Reboot-based Recovery of Unikernels at the Component Level

Takeru Wada
TUAT
Tokyo, Japan
takew@asg.cs.tuat.ac.jp

Hiroshi Yamada
TUAT
Tokyo, Japan
hiroshiy@cc.tuat.ac.jp

***Abstract*—The unikernel is a library operating system (OS) where OS functions are linked to the target applications. Making the unikernel layer as reliable as possible is mandatory because it controls the linked application's execution. However, like commodity OS kernels, the unikernel suffers from software bugs and non-deterministic hardware failures. The current standard for recovering a failed unikernel is reboot-based and it involves restarting the whole unikernel-linked application, which leads to service stops and long downtimes. This paper presents VampOS that performs efficient reboot-based recovery of the unikernel layer. VampOS forces the unikernel components to interact with each other in a message-passing manner to restart *only* the damaged one while keeping the others and the application running. We prototyped VampOS on Unikraft 0.8.0 and QEMU 6.1.50. The experimental results show that the prototypes for four applications effectively recover the failed components with almost zero downtime.**

***Index Terms*—Reboot-based Recovery, Unikernel, Software Rejuvenation**

I. Introduction

The unikernel [24], [26], [31], [41], [57] is a library operating system (OS) where OS functions are linked to the target applications like libraries. The unikernel is suitable for modern cloud platforms because each virtual machine (VM) offered by a cloud vendor typically runs only one application and uses some parts of the OS functionalities. This offers several benefits. First, the overhead of system call issues is mitigated since the unikernel runs in the same protection mode as the applications, so that the system calls do not cause mode changes. Second, the memory footprint of the OS layer is small since the applications can link only OS components required for their execution, which is different from the widely used monolithic OS kernels that contain full OS functionalities. Third, the trusted computing base (TCB) of the unikernel-linked applications can be made small by removing unnecessary OS components, which means attack surfaces are reduced.

Making the unikernel layer as reliable as possible is mandatory because it controls the linked application's execution on top of the hypervisor. Failures and error propagation at the unikernel affect the applications. For example, even if the application layer is undamaged, a crash of the unikernel due to non-deterministic memory errors aborts its execution. Also, when software bugs in unikernels cause memory leaks, the performance of the applications is significantly degraded. Because of their complex and ever-changing functionalities, OS components are still far from being bug-free despite numerous efforts in the past decades [2], [9], [52]–[54], [58], [62]. Unikernels, whose source code is much smaller than commodity OS kernels, suffer from software bugs [10], [45], [66]. For instance, a bug found in Unikraft causes memory leaks in ukallocbuddy [67], while another one triggers a crash by an invalid pointer reference in musl [65].

The current standard for recovering a failed unikernel is reboot-based; it is often the only remedy that disposes of the current states of the unikernel-linked application and initializes them from scratch. Reboot-based recovery is, however, expensive and sometimes unacceptable for stateful applications. Since it restarts the *whole* unikernel-linked application, undamaged unikernel components and the application have to be initialized. It thus loses the applications' running states, such as the session information in the web servers and data items in in-memory databases, leading to sudden service stops and long downtimes [18]. Since unikernels are used as core system components, e.g., the critical service domain [44] and OS servers [48], an efficient recovery method from failures at runtime is needed to achieve high availability. Software mechanisms [4], [15], [38], [64] for rebooting the OS kernel without restarting the running applications have been explored, but the implicit assumption of their designs is that the internals of the target OS kernels are fixed; this assumption is not acceptable for unikernels whose OS components vary among applications.

This paper presents *VampOS* for efficient reboot-based recovery of the unikernel layer. VampOS allows us to perform component-level reboots of the unikernel. The key idea behind it is to exploit the unikernels' customizability; modern unikernel internals are so componentized that their components are highly independent and interfaces between them are well-defined. VampOS, whose components interact in a message-passing manner, restarts only the target components while keeping the others and the linked application running. To maintain the consistency between the restarted and running components, it restores the running states of the restarted components by logging their interactions and replaying the logs.

The contributions of this paper are as follows:

- We present VampOS, which reboots the unikernel at the component level. Compared with state-of-the-art methods, our approach has unique characteristics. VampOS

allows us to restart the unikernel layer without rebooting the linked application. It is applicable to any unikernel component. It makes the downtime incurred by its reboot-based recovery as short as possible (§IV).
- We show software mechanisms for efficient reboot-based recovery of the unikernel layer. To avoid disturbing running components when rebooting a failed component, VampOS forces its components to communicate in a message-passing style and schedules the threads assigned to each component. At the same time, it offers memory isolation between the components to prevent errors from propagating out of the failed one. It also provides mechanisms, including dependency-aware scheduling, session-aware log shrinking, component merging, and checkpoint-based restoration, to minimize the runtime overhead and downtime for recovery (§V).
- We prototyped VampOS on Unikraft 0.8.0 and QEMU 6.1.50. The prototype supports four applications: *SQLite*, *Nginx*, *Redis*, and *Echo*. Using them, we conducted experiments to show the effectiveness of VampOS. The experimental results show that the performance penalty caused by VampOS is up to 1.4× and that the prototypes efficiently perform reboot-based recovery in two scenarios. In the first scenario, VampOS-based Nginx achieves software rejuvenation without any loss of connections, whereas the default full reboot loses 25.1% connections across the rejuvenation. In the second scenario, VampOS-based Redis recovers from an injected failure at the unikernel layer with almost zero penalty to throughput and latency. In contrast, the full reboot entails a signification performance degradation (§VI and §VII).

II. Motivation

A. Reboot-based Recovery of Unikernels

The unikernel is linked to the applications and sends requests for computational resources to the underlying hypervisor via hypercalls in the same protection mode as the applications. The unikernel shares the address space with the linked applications; typical unikernels support single-process applications, not multi-process ones. Researchers have studied unikernel architectures and their applicability to secure architectures [41], supports for various applications [17], [26], [33], [49], [63], [69], [74], multi-tenant controllers for embedded clouds [40], lightweight network function virtualization [42], lightweight privilege functionalities [44], [48], improvements to modularity [31], [32], workload offloading to various embedded boards [50], mechanisms for supporting multi-processes [37], [39], and enhancement of isolation between components [35], [57].

In this paper, we try to answer the following question: *How can we perform reboot-based recovery of unikernels efficiently?* Here, it is better to make the unikernel layer as reliable as possible. Since unikernels are intermediate between applications and hypervisors to obtain computational resources from the hypervisor, the linked applications cannot keep running when their unikernel layers fail. Reboot-based recovery is a simple but powerful way to improve the availability of computer systems. Periodic reboots of the target software, a.k.a., software rejuvenation [11], [12], [21], reclaim stale or leaked resources such as memory leaks and descriptor leaks to proactively prevent failures such as crashes and hangs. Reboots are used to recover the affected target software reactively. Reboot-based recovery can be used without analyzing the root causes of the failures and is often the only remedy against software errors and failures for end users. To extend its applicability, researchers have proposed effective software mechanisms for various software layers such as java applications [8], in-memory databases [22], [23], operating systems (OSes) [6], [15], [28], [64], [71], and hypervisors [29], [30], [34].

However, efficient reboot-based recovery of unikernels does not come for free. Since unikernels are so tightly coupled with the applications and their memory segments are shared, rebooting the unikernel layer involves a linked application restart, which eliminates the whole contents of memory in the unikernel-linked application even if the application memory is undamaged. The conventional reboot quickly recovers stateless and small memory footprint applications like web servers from their failures, but is unsuitable for memory-intensive and stateful applications. For example, in-memory key-value stores, like memcached and Redis, typically utilize tens to hundreds of GB of memory, and restoring running states is time-consuming. A research paper reported that restarting only 2% of Facebook's servers at a time prolongs the restart duration to about 12 hours, during which time users see only partial query results [18]. Also, some research has been aimed at making the core of the software systems, such as the critical service domain [44] and OS function servers [48], unikernel-based. Since their failures affect all of the running software, quick recovery of the failed component directly results in the availability of the system. These issues motivate us to explore efficient reboot-based recovery of unikernels.

B. Fault Model

VampOS recovers from fail-stop faults of the unikernel components. Similar to previous efforts based on restarts, it keeps executing when the faults are non-deterministic. Examples of software ones are deadlocks and crashes caused by race conditions that occur for particular scheduling decisions, while non-deterministic hardware faults include bit flips in memory and registers. Re-execution by restarting the faulty component and replaying the same inputs will avoid triggering these faults again. Because of its software approach, VampOS-based recovery differs slightly from those of regular reboot-based recovery in that VampOS cannot handle failures that require hardware reboots, such as firmware crashes.

VampOS-based recovery covers faults due to software aging, such as memory leaks and fragmentation. It is known that proactive restarts of the target software prevent crashes and hangs caused by software aging. VampOS periodically eliminates the software aging phenomenon in the unikernel without restarting the application components.

