even proactively avoid failures, particularly transient and aging-related bugs [42]. Such restarts can be performed at both the operating system and application level. Phase-based reboot [29] aims at speeding up rebooting the operating system by reusing system state from previous reboots. A reboot is expected to bring back the system to a known consistent state. However, it comes at the cost of losing prior execution state of the system and contributes to downtime. MINIX 3 [25], [8] applies fault isolation in its design to enable restarting only crashed drivers. This is effective for performing stateless recovery, whereas our approach can support fully stateful recovery for arbitrary OS components. Application recovery has also been made possible by design-level considerations to enable restarting only the affected constituent components [43], [30]. In the distributed systems domain, Neutron [44] takes a similar approach towards reviving by restarting components in sensor network systems. In our work, hardware-assisted fault isolation has a similar effect in organizing the operating system into individually restartable fault domains. Although restart-based systems are relatively efficient and have the benefit of being relatively simple to implement, they all suffer from loss of state, which means that they can only be applied to stateless components for transparent recovery purposes.

### B. Checkpoint-restart/rollback

Checkpointing enables reviving a system without losing much of its execution context. Checkpointing solutions have been implemented at the level of operating systems, individual applications, and distributed systems.

At the operating system level, CuriOS [18] takes a virtual memory isolated multi-server approach where server state is persisted in respective client-side memory. This allows affected servers to be restarted without losing clients' context. While this circumvents the need for checkpointing, it is only suitable for systems where frequent (per-request or more) accesses to the clients' address space from the server processes are inexpensive. Fine-grained fault tolerance [15] relies on power management code to record device states—while also

checkpointing device driver state. A crash causes the system to roll back to the last driver entry-point state and restore the corresponding device state. While this has the benefit of being able to reuse existing code, this solution only applies to drivers. In addition, enforcing entry-point execution on a copy of required driver and kernel state obtained through static analysis cannot scale well to generic and complex operating system's core components.

Checkpoint-based recovery has been proposed for application recovery in various forms [45], [19], [46], ranging from applying hypervisor-assisted techniques to fast in-memory checkpointing. Vogt et al. [22] describe user-space memory checkpointing techniques that rely on compiler-based instrumentation to support high-frequency checkpointing. OSIRIS relies on similar compiler-based techniques, but selectively disables the checkpointing instrumentation during out-of-window execution.

Fault tolerance is a well-studied topic in distributed systems. Other than redundancy- and replication-based fault tolerance, checkpoint/snapshot-based recovery and message logging are also popular in distributed systems. Participant nodes take local snapshots in a coordinated or uncoordinated fashion to achieve fault tolerance in the face of faults or intrusions in the system [20], [44], [47], [48]. Message logging and replay is another alternative applied to message passing-based distributed systems [49], [21]. The multi-server architecture coupled with message passing-based communication in OSIRIS resembles a miniature distributed system. However, high-frequency and near-instantaneous message transfers observed in an operating system setting make a request-oriented local checkpointing scheme more suitable for our design.

### C. Dependency tracking

Consistent recovery entails rolling back not just the component that failed, but also all its dependent components whose state may get invalidated due to the rollback that occurred in the crashed component. Nooks [24] performs runtime object tracking to track manipulations of kernel data structures by kernel extensions in order to clean them up, should the extension crash. This is to protect the entire kernel from faulty extensions. Swift et al. [14] introduce a shadow driver mechanism that monitors driver-kernel interactions and when a driver fails, it turns a crash into an error condition by servicing requests on its behalf. Akeso [19] organizes the Linux kernel in a request-oriented hierarchy of recovery domains. Inter-recovery domain dependencies are tracked at runtime. When a fault occurs, Akeso initiates recovery of the dependent domains and recovers the failed recovery domain. In distributed systems, optimistic recovery [21] uses causal dependency tracking to detect computational dependencies among participating processes, so that dependent processes can be also rolled back during recovery. However, runtime dependency tracking generally incurs nontrivial performance overhead. Inter-component/process dependencies may even lead to a cascade of component rollbacks, which requires special care to avoid a domino effect. The SEEP channel in our design eliminates runtime dependency monitoring and the associated complexities, thereby limiting performance degradation.

### D. Other techniques

In addition to traditional checkpoint/restart, prior research has looked at recovering from system crashes in various ways. For example, ASSURE [26] and REASSURE [27] reuse existing error handling logic in the application to turn crashes into erroneous function return values. Carburizer [50], in turn, uses shadow drivers to turn device failures into software errors and avoid kernel crashes. OSIRIS relies on similar error virtualization strategies, but gracefully propagates error conditions through the message passing interface in compartmentalized operating system architectures.