VampOS fail-stops when the rebooted components face the failure again. Similar to the regular reboot-based recovery, it fails to recover from deterministic faults, such as permanent hardware failures and deterministic bugs. Here, unikernel-linked applications after VampOS-based recovery must face these failures again, since the restoration phase feeds the same inputs to the restarted components and activates the fault in the same way as the previous execution. Note that VampOS tries to reboot only the failed component; it does not detect or recover the root-cause components that triggered the failure. Such faults are out of the scope of VampOS. As well, it does not target the validity of the arguments or function call sequences used across components, but rather prevents components from undermining any invariant maintained by such components. Although our protection mechanisms, as described in §V, can detect failures caused by argument errors, VampOS does not cover all of such failures.

III. Related Work

There are a number of approaches that aim to improve the reliability of the unikernel. Unikraft [31] enhances the modularity of the unikernels to improve their customizability and reliability by making their TCB as small as possible. CubicleOS [57] and FlexOS [35] strongly isolate unikernel components to prevent errors from propagating from faulty components. These approaches complement ours in attaining reliable unikernel-linked applications. They do not guarantee that the unikernels will never suffer from aging-related bugs, whereas our approach can mitigate their adverse effects by making the unikernels efficiently rebootable.

Some research has focused on the reliability of microkernels. Theseus [5] and ReadLeaf [46], both of which are written in Rust from scratch, respectively isolate the kernel components in a fine-grained manner and avoid holding states in the system server for each other to enhance the fault recovery capability. TxIPC [38] recovers the stateful servers on the microkernel by aborting all the updates made by the IPC. OSIRIS [4] recovers stateful system servers without modifying the source code by transparently recording undo logs at runtime. CuriOS [14] stores client-specific states in client-associated but client-inaccessible memory in order to restore the states of the system server. These approaches are aimed at the microkernels, while VampOS targets recovery of the unikernels. In addition, these approaches do not handle software aging.

Moreover, some researchers have explored recovery and isolation mechanisms for monolithic kernels. Akeso [36] is a kernel-level, request-oriented mechanism in that it handles recoveries at the request level, such as through system calls or interrupts. It rolls back the kernel state to the beginning of the function and makes the function return an error. Membrane [59] performs reboot-based recovery of file system components, while Shadow Driver [60] restarts faulty device drivers and restores their runtime states by monitoring the drivers' interactions with the OS kernel and devices. Nooks [61] isolates device drivers from the other OS kernel components. VampOS does not target specific components. IskiOS [19] and HAKC [43] isolate memory segments and loadable kernel modules by using CPU-supported lightweight protection domains like Intel MPK. VampOS offers an efficient recovery method after detecting failures.

Barrelfish [3] is a message-passing-based OS designed for high multicore scalability. Each kernel thread executing an OS component is pinned on a core and sends/receives messages to/from each other. Barrelfish/DC [73] allows Barrelfish to perform dynamic reconfigurations of the kernel threads and core numbers. When an OS component fails and reboot-based recovery is needed, the recovery on the OSes restarts all of the components. In contrast, VampOS reboots only the damaged components.

Software mechanisms for efficiently rebooting the target software have been explored. Some of them focus on efficient OS kernel reboots. KUP [25] keeps the application running across the OS kernel reboots. It takes snapshots of the target applications in memory with user-space checkpointing mechanisms [1] and restores them after the OS reboot. This approach is not applicable to unikernel-linked applications. Unlike conventional OS kernels, the checkpointing process cannot run with the target application on a unikernel that supports only a single address space shared with the linked application. Also, taking snapshots of only the application layers is difficult outside unikernel-linked applications because the boundary between the applications and the unikernels is unclear.

Otherworld [15] and Dwarf [64] restart the OS kernels while keeping applications running. In restarting the OS kernel, Otherworld launches the newer kernel and forces it to salvage kernel memory objects of the process contexts in the older kernel's memory and restore its internal structures. Dwarf stores the cores of process contexts for the target applications to the hypervisor, launches the OS kernel on a newly launched VM, and forces the kernel to restore them while migrating the target applications from the old VM to the newly launched one. The assumption behind these approaches is that the target OS kernels are monolithic. Thus, they are not suited to the characteristics of the unikernels; since the unikernel's components vary between applications, the kernel objects for maintaining process contexts are not constant. These mechanisms have to be redesigned for each unikernel-linked application, which is a non-trivial task. In Otherworld, for example, we investigate memory objects required to keep running applications from their source code, memorize their addresses to reuse them, and develop a crash kernel that loads unikernel components, tracks the memory objects, and restores the kernel contexts based on them after the unikernel-linked applications crash. These steps have to be taken for each unikernel-linked application.

Microreboot [8] achieves fine-grained software reboots. To enable a microreboot, the target application is divided into small independent software components, which become units for the reboot. If rebooting a small component cannot recover from a failure, a bigger component will be rebooted. The implicit assumption of this approach is that each component

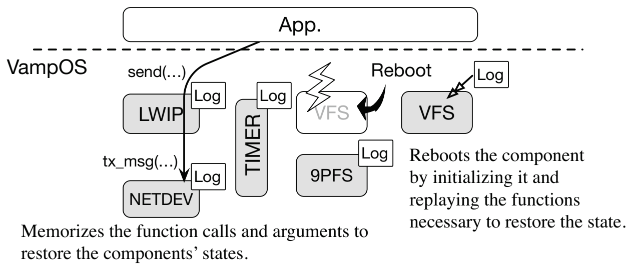

App.
VampOS
send(...)
Reboot
Log
LWIP
Log
Log
VFS
VFS
TIMER
tx_msg(...)
Log
Log
9PFS
NETDEV
Reboots the component by initializing it and replaying the functions necessary to restore the state.
Memorizes the function calls and arguments to restore the components' states.

Fig. 1. **Overview of VampOS.** *VampOS reboots the unikernel layer at the component level. To consistently restart stateful components, VampOS logs function invocations during their execution and replays the functions necessary for runtime state restoration.*

to be microrebooted is so stateless that the target applications consistently run across some components' reboots. However, the components of the unikernels are often stateful; thus, it is difficult for the linked applications to consistently run across rebooting stateful components such as file systems and networks.

Approaches for efficient OS reboots, including phase-based reboots [71], ShadowReboot [70], and CacheMind [28], reduce downtime and performance degradation incurred by the OS kernel reboots. Like regular OS reboots, they involve restarting all running applications and thus cause significant performance degradations in modern in-memory applications with large memory footprints. Approaches that support hypervisor rejuvenation [29], [30], [34] allow us to keep the running states of the virtual machines across the hypervisor rejuvenation. These approaches are complementary to ours.

IV. VampOS

This paper presents *VampOS* that reboots unikernels efficiently. VampOS is driven by the following design goals.

- **Reboots only the unikernel layer.** Different from conventional full reboots, VampOS reboots the unikernel components while preserving the contents of the applications' memory. This enables the applications to run consistently across VampOS-based reboots.
- **Does not depend on specific unikernel configurations.** Recovery mechanisms designed for specific unikernel configurations are unreasonable since the unikernel structures differ among applications. Our approach is applicable to any unikernel component.
- **Makes downtime as short as possible.** To minimize service disruption caused by reboot-based recovery, VampOS shortens the downtime of the applications as much as possible.

To meet these goals, VampOS exploits the customizability of the unikernels; i.e., unikernels offer numerous components, and the interfaces between components are well-defined so that the applications only select the components required to run. An overview of VampOS is shown in Fig. 1. VampOS reboots the unikernel at its component level and restores the running states of the restarted components in order to execute the linked applications consistently. Specifically, VampOS logs function calls between components by hooking their exposed interfaces, restarts the target one, and replays the selected function calls to restore its running states if it is stateful, as described in §V. Since VampOS chooses the functions necessary to restore the running states of the component, memory fragmentation and resource leaks caused by aging-related bugs, which are triggered by numerous resource allocations/releases for long time execution, are eliminated in the rebooted one. For example, in rejuvenating a file system component, VampOS restarts only the component and calls the selected functions inside the rebooted component.

VampOS is effective in the following scenarios:

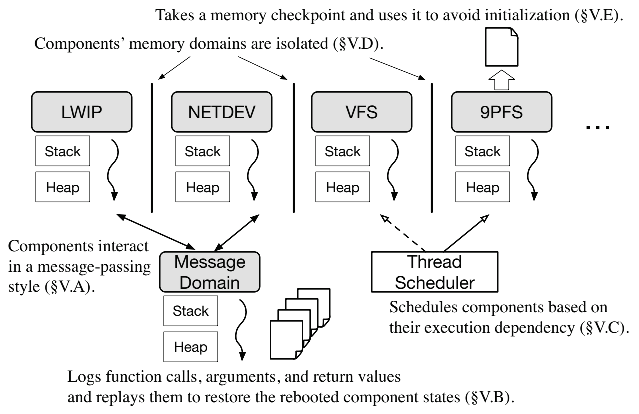
- **Recovering unikernels without initializing in-memory applications:** Since VampOS keeps the memory content of applications across its unikernel reboots, there is no need to restore data items in in-memory KVSes (Key Value Stores) or re-connect clients in web servers, both of which would otherwise cause service disruptions.
- **Frequent software rejuvenation:** The component-level reboot in VampOS is so lightweight that its service disruption is negligible. Accordingly, administrators can carry out VampOS-based reboots for software rejuvenation more frequently than in the case of a regular reboot.
- **Restarting only the failed unikernel component:** Unlike a whole reboot, VampOS reboots the damaged component and restores it while reusing the undamaged ones. To enforce this recovery, it also isolates each component to prevent an error propagation of the faulty component from damaging other components as much as possible.

Note that the goal of VampOS is to obtain the effect of reboot-based recovery, not to be completely compatible with regular reboots. We can perform VampOS's rejuvenation instead of rebooting the unikernel-linked application to rejuvenate the unikernel. Regular reboots are used for other purposes, such as software updates and reconfiguration. VampOS's mechanisms cannot be used for such purposes; regular reboots need to be used for them.

Designing VampOS poses the following technical challenges: How can we rejuvenate the unikernel at the component level? How can we restore the only target component? How can we restore the states of the rebooted stateful components efficiently? And how can we mitigate thread scheduling overhead? This paper tackles these challenges. Although the following discussion is based on the development of our prototype on Unikraft [31], we believe that it is general enough to be applicable to other unikernels such as IncludeOS [7] and OSv [26].

V. Design Details

VampOS offers software-level solutions for the design challenges. The design of VampOS is shown in Fig. 2. To efficiently reboot the unikernel at the component level, it binds a dedicated thread to each component and the threads communicate by message passing. It also isolates the memory domains of the components to prevent errors from propagating from faulty ones. To avoid affecting running components



Takes a memory checkpoint and uses it to avoid initialization (§V.E).
Components' memory domains are isolated (§V.D).
LWIP
NETDEV
VFS
9PFS
...
Stack
Heap
Components interact in a message-passing style (§V.A).
Message Domain
Thread Scheduler
Schedules components based on their execution dependency (§V.C).
Logs function calls, arguments, and return values and replays them to restore the rebooted component states (§V.B).

Fig. 2. **VampOS Design.** *VampOS forces components to communicate in a message-passing style to efficiently trigger their reboots. To consistently restore the rebooted components without having side effects on the running ones, VampOS performs encapsulated restoration and checkpoint-based initialization. In addition, it isolates the memory region of each component to prevent errors from propagating from the failed component to others.*

connected to the rebooted component, it encapsulates the rebooted component in playing the log. In addition, it uses snapshots of the component memory images just after their initialization and plays only the selected functions that change the state of the target component.

A. Message-passing Component Interaction

We take into account the execution of the unikernel components to reboot them at the component level. Since the unikernels run as libraries, their components are executed via function calls. When a thread of the unikernel-linked application requests a file to be opened, it executes the file system-related components. A naive approach is a single-threaded one where the failed component is rebooted when the context faces a crash or hang. However, this cannot move the path for the recovery and thus cannot achieve a component reboot.

To address this issue, VampOS assigns threads to all the components, forcing them to interact with each other in a message-passing manner. In so doing, it reboots the target components while simultaneously keeping the other components running. When a failure occurs in a component, VampOS stops the thread, initializes the component, and reassigns a thread. This design is also tightly coupled with the component restoration for the VampOS-linked applications to continue to run consistently across the reboot, as described in §V-B.

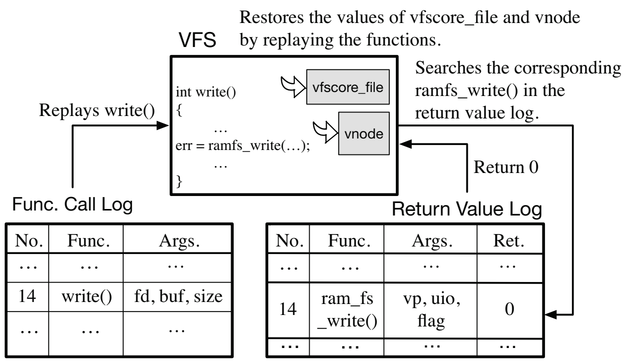
The components of VampOS, each with its own data, heap, and stack regions, are executed by their threads, not the callee thread context. Each thread invokes a function of the other components by passing its arguments to the component's thread. In the regular unikernel, when the application issues a open() exposed by the VFS component to open a file, its thread context jumps to the open() in the VFS component. In contrast, a thread of the VampOS-linked application passes the arguments of the open() to the VFS's thread, and it executes the open() with the passed arguments. VampOS hooks the interfaces exposed by the components, extracts the arguments, puts them in shared memory (*message domains*) between components, and notifies the target thread. The thread executes the requested function with the arguments on the message domain. The message thread plays the role of monitoring the running components and triggering their reboots if they face failures by using a simple detector that checks the states in a heart-beat manner; illegal memory accesses and panic() invocations transfer the control to error handlers and triggers the reboot. In addition, the current prototypes have a simple detector of component hangs that periodically checks the processing time of a pulled message. They treat components as hangs when the processing time exceeds a threshold (= 1.0 s). Also, they do nothing for some components, such as LWIP, that wait for external events like network connection requests. To detect component failures precisely, we can use sophisticated runtime failure sensors [13], [16], [47], [51].

We note that VampOS assigns threads to components on demand. Conceptually, each component is executed by a dedicated thread in VampOS. With this design, the component cannot process messages when the thread is blocked, e.g., lock release waiting, inside the component, sometimes causing deadlocks. To avoid such situations, VampOS attaches a newly-spawned thread when dispatching it. Even if a running thread is blocked in a component, another thread is allocated by the scheduler to handle the arriving message.

B. Encapsulated Restoration

We must pay attention to the running states of the rebooted components. Many OS subsystems, such as file systems and networks, are stateful, and rebooting a unikernel component of such a subsystem fails to keep the application and the other components running consistently. When we reboot a VFS component that maintains the file offset, the file operation of the application after the rejuvenation cannot be done correctly since the file offset is initialized to be zero. From the viewpoint of software rejuvenation, it does not make sense to take a memory snapshot of the target component just before it restarts and its running state is restored. If the system states change and the function returns a different value from the previously issued one before replaying the log, the rebooted component will not have the same states as before reboot. The restored offset to the shared file with multiple threads is wrong if a thread moves its offset using SEEK_END and then another thread changes the file size before replaying the log. This is because the restoration respawns the aged memory image, which can include memory/descriptor leaks and memory fragmentation.

To avoid the inconsistency caused by stateful component reboots, we restore the running states of the stateful components after their restart. For this restoration, VampOS memorizes function calls in the target components invoked by the other components and plays the logs of the components just after their reboots. Different from replaying all the called functions until reboot, VampOS invokes only the selected function calls to shorten the restoration time and avoid regenerating error


Restores the values of vfscore_file and vnode by replaying the functions.
VFS
Searches the corresponding ramfs_write() in the return value log.
vfscore_file
vnode
int write()
{
…
err = ramfs_write(…);
…
}
Replays write()
Return 0
Func. Call Log
Return Value Log

Func. Call Log

No.	Func.	Args.
⋯	⋯	⋯
14	write()	fd, buf, size
⋯	⋯	⋯

Return Value Log

No.	Func.	Args.	Ret.
⋯	⋯	⋯	⋯
14	ram_fs _write()	vp, uio, flag	0
⋯	⋯	⋯	⋯

Fig. 3. **Encapsulated Restoration of Rebooted Component.** *To avoid changing the state of the running components, VampOS restores the rebooted stateful components without any invocation in the other ones. It replays the selected function calls in the log and feeds the corresponding return values.*

states caused by aging-related bugs such as memory leaks and fragmentation. It calls only enough functions to restore the running states of the components just before rejuvenation. Specifically, it skips functions that do not change the component states, such as file status reads (fstat()-related functions).

In addition, VampOS also logs the return values of functions in other components and uses them to restore the state of *only* the target component. A simple approach for replaying the functions is to call them with the logged arguments. In this approach, during a restoration, the component can invoke functions of other components and change their current states. These state changes are inconsistent with the running application and corresponding components, since they are triggered by restoring operations, not the regular operations. To avoid spilling the state-change operations from the rebooted component during its restoration, VampOS returns the logged return values of the other component's functions, instead of calling them. Fig. 3 is an overview of VampOS's component restoration. As the log is played, VampOS forces the restoring component to use the return values logged before the reboot.

VampOS also extracts runtime data from some of the stateful components to restore their states. A component restoration consistent with the other components and the application cannot be achieved simply by replaying the functions. For example, packet sequence numbers and ACK numbers in TCP connections, managed by LWIP, are given at runtime and updated via interactions with external communication partners. The function replaying initializes connections but cannot restore the TCP connection states. To handle this runtime state issue, VampOS tracks and saves specific data every time their updates are directly used in the component restoration. Although this optimization is an ad-hoc one for the components, the optimized components are not application-specific and can be used in other applications. From our experience in designing VampOS on three real-world applications, this special data saving is required by only one component, LWIP, where the packet sequence and ACK numbers are stored.

C. Dependency-aware Scheduling

Compared with the default that uses the target component via function calls, message passing incurs additional overhead due to thread scheduling. Since each component has its own thread context, the component thread that receives a message cannot run until the internal scheduler dispatches it. The round-robin scheduler becomes less efficient when there are more unikernel components. Due to the simplicity and responsiveness of messages, as in a previous study [35], the dispatched components poll their messages. Since this design wastes the CPU time when messages do not arrive, we can implement yield() and have the component calls it to avoid busy-waiting. SQLite, for example, consists of seven components and VFS invokes write instructions when storing an item in its database. 9PFS may wait to be scheduled when the round-robin scheduler dispatches other components.

To ensure both the design simplicity and high message responsiveness without busy-waiting as much as possible, VampOS uses *dependency-aware scheduling* to dispatch component threads in an efficient manner. Dependency-aware scheduling exploits the dependency between components; a component invokes a subset of all the running components and thus we can infer the components for the next scheduling from the currently executing component. For example, VFS passes messages to two components (9PFS and LWIP), while LWIP communicates with VFS and NETDEV. Dependency-aware scheduling dispatches components with this correlation specified in advance. When a component is running and sends a message, the scheduler selects the correlated components as scheduling candidates and dispatches one at a time. When VFS is running, our scheduler preferentially dispatches 9PFS and LWIP.

In addition, dependency-aware scheduling takes into account the log requirement of the next scheduling candidate. The scheduler dispatches the message thread to store the arguments before dispatching a component. When the executing component has stored the message in the message domain, the scheduler dispatches the message thread if the next candidate requires the argument logging for its restoration. Then, the next component starts running. In sending the return value, the scheduler dispatches the message thread to preserve it. All the interactions between components are done in the above way.

D. Component-level Protection Domains

VampOS isolates the running components to confine error propagation from the faulty component. Since unikernel's components run in the same address space and at the same privilege level, a faulty component can illegally modify the memories of the other components. In this case, even if the component is rebooted, the recovered applications cannot run correctly since the other ones access the damaged memory. Such an error propagation triggers reboots of multiple components and, at worst, damages the application layer.

To prevent a faulty component from propagating its errors, we take inspiration from previous unikernel's isolation mechanisms [35], [57] and make VampOS enforce memory isolation with low overheads by using lightweight in-process protection mechanisms, such as Intel Memory Protection Keys (MPK) and ARM Memory Domains. These are ISA extensions that manage access permissions on groups of pages. For example, Intel MPK, which is used in our prototypes, assigns a 4-bit key to each virtual page by extending the page table structures, and adds a new 64-bit PKRU register that defines the access permissions to all pages on a key. VampOS maps the memory regions of each component thread into a separate protection key and the thread scheduler dynamically manages the access permissions. A component thread cannot access another components' memory since the thread's key is permitted to only access its heap, text, and stack. The threads change the permission of message domains to writable only when the messages are sent to other components. The receivers keep the message domains read-only since their accesses to the regions are to read the sent messages only. In switching a component thread to another one, the thread scheduler also changes the current MPK tag to the corresponding tag.

The tasks of the message thread are summarized as follows. First, the thread maintains message buffers and the logs of function calls and return values in a message domain isolated from the other components. Management of these objects inside each component is not reasonable since the errors of the faulty component can damage their buffers and logs, and the recovery procedure can be done using damaged ones. The thread also releases buffers when they are used by the target component and are not needed for the restoration. Second, the message thread triggers the VampOS-based reboot of the faulty component. In detecting the component's failure, it initializes the component and performs the encapsulated restoration.

Physical protection keys can be fewer than the running components. For example, there are 16 protection keys in Intel MPK and 32 in ARM Memory Domains. When the components are more than the physical keys, VampOS fails to isolate the internals of VampOS-linked applications. The maximum number of the running components in our experiments was up to 12 in Redis and Nginx. To isolate more components than keys, we can use techniques to increase the number of protection keys [20], [55], [72].

E. Checkpoint-based Initialization

The regular component restart that executes their shutdown and boot routines is unsuitable for our component-level rejuvenation. These routines involve function calls in the other components and hardware operations, resulting in changes to their running states. For example, VFS, which interacts with file system and network components such as 9PFS and LWIP, triggers the initialization of corresponding subsystems in its boot phases, which means the VFS restart affects currently running components.

To address this issue, VampOS leverages the memory images of the components *just after* their initialization, borrowing an idea from the phase-based reboot [71]. The phase-based reboot obtains the rejuvenation effect by restoring the memory images of the system in its boot phase. VampOS offers a component-level checkpointing mechanism and takes memory snapshots of the initialized components. In rebooting a component, it restores the corresponding memory snapshot and performs our log replay as described in the previous section.

F. Other Optimizations

Component Merging: The message-passing approach additionally incurs runtime overhead for scheduling component threads and pushing/pulling messages. To mitigate the overhead, VampOS supports component merging that combines components that interact with each other. The overhead mitigation of the component merging depends on the interaction frequency of the target components; it becomes more effective as more interactions occur between the components since the message-passing procedure can be skipped. For example, when we merge VFS and 9PFS, their functions such as uk_9pfs_lookup() and uk_9pfs_write() are invoked as function calls. The merged component shares a single thread and the memory region, such as a heap and stack, which means that a single MPK tag manages the memory domain. In rebooting a composite component consisting of three primitive components, VampOS loads the snapshots of the three primitive components and replays their logs on each component.

Session-aware Log Shrinking: The log sizes naturally become large over time, especially in long-running applications such as web servers and in-memory KVSes. This incurs more memory space overhead and longer log replies. To address this issue, VampOS performs the session-aware log shrinking to decrease the log sizes. The session-aware log shrinking prunes log entries, function calls, and their return values unnecessary for the component restoration. The key behind this method is that some functions, called *canceling functions*, make it unnecessary to preserve a series of function calls that change the component states. For example, write() and read() update the file structure, including the offset value, in VFS, and their changes are not needed for VFS restoration after issuing close() for the corresponding file descriptors. Another example is that socket states are no longer required after closing the corresponding connections.

VampOS monitors the calls of the canceling functions and eliminates the log entries when such a function is invoked. VampOS specifically removes read() and write() logs when the corresponding file descriptor is closed. In addition, VampOS takes the same or similar effect as forcing components to invoke canceling functions to avoid increasing the log sizes when the functions are not invoked for a long time. We can shrink a series of write() by preserving the offset and contents to write, leading to shortening of the log replay. For ease of implementation, our prototypes restore the current states of the components affected by the function invocation after calling the canceling function intentionally. For example, VampOS extracts and resets the offset value in VFS after calling close() to shrink the logs. VampOS does so when the log size is over the threshold (= 100 entries) specified in advance. Although

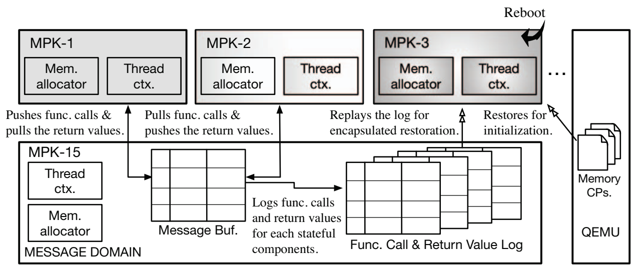

Reboot
MPK-1
Mem. allocator
Thread ctx.
MPK-2
Mem. allocator
Thread ctx.
MPK-3
Mem. allocator
Thread ctx.
...
Pushes func. calls & pulls the return values.
Pulls func. calls & pushes the return values.
Replays the log for encapsulated restoration.
Restores for initialization.
MPK-15
Thread ctx.
Mem. allocator
Message Buf.
Logs func. calls and return values for each stateful components.
Func. Call & Return Value Log
MESSAGE DOMAIN
Memory CPs.
QEMU

Fig. 4. **VampOS Implementation.** *Each component in VampOS consists of thread contexts and memory regions and is isolated by the Intel MPK feature. The message domain maintains a message buffer for message passing communication and function-call and return-value logs for encapsulated restoration.*

this implementation applies file modifications to the file system and the result conflicts with the application logic, the four applications in our case studies ran consistently.

Note that log entries could not be removed even with the session-aware log shrinking that effectively prunes log entries of the components in our case studies, including server applications such as in-memory KVS (Redis) and web servers (Nginx), which are long-running; clients close their connections after fetching the target contents, and thus the log shrinking removes the session-related log entries. For example, if a component does not have canceling functions, its log size becomes bigger over time. At worst, when the component has few state-unchanged functions that skip being logged, VampOS's rejuvenation can regenerate software aging due to long log replaying by the encapsulated restoration. The current prototypes use the regular reboot to rejuvenate the unikernel-linked applications. Although we have yet to face such a component, exploring ways to handle this case is one of our future work.

VI. Implementation

We prototype VampOS on Unikraft 0.8.0 and QEMU 6.1.50, running on x86 CPUs. The VampOS mechanism leverages the Intel MPK feature to create protection domains for components. Unikraft is a framework for building unikernels. It has a modular architecture in which each component implements a single OS function (the virtual file system, file system back-ends, memory allocation, network stack, etc.). Components are selected at compile time and linked into a monolithic image together with the application. When executing the VampOS-linked application, the current implementation allocates fixed memory to each component and loads its segments into the memory. Then, it assigns memory for a message domain and a thread to each component and launches the thread scheduler. The upper limit of memory is set to 88 MB from our experiments.

An overview of our implementation is shown in Fig. 4. Each component has thread contexts, a memory allocator to create its own heap, and an MPK key to isolate its memory region. The static data in the .bss and .data regions is placed via a compiler annotation section("component-section-name"). The message domain exposes message passing interfaces,

TABLE I
Components Used in the Current Prototypes.

Component	Description
VFS	Exposes POSIX APIs for file systems and networks.
LWIP	A network subsystem that implements a protocol stack.
9PFS	A file system based on the 9P network protocol.
PROCESS	Implements process-related functions like getpid().
SYSINFO	Implements functions for system information such as uname()
USER	Implements functions user information like getuid().
TIMER	Implements time-related operations.
NETDEV	Implements low packet operations.
VIRTIO	A device driver for the virtio device.

vo_push_msgs() and vo_pull_msgs(). vo_push_msgs() pushes function requests or return values into the messages buf log and logs them into the function call and return value logs if necessary, while vo_pull_msgs() pulls the requested functions and the values. Each log entry is different since the sizes of the return values and arguments depend on the function calls. Also, the current implementation does not have an upper limit to the log size, but the session-aware log shrinking threshold is 100 entries. For checkpoint-based initialization, we reuse the QEMU snapshot feature and our module running inside QEMU takes a memory snapshot of the component unit.

We apply VampOS to four real-world applications supported by the Unikraft project: *SQLite*, *Nginx*, *Redis*, and *Echo*. All of them are running on the platform code. The details are as follows.

SQLite: SQLite is a widely used relational database management system. SQLite handles SQL queries sent via its functions and stores data items into a database file. It comprises seven components: PROCESS, SYSINFO, USER, TIME, VFS, 9PFS, and VIRTIO. We integrate the software mechanisms of VampOS into SQLite's components. The VampOS-based SQLite uses ten MPK tags (application + seven components + message domain + thread scheduler).

Nginx: Nginx is a well-known web server with nine components: PROCESS, SYSINFO, USER, NETDEV, TIME, VFS, 9PFS, LWIP, and VIRTIO. The VampOS-based Nginx uses 12 MPK tags (application + nine components + message domain + thread scheduler).

Redis: Redis is a representative in-memory KVS used in modern cloud platforms. It consists of nine components: PROCESS, SYSINFO, USER, NETDEV, TIME, VFS, 9PFS, LWIP, and VIRTIO. The VampOS-based Redis uses 12 MPK tags (application + nine components + message domain + thread scheduler).

Echo: Echo is a simple server that sends the same messages received from clients. Its components are PROCESS, USER, NETDEV, TIME, VFS, LWIP, and VIRTIO. The VampOS-based Echo uses ten MPK tags (application + seven components + message domain + thread scheduler).

We make the components used in these applications VampOS-aware, as shown in Table I. Our prototypes reboot four stateless components (PROCESS, SYSINFO, USER, and NETDEV) by restarting them without function call logging or encapsulated restoration. We integrate these functionalities into three stateful components: VFS, LWIP, and 9PFS. We note

TABLE II
FUNCTION CALLS LOGGED FOR ENCAPSULATION REBOOTS.

Component	Logged Function Calls
VFS	create(), open(), write(), pwrite(), read(), pread(), close(), mount(), fcntl(), lseek(), vfscore_vget(), pipe(), ioctl(), writev(), fsync(), vfs_alloc_socket()
LWIP	socket(), bind(), listen(), connect(), getsockopt(), setsockopt(), shutdown(), sock_net_close(), sock_net_ioctl()
9PFS	uk_9pfs_mount(), uk_9pfs_unmount(), uk_9pfs_open(), uk_9pfs_close(), uk_9pfs_lookup(), uk_9pfs_inactive(), uk_9pfs_mkdir()

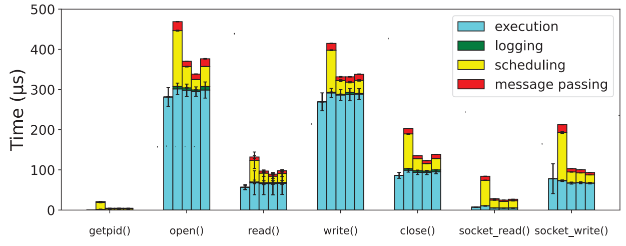


Fig. 5. **System Call Overheads.** *The graph shows the execution time in each system call, Unikraft, VampOS-Noop, VampOS-DaS, VampOS-FSm, and VampOS-NETm. The runtime overheads depend on system calls, and our optimizations effectively mitigate the overheads.*

that our prototypes do not reboot VIRTIO because its states are shared with the host Linux and the rebooted VIRTIO does not consistently work even with our restoration feature. We discuss this issue in §VIII.

VampOS memorizes the function calls summarized in Table II and the return values of the functions in other components. In rebooting the three components, the prototypes initialize them, assign threads, and perform the encapsulated restoration. Additionally, the prototypes apply the checkpoint-based initialization to VFS and LWIP since their initialization procedures involve changes in other components. They perform session-aware log shrinking upon close() and functions triggered by its issues. Our VFS removes entries for operations on the closed fd number. 9PFS and LWIP in the prototypes dispose of logged function calls to the file and the network connections corresponding to the closed fd.

VII. EXPERIMENTS

To show the effectiveness of VampOS, we conducted several experiments with the prototypes. We used a PowerEdge R740 which has 12 Intel Xeon Silver 2.20 GHz cores and 92 GB of memory. We turned off the hyperthreading feature. The experiments try to answer the following fundamental questions: 1.) How much performance and space overhead does VampOS incur? 2.) How long does VampOS take in component reboots? 3.) How much overhead does VampOS cause on real-world applications, and 4.) How effective is VampOS in reboot-based recovery scenarios?

A. System Call Overhead

To confirm the runtime overhead incurred by VampOS, we measure the execution times for system calls on our prototypes and vanilla Unikraft. In particular, we measure the execution times and log entry increases for seven system calls: getpid(), open(), write(), read(), close(), socket_read()(read() for a socket), and socket_write()(write() for a socket). The number of component transitions in these system calls is 4, 41, 65, 28, 37, 50, and 63, respectively. In write() and read(), we issue a write/read request of 1 byte of a file. Open() and close() create and release the file descriptor for a file. In socket_read() and socket_write(), we receive/send 222 bytes network messages. For comparison, we prepare four configurations of VampOS: *VampOS-Noop*, *VampOS-DaS*, *VampOS-FSm*, and *VampOS-NETm*. In VampOS-Noop, all of the components are message-passing-based and scheduled by a round-robin scheduler. VampOS-DaS adds dependency-aware scheduling to VampOS-Noop. VampOS-FSm and VampOS-NETm merge file system and network components into a single component in VampOS-DaS, respectively. Specifically, VampOS-FSm merges VFS and 9PFS, while VampOS-NETm groups LWIP and NETDEV. The trials are run 100 times.

The results are shown in Fig. 5. The x-axis represents the system calls, and the y-axis is the average execution time. The bars represent the range of the standard deviation. The figure reveals that the performance penalty depends on the system call type. Compared with the vanilla Unikraft, the additional tasks in VampOS are the scheduling for the component thread, function logging for the component restoration, and message passing between component threads. The overhead of VampOS-Noop adds 21.0 microseconds in getpid() and almost no time in the other VampOS, since there are few component transitions and thus very few message passing communications. The other system call issues involve message communications, and VampOS's mechanisms incur additional overhead. The biggest relative overhead is getpid(), because its execution time is so short that the overhead becomes relatively high even if the VampOS's execution is shorter than other system calls like open() and read(). Although VampOS's execution takes longer in open() and read() than the other system calls, the overhead is relatively more minor (1.32× and 1.23×) as their executions are long.

The figure also shows that our optimizations, dependency-aware scheduling and component merges, effectively reduce the overhead. The dependency-aware scheduling mitigates the component scheduling overheads in all cases and becomes more effective as more component transitions occur. In socket_read() and socket_write(), VampOS-DaS is 0.34× and 0.49× shorter than VampOS-Noop. The component merges are also effective for merged components-related system calls. For example, VampOS-FSm in open() and close() is 0.91× and 0.91× shorter than VampOS-DaS. On the other hand, VampOS-NETm in socket_write() and socket_read() is 0.91× and 0.95× shorter than VampOS-DaS.

Table III shows the memory space overhead in each system call. VampOS stores ten and seven log entries in open() and close() since these system calls transit more than two stateful components, such as VFS and 9PFS, and change their states. The space overhead of the other system calls is up to two entries. This is because these system calls change the states of fewer components. The log shrinking feature effectively

TABLE III
LOG SPACE OVERHEADS IN SYSTEM CALLS.

System call	Normal Log Entries	Shrunk Log Entries
getpid()	0	0
open()	10	-1
read()	2	2
write()	2	2
close()	7	1
socket read()	2	0
socket write()	2	0

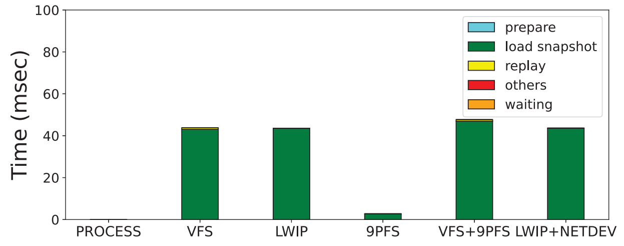


Fig. 6. **Component Reboot Times.** *The reboot time of PROCESS, a stateless component, is less than seven milliseconds. The snapshot loads are the dominant factor in rebooting stateful components, but the reboot times are less than 48 milliseconds.*

reduces these space overheads, 2 in close() and 0 in socket read()/write. -1 in open() means that this removes a pair of the previous open()/close() whose file descriptor number is the same. Without the corresponding pair, one log entry (open()) is added.

B. Component Reboot Time

We measure the reboot time for restarting components to demonstrate the lightweightness of VampOS-based reboots. This experiment involves six components used in Nginx: PROCESS, VFS, LWIP, 9PFS, VFS+9PFS(merged), and LWIP+NETDEV(merged). PROCESS is a stateless component implementing system calls for process management like getpid(). Here, we show the reboot time of PROCESS since the reboot of stateless components is the same. Also, we measure the reboot time for all of the stateful components used in our prototypes. VFS and 9PFS are file system components, while LWIP and NETDEV are ones for the network protocol stack. We measure the reboot time for each component after sending 1,000 GET requests to Nginx. These trials are done ten times.

Fig. 6 shows the reboot times for each component. The figure reveals that component reboot times are up to 48 msec. The reboot for PROCESS takes less than 7,428 nanoseconds since VampOS restarts it without replaying the logs and restoring a snapshot. The dominant factor in the reboot times for the stateful components is the snapshot restoration, and the reboot time depends on the memory footprint of the components. 9PFS is the fastest since the number of snapshots is small. Unlike other stateful components, there is no data or bss region in 9PFS; only the heap snapshot is loaded for the memory initialization of 9PFS. We conducted some experiments to check the reboot times with various log sizes. The log sizes do not matter because the snapshot restoration is dominant; its time is in on the order of ten milliseconds, while other overheads, like the log replay, are of the hundred-microsecond order.

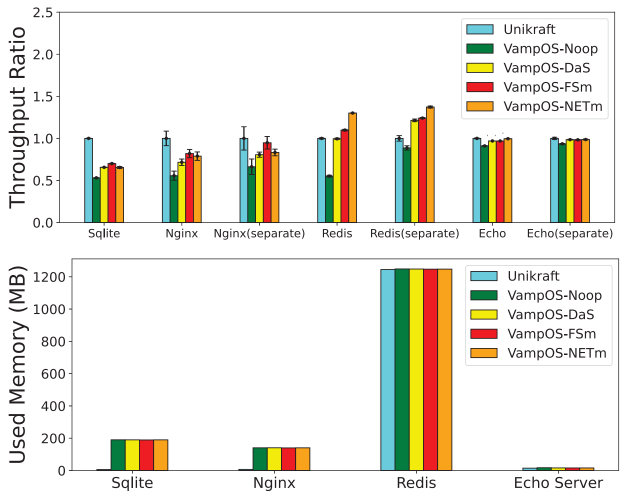


Fig. 7. **Real-world Application Overheads.** *VampOS's runtime overheads in real-world applications are up to 1.46×, while memory space overhead due to their mechanisms is less than 200 MB.*

C. Overheads of Real-world Applications

To confirm the runtime overheads of real-world applications, we measure the execution times and memory utilization of four real-world applications, SQLite, Nginx, Redis, and Echo, as described in §VI. Our workloads, whose configurations are based on [31] and [57], generate 1, 25, 4, and 1 threads. Moreover, they are run on the same machine to measure the overhead in the worst case, and on a separate machine connected via a gigabit Ethernet, whose spec is the same as another machine, for server applications. Each of them performs 10,000 inserts of a 1-byte data item, requests a 180-byte html file for a minute via 40 connections, sends 1,000,000 SET for a 4-byte key and 3-byte value, and sends a 159-byte html file for a minute to SQLite, Nginx, Redis, and Echo respectively. To make the unikernel layer rebootable, we turn on the AOF (Append Only File) backup feature in Unikraft-based Redis. It preserves volatile KVs into storage synchronously via fsync() to restore its KVs after its reboot. We run these applications on the four VampOS configurations described in §VII-A.

The results are shown in Fig. 7. Fig. 7(a) shows the execution time of each application and reveals that VampOS's performance penalty is up to 1.46×. The overhead of SQLite is 1.24× in VampOS-FSm, while Nginx's overhead is 1.24× in VampOS-FSm. We note that Nginx and its clients run on the same host so that the overhead can be amplified. In Nginx, the throughput of VampOS is comparable to that of Unikraft when they run on a separate machine. In Redis, VampOS's throughput outperforms that of Unikraft except for VampOS-Noop, since the overhead of AOF's synchronous storage accesses becomes larger than that of VampOS. To take a closer look at the behavior, we measure storage access time in their

TABLE IV
SQLITE THROUGHPUTS OVER LOG SHRINK THRESHOLD CHANGES.

Log shrink threshold	SQLite [req/sec]	Nginx [req/sec]	Redis [req/sec]
20	138.57	344.23	16894.75
100	143.57	344.00	16879.63
1000	144.28	344.83	16870.52

execution. The I/O time of Unikraft-based Redis in the same machine is 63.5% in the execution (17.2 sec.) due to the AoF feature, while that of VampOS's one is zero. The improvement of VampOS is more substantial in the same machine than in the separate machines since the storage accesses of AOF are more; due to network latency, one KV preservation contains fewer KVs. The I/O time in the separate machine is 63.0% in the execution time (21.7 sec.), which means the I/O time is longer than in the same machine. VampOS's throughput of Echo is comparable to that of Unikraft. In addition, our dependency-aware scheduling mitigates the performance penalty in all the cases.

Fig. 7b shows the memory utilization. The figure shows that VampOS's memory space overhead is less than 200 MB. VampOS consumes 200 MB due to the log's space overhead. The overhead is suppressed by the session-aware log shrinking that periodically removes the log entries. Since the memory footprints of SQLite and Nginx are 190 MB and 141MB, respectively, their memory space overheads are relatively big. On the other hand, the memory space overhead in Redis is small since its memory footprint (1,247 MB) is much larger than VampOS's constant memory consumption (200 MB). The space overhead of Echo is negligible since it closes connections after returning the messages and thus its log size does not increase.

Table IV lists the throughputs of the applications over log-shrink-threshold changes. The table indicates that the frequent log shrinking affects the application's performance. In the SQLite case, the throughput with a threshold of 1,000 is 1.04× better than that with a threshold of 20. The throughput with a threshold of 100 is slightly better than that of 1,000. Log shrinking does not affect Nginx and Redis significantly since the number of log entries does not often exceed the thresholds in these applications.

D. Case Study: Software Rejuvenation

To demonstrate how effective VampOS is in a software rejuvenation scenario, we rejuvenate a running Nginx with and without VampOS. We use the siege benchmark that spawns 100 threads, each sending GET requests. We measure the transaction success rate during the software rejuvenation. We use the VampOS-DaS configuration. To perform software rejuvenation that reboots the target software proactively against failures, we carry out VampOS-based reboots of each component one by one every 30 seconds. For comparison, we conduct a Unikraft-based reboot (full reboot).

Table V shows the request success rates for Unikraft- and VampOS-based software rejuvenation. The results demonstrate that VampOS rejuvenates the unikernel components without losing client requests. Since VampOS reboots only one component each time and restores the runtime state of a restarted stateful component like VFS, the connections are kept across the rejuvenations of all of the components. The benchmark reports that Unikraft-based rejuvenation loses 64 connections; it reboots all the running components, including the application and the unikernel components, losing their TCP connections. The results are independent of the type of rebooted component; each reboot time is small enough to significantly affect success/fail rates.

TABLE V
REQUEST SUCCESSES ACROSS UNIKRAFT- AND VAMPOS-BASED SOFTWARE REJUVENATION.