## IX. Conclusion

We presented a recovery strategy for fatal persistent software faults in compartmentalized operating systems that does not compromise on system state consistency. We demonstrated the effectiveness of our design, which trades off total recovery surface of the system for performance and design simplicity. The key idea is to limit recoverability to execution paths that do not affect the global state of the system, greatly reducing runtime complexity. This enabled us to limit the RCB size to only 12.5% of our OSIRIS prototype—demonstrating that its reliability goals are practically achievable. We implemented our recoverability mechanisms using LLVM-based instrumentation, which can be also reused for other compartmentalized systems. Our framework is customizable and allows new classes of SEEPs and recovery actions to be defined for new target systems. Our experimental results demonstrate that OSIRIS' design is practical and effective in consistently recovering from even persistent software faults.

Overall, OSIRIS demonstrates that balancing recoverability, runtime performance, and simplicity of the reliable computing base can be an effective strategy to enhance dependability of compartmentalized operating systems. To foster further research in the area and in support of open science, we are open sourcing our OSIRIS prototype, available at http://github.com/vusec/osiris.

## X. Acknowledgements

We would like to thank the anonymous reviewers for their comments. This work was supported by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 644571 and by the Netherlands Organisation for Scientific Research through the NWO 639.023.309 VICI "Dowsing" project and the NWO "Re-Cover" project.

## References

[1] A. Ganapathi and D. A. Patterson, "Crash data collection: A windows case study." in *DSN*, 2005, pp. 280–285.

[2] R. Matias, M. Prince, L. Borges, C. Sousa, and L. Henrique, "An empirical exploratory study on operating system reliability," in *SAC*, 2014, pp. 1523–1528.

[3] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *ISSTA*, 2002, pp. 55–64.

[4] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ISSTA*, 2004, pp. 86–96.

[5] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, "An empirical study of dormant bugs," in *MSR*, 2014, pp. 82–91.

[6] M. M. Swift, B. N. Bershad, and H. M. Levy, “Improving the reliability of commodity operating systems,” *ACM Trans. Comput. Syst.*, vol. 23, no. 1, pp. 77–110, Feb. 2005.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An empirical study of operating systems errors,” in *SOSP*, 2001, pp. 73–88.

[8] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, “Construction of a highly dependable operating system,” in *EDCC*, 2006, pp. 3–12.

[9] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, “Faults in linux: ten years later,” in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, 2011, pp. 305–318.

[10] S. Chandra and P. M. Chen, “Whither generic recovery from application faults? a fault study using open-source software,” in *DSN*, 2000, pp. 97–106.

[11] J. Gray, “Why do computers stop and what can be done about it?” in *The 5th Symposium on Reliablity in Dist. Softw. and Database Sys.*, 1985.

[12] A. Depoutovitch and M. Stumm, “Otherworld: giving applications a chance to survive os kernel crashes,” in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 181–194.

[13] S. Sundararaman, S. Subramanian, A. Rajimwale, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift, “Membrane: Operating system support for restartable file systems,” *TOS*, vol. 6, no. 3, p. 11, 2010.

[14] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, “Recovering device drivers,” *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 4, pp. 333–360, 2006.

[15] A. Kadav, M. J. Renzelmann, and M. M. Swift, “Fine-grained fault tolerance using device checkpoints,” in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1. ACM, 2013, pp. 473–484.

[16] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, “Safedrive: Safe and recoverable extensions using language-based techniques,” in *OSDI*, 2006, pp. 45–60.

[17] R. Nikolaev and G. Back, “Virtuos: an operating system with kernel virtualization,” in *SOSP*, 2013, pp. 116–132.

[18] F. M. David, E. Chan, J. C. Carlyle, and R. H. Campbell, “Curios: Improving reliability through operating system structure.” in *OSDI*, 2008, pp. 59–72.

[19] A. Lenharth, V. S. Adve, and S. T. King, “Recovery domains: an organizing principle for recoverable operating systems,” in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1, 2009, pp. 49–60.

[20] D. J. Sorin, M. M. Martin, M. D. Hill, D. Wood *et al.*, “Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery,” in *ISCA*, 2002, pp. 123–134.

[21] R. Strom and S. Yemini, “Optimistic recovery in distributed systems,” *TOCS*, vol. 3, no. 3, pp. 204–226, 1985.