	Unikraft	VampOS
Success	191	255
Fails	64	0
Success Ratio	74.9 %	100 %

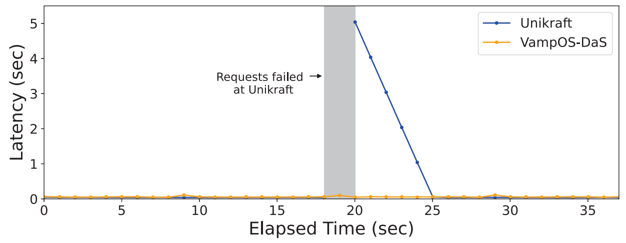


Fig. 8. **Redis Request Latency across Unikraft- and VampOS-based Failure Recovery.** *VampOS-based Redis recovers from the failure without almost zero performance penalty, while the full reboot fails to handle requests and significantly worsens request latencies.*

E. Case Study: Failure Recovery

To confirm the effectiveness of VampOS in a failure recovery scenario, we recover a running Redis from a failure by using VampOS. We prepare a warm-up Redis that has 1,000,000 key values, utilizes 1.2 GB of memory, and sends 1,000 GET requests, each of which randomly gets one value per second. We measure the response time by additionally sending one GET method per second. We intentionally inject a fail-stop failure into 9PFS. Specifically, we force 9PFS to call panic() and trigger its reboot. We use VampOS-DaS- and Unikraft-based reboots for the recovery.

Fig. 8 plots the request latency of Redis across Unikraft- and VampOS-based reboots. The figure reveals that VampOS recovers Redis from the failure with almost zero performance penalty. Since the Unikraft-based recovery restarts all the components of Redis and eliminates its KVs, a KV restoration using the AOF feature is required. The requests are not handled as a whole, and their latencies are degraded significantly until the restoration is completed. On the other hand, VampOS reboots only the faulty 9PFS component, restores its runtime states, and keeps Redis running across the reboot, which means the restoration is unnecessary. Redis, therefore, maintains its request latency during the VampOS-based recovery.

VIII. DISCUSSIONS AND LIMITATIONS

Handling the failures of unikernel-linked applications is a challenging problem for modern networked systems in data centers. To tackle this problem, VampOS is a first step to efficiently recover from the failures at the unikernel layer on the basis of reboot-based recovery; we show a design that allows us to perform reboot-based recovery at the component level. Below, we discuss its aspects and limitations.

Applicability of Other System Software: Componentization contributes to partial reboots like VampOS. In this work, we focus on unikernels, but modern system software layers such as hypervisors and OS kernels have a modular architecture, and thus, some of VampOS's mechanisms are applicable to them for fine-grained reboot-based recovery even with numerous stateful components.

Recovery of Unrebootable Components: VampOS cannot consistently restart stateful components that share the data with the underlying hypervisor. A unikernel-linked application that restarts such components is hard to run consistently and sometimes crashes since its states are inconsistent with the hypervisor's. For example, VIRTIO, which interacts with virtio devices exposed by the host Linux, shares the ring buffers with Linux. The restart of VIRTIO initializes the ring buffers, causing I/O requests to become lost in the operation and pointers to be misaligned to the ring buffers between VIRTIO and Linux. Moreover, we cannot reboot the component that manages a DMA region being processed by the hypervisor via I/O pass-through virtualization; the hypervisor fails to perform I/O since the reboot clears the DMA region. Reboot-based recovery of such components requires additional software mechanisms that orchestrate the unikernel components and hypervisor to build consistent states.

Graceful Termination with Unrecoverable Components: VampOS cannot recover from all the types of failures, as described in §II-B. The current prototypes terminate execution when the rebooted components face failure again. One way to move forward is to use undamaged components to restart the unikernel-linked application smoothly after a fail-stop. Some component failures do not affect execution directly. For example, Redis can handle client requests and store its KVs into storage when Sysinfo stops. In the Redis case, storing the current in-memory KVs in storage just before a fail-stop is more helpful for restoring the running state than eliminating all the KVs. Even when VampOS fails to recover from a component failure, partial recovery can still be achieved if the Redis layer and file system-related components are undamaged. Exploring such procedures to support the subsequent launch of applications is attractive to improve their reliability.

Handling Failures from Deterministic Bugs: When a component crashes due to a deterministic bug, which is not considered in our fault model, the VampOS recovery causes it to crash again since the restarted component triggers the bug by executing the same code path. The ability to handle such failures triggered by deterministic bugs improves the reliability of unikernel-linked applications. To do so, multi-versioning components can be helpful for deterministic bug mitigation [27], [56], [68], since their source code differs from that of the failed component. When a component fails, VampOS could insert a different version of the component, whose functionalities and interfaces are the same as in the failed one, thereby eliminating the execution of the buggy code path. Exploring the effectiveness of this approach is also an interesting topic.

Applying VampOS to Other Applications: In this work, we made unikernel components for four applications VampOS-aware and confirmed their effectiveness with prototypes. Although our prototypes cover the main OS components, including file systems (VFS and 9PFS), networks (IWIP and NETDEV), and core utilities (Process, Time, Platform, and so on), we need to prototype components used in other applications and integrate VampOS's mechanisms into them to show its applicability more clearly.

Reboots for Component Updates: VampOS focuses on an aspect of the reboot, i.e., recovery. Reboot is also used for software updates, an important administration task because software updates include ones that improve performance, add new functionalities, and repair security vulnerabilities. Software mechanisms to extend VampOS-based reboots for software updates lead to dynamic unikernel updates without interfering with the running application layer.

Support Tools for Making Unikernels VampOS-aware: We made components VampOS-aware manually; the VampOS mechanisms, such as message passing and component protection, were developed on Unikraft from scratch. Also, some of our optimizations are ad-hoc to a certain component. We believe that these mechanisms are independent of components and thus the developed components are reusable on applications. Some support tools to make components VampOS-aware and automatically suggest optimization schemes suitable to the target component are useful.

IX. Conclusion

Improving the reliability of unikernel is essential because it is a primary control layer that obtains computational resources from the hypervisor and allocates them to the linked application. Reboot-based recovery from failures at the unikernel layer involves service disruptions due to restarts of the whole unikernel-linked application. VampOS, presented in this paper, performs reboot-based recovery on only the failed components by forcing components to interact with each other in a message-passing manner, while keeping the other components and applications running. We prototyped VampOS in SQLite, Nginx, Redis, and Echo components. The experimental results show that the prototypes effectively perform reboot-based recovery of the target unikernel components in software rejuvenation and failure recovery scenarios. VampOS's code is available at https://zenodo.org/records/10784506.

Acknowledgments

We thank Hamed Okhravi (shepherd) and the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by JST, PRESTO Grant Number JPMJPR21P9, Japan.

REFERENCES

[1] CRIU: Checkpoint/Restore In Userspace.

[2] Syzkaller: an unsupervised, coverage-guided linux system call fuzzer, 2019.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 29–44, 2009.

[4] K. Bhat, D. Vogt, E. van der Kouwe, B. Gras, L. Sambuc, A. S. Tanenbaum, H. Bos, and C. Giuffrida. OSIRIS: Efficient and Consistent Recovery of Compartmentalized Operating Systems. In *Proc. of the 46th Annual IEEE/IFIP International Conference on Dependabile Systems and Networks (DSN '16)*, pages 25–36, 2016.

[5] K. Boos, N. Liyanage, R. Ijaz, and L. Zhong. Theseus: an experiment in operating system structure and state management. In *Proc of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, pages 1–19, 2020.

[6] A. Bovenzi, J. Alonso, H. Yamada, S. Russo, and K. S. Trivedi. Towards fast os rejuvenation: An experimental evaluation of fast os reboot techniques. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 61–70. IEEE, 2013.

[7] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *Proc. of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom '15)*, pages 250–257, 2015.

[8] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot–a technique for cheap recovery. In *USENIX Association OSDI '04: 6th Symposium on Operating Systems Design and Implementation*, pages 31–44, 2004.

[9] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proc. of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 73–88, 2001.

[10] Cloudius System. Issues cloudius-systems/osv, Accessed: 2023-08-01. https://github.com/cloudius-systems/osv/issues.

[11] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo. Software aging and rejuvenation: Where we are and where we are going. In *2011 IEEE Third International Workshop on Software Aging and Rejuvenation*, pages 1–6. IEEE, 2011.

[12] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo. A survey of software aging and rejuvenation studies. *J. Emerg. Technol. Comput. Syst.*, 10(1), jan 2014.

[13] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proc. of Twenty-First ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 351–366, 2007.

[14] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving Reliability through Operating System Structure. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 59–72, Dec. 2008.

[15] A. Depoutovitch and M. Stumm. Otherworld - Giving Applications a Change to Servive OS Kernel Crashes. In *Proc. of the 5th ACM European Conference on Computer Systems (EuroSys '10)*, pages 181–194, 2010.

[16] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *Proc. of 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 75–88, 2006.

[17] G. Gain, C. Soldani, F. Huici, and L. Mathy. Want more unikernels? inflate them! In *Proc. of the 13th Symposium on Cloud Computing (SoCC '22)*, pages 510–525, 2022.

[18] A. Goel, B. Chopra, C. Gerea, D. Mátáni, J. Metzler, F. Ul Haq, and J. Wiener. Fast Database Restarts at Facebook. In *Proc. of the 2014 ACM SIGMOD international conference on Management of data (SIGMOD '14)*, pages 541–549, 2014.

[19] S. Gravani, M. Hedayati, J. Criswell, and M. L. Scott. Fast intra-kernel isolation and security with iskios. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 119–134, 2021.

[20] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen. "epk: Scalable and efficient memory protection keys". In *Proc of the 2022 USENIX Annual Technical Conference (USENIX ATC '22)*, pages 609–624, 2022.

[21] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of the IEEE 25th Int'l Symp. on Fault-Tolerant Computing (FTCS '95)*, pages 381–390, 1995.

[22] Y. Jumonji and H. Yamada. Efficient software rejuvenation of in-memory key-value storages. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 280–285. IEEE, 2017.

[23] Y. Jumonji and H. Yamada. Efficent reboot-based recovery of in-memory databases. *IEICE Trans. on Information and Systems*, E104.D(12):2164–2172, 2021.

[24] A. Kantee and J. Cormack. Rump kernels no os? no problem! *USENIX; login: magazine*, 39(5):11–17, 2014.

[25] S. Kashyap, C. Min, B. Lee, T. Kim, and P. Emelyanov. Instant OS Updates via Userspace Checkpoint-and-Restart. In *Proc. of the 2016 USENIX Annual Technical Conference (ATC '16)*, pages 605–619, Jun. 2016.

[26] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. OSv—Optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC '14)*, pages 61–72, 2014.

[27] B. H. Koning Koen and G. Cristiano. Secure and Efficient Multi-Variant Execution Using Hardware-Assisted Process Virtualization. In *Proc. of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '16)*, pages 431–442, 2016.

[28] K. Kourai. Cachemind: Fast performance recovery using a virtual machine monitor. In *2010 International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 86–92. IEEE, 2010.

[29] K. Kourai and S. Chiba. A fast rejuvenation technique for server consolidation with virtual machines. In *37th annual IEEE/IFIP international conference on dependable systems and networks (DSN'07)*, pages 245–255. IEEE, 2007.

[30] K. Kourai and S. Chiba. Fast software rejuvenation of virtual machine monitors. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 8(6):839–851, 2010.

[31] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, c. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici. Unikraft: Fast, specialized unikernels the easy way. In *Proc. of the Sixteenth European Conference on Computer Systems (EuroSys '21)*, pages 376–394, 2021.

[32] S. Kuenzer, S. Santhanam, Y. Volchkov, F. Schmidt, F. Huici, J. Nider, M. Rapoport, and C. Lupu. Unleashing the power of unikernels with unikraft. In *Proc. of the 12th ACM International Conference on Systems and Storage (SYSTOR'19)*, page 195, 2019.

[33] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan. A linux in unikernel clothing. In *Proc. of the 15th ACM European Conference on Computer Systems (EuroSys '20)*, pages 1–15, 2020.

[34] M. Le and Y. Tamir. Rehype: Enabling vm survival across hypervisor failures. In *7th ACM Int'l Conf. on Virtual Execution Environments (VEE'11)*, pages 63–74, 2011.

[35] H. Lefeuvre, V.-A. Bădoiu, A. Jung, S. L. Teodorescu, S. Rauch, F. Huici, C. Raiciu, and P. Olivier. Flexos: Towards flexible os isolation. In *Proc. of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, pages 467–482, 2022.

[36] A. Lenharth, V. Adve, and S. T. King. Recovery domains: An organizing principle for recoverable operating systems. In *Proc. of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 49–60, 2009.

[37] G. Li, D. Du, and Y. Xia. Iso-unik: lightweight multi-process unikernel through memory protection keys. *Cybersecurity*, 3(1):11, 2020.

[38] W. Li, J. Gu, N. Liu, and B. Zang. Efficiently recovering stateful system components of multi-server microkernels. In *Proc. of IEEE 41st International Conference on Distributed Computing Systems (ICDCS '21)*, pages 494–505, 2021.

[39] C. Lupu, A. Albiundefinedoru, R. Nichita, D.-F. Blânzeanu, M. Pogonaru, R. Deaconescu, and C. Raiciu. Nephele: Extending virtualization environments for cloning unikernel-based vms. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 574–589, 2023.

[40] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie. Jitsu: Just-In-Time summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, pages 559–573, 2015.

[41] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 461–472, 2013.

[42] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the art of network function virtualization. In *Proc. of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, 2014.

[43] D. McKee, Y. Giannaris, C. O. Perez, H. Shrobe, M. Payer, H. Okhravi, and N. Burow. "preventing kernel hacks with hakc". In *Proc of the 2022 Network and Distributed Systems Security Symposium (NDSS '22)*, pages 1–17, 2022.

[44] A. K. M. F. Mehrab, R. Nikolaev, and B. Ravindran. Kite: Lightweight critical service domains. In *Proc. of the Seventeenth European Conference on Computer Systems (EuroSys'22)*, pages 384–401, 2022.

[45] MirageOS. Issues mirage/mirage, Accessed: 2023-08-01. https://github.com/mirage/mirage/issues.

[46] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev. "redleaf: Isolation and communication in a safe operating system". In *Proc of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, pages 21–39, 2020.

[47] R. Nikolaev, H. Nadeem, C. Stone, and B. Ravindran. Adelie: Continuous address space layout re-randomization for linux drivers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 483–498, 2022.

[48] R. Nikolaev, M. Sung, and B. Ravindran. Librettos: A dynamically adaptable multiserver-library os. In *Proc. of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*, pages 114–128, 2020.

[49] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran. A binary-compatible unikernel. In *Proc. of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'19)*, pages 59–73, 2019.

[50] P. Olivier, A. K. M. F. Mehrab, S. Lankes, M. L. Karaoui, R. Lyerly, and B. Ravindran. Hexo: Offloading hpc compute-intensive workloads on low-cost, low-power embedded systems. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 85–96, 2019.

[51] S. Österlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and C. Giuffrida. kMVX: Detecting Kernel Information Leaks with Multi-Variant Execution. In *Proc. of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, pages 559–572, 2019.

[52] S. Pailoor, A. Aday, and S. Jana. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *Pro.c of the 27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, 2018.

[53] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in linux: Ten years later. In *Proc. of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 305–318, 2011.

[54] J. Pan, G. Yan, and X. Fan. Digtool: A Virtualization-Based framework for detecting kernel vulnerabilities. In *Proc. of the 26th USENIX Security Symposium (USENIX Security 17)*, pages 149–165, 2017.

[55] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *Proc of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, pages 241–254, 2019.

[56] L. Pina, A. Andronidis, M. Hicks, and C. Cadar. MVEDSUA: Higher Availability Dynamic Software Updates via Multi-Version Execution. In *Proc. of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, pages 573–585, 2019.

[57] V. A. Sartakov, L. Vilanova, and P. Pietzuch. Cubicleos: A library os with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, pages 546–558, 2021.

[58] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-Assisted feedback fuzzing for OS kernels. In *Proc. of the 26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[59] 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. In *Proc. of the 8th USENIX Conference on File and Storage Technologies (FAST '10)*, pages 281–294, 2010.

[60] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. In *Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 1–16, 2004.

[61] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 207–222, 2003.

[62] S. M. S. Talebi, Z. Yao, A. A. Sani, Z. Qian, and D. Austin. Undo workarounds for kernel bugs. In *Proc. of the 30th USENIX Security Symposium (USENIX Security '21)*, pages 2381–2398, 2021.

[63] H. Tazaki, A. Moroo, Y. Kuga, and R. Nakamura. How to design a library os for practical containers? In *Proc. of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '21)*, pages 15–28, 2021.

[64] K. Terada and H. Yamada. Dwarf: Shortening Downtime of Reboot-based Kernel Updates. In *Proc. of the 12th European Dependable Computing Conference (EDCC '16)*, pages 208–217, Sep. 2016.

[65] Unikraft. Crash when running app-lua, Accessed: 2023-08-01. https://github.com/unikraft/unikraft/issues/841.

[66] Unikraft. Issues unikraft/unikraft, Accessed: 2023-08-01. https://github.com/unikraft/unikraft/issues.

[67] Unikraft. lib/ukallocbbuddy: fix a memory leak, Accessed: 2023-08-01. https://github.com/unikraft/unikraft/issues/689.

[68] J. Vinck, B. Abrath, B. Coppens, A. Voulimeneas, B. De Sutter, and S. Volckaert. Sharing is Caring: Secure and Efficient Shared Memory Support for MVEEs. In *Proc. of the Seventeenth European Conference on Computer Systems (EuroSys '22)*, pages 99–116, 2022.

[69] D. Williams, R. Koller, M. Lucina, and N. Prakash. Unikernels as processes. In *Proc. of the ACM