[22] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum, “Lightweight memory checkpointing,” in *DSN*, 2015, pp. 474–484.

[23] G. C. Hunt and J. R. Larus, “Singularity: rethinking the software stack,” *SIGOPS OSR*, vol. 41, no. 2, pp. 37–49, 2007.

[24] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers, “Nooks: An architecture for reliable device drivers,” in *ACM SIGOPS European workshop*, 2002, pp. 102–107.

[25] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, “Failure resilience for device drivers,” in *DSN*, 2007, pp. 41–50.

[26] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, “Assure: automatic software self-healing using rescue points,” *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1, pp. 37–48, 2009.

[27] G. Portokalidis and A. D. Keromytis, “Reassure: A self-contained

[27] G. Portokalidis and A. D. Keromytis, “Reassure: A self-contained mechanism for healing software using rescue points,” in *IWSEC*, 2011, pp. 16–32.

[28] G. Janakiraman, J. R. Santos, D. Subhraveti, and Y. Turner, “Cruz: Application-transparent distributed checkpoint-restart on standard operating systems,” in *DSN*, 2005, pp. 260–269.

[29] K. Yamakita, H. Yamada, and K. Kono, “Phase-based reboot: Reusing operating system execution phases for cheap reboot-based recovery,” in *DSN*, 2011, pp. 169–180.

[30] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, and R. Gowda, “Reducing recovery time in a small recursively restartable system,” in *DSN*, 2002, pp. 605–614.

[31] C. Giuffrida, L. Cavallaro, and A. S. Tanenbaum, “We crashed, now what,” in *HotDep*, 2010, pp. 1–8.

[32] D. Vogt, A. Miraglia, G. Portokalidis, H. Bos, A. Tanenbaum, and C. Giuffrida, “Speculative memory checkpointing,” in *Middleware*, 2015, pp. 197–209.

[33] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *CGO*, 2004, pp. 75–86.

[34] M. Engel and B. Dbel, “The reliable computing base: A paradigm for software-based reliability,” in *Workshop on SOBRES*, 2012.

[35] “A unixbenchmark suite, the original byte unix benchmark suite, updated and revised by many people over the years.” https://github.com/kdlucas/byte-unixbench, accessed: July 24th, 2015.

[36] “Minix 3 source repository,” http://git.minix3.org/index.cgi?p=minix.git.

[37] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, “Edfi: A dependable fault injection tool for dependability benchmarking experiments,” in *PRDC*, 2013, pp. 1–10.

[38] H. H. M. H. J. Liedtke and S. S. J. Wolter, “The performance of micro-kernel-based systems,” in *SOSP*, 1997.

[39] J. Liedtke, *On micro-kernel construction*. ACM, 1995, vol. 29, no. 5.

[40] J. Liedtke, “Improving ipc by kernel design,” in *SIGOPS OSR*, vol. 27, no. 5, 1994, pp. 175–188.

[41] J. Wu, H. Cui, and J. Yang, “Bypassing races in live applications with execution filters.” in *OSDI*, vol. 10, 2010, pp. 1–13.

[42] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, “Software rejuvenation: Analysis, module and applications,” in *FTCS*, 1995, pp. 381–390.

[43] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, “Microreboot-a technique for cheap recovery.” in *OSDI*, vol. 4, 2004, pp. 31–44.

[44] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr, “Surviving sensor network software faults,” in *SOSP*, 2009, pp. 235–246.

[45] M. Lee, A. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik, “Hypervisor-assisted application checkpointing in virtualized environments,” in *DSN*, 2011, pp. 371–382.

[46] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, “Rx: treating bugs as allergies—a safe method to survive software failures,” in *SIGOPS OSR*, vol. 39, no. 5, 2005, pp. 235–248.

[47] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo, “Highly available intrusion-tolerant services with proactive-reactive recovery,” *TPDS*, vol. 21, no. 4, pp. 452–465, 2010.

[48] A. Agbaria and R. Friedman, “Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations,” *Cluster Computing*, vol. 6, no. 3, pp. 227–236, 2003.

[49] A. Borg, J. Baumbach, and S. Glazer, “A message system supporting fault tolerance,” *ACM SIGOPS Operating Systems Review*, vol. 17, no. 5, pp. 90–99, 1983.

[50] A. Kadav, M. J. Renzelmann, and M. M. Swift, “Tolerating hardware device failures in software,” in *SOSP*, 2009, pp. 59–72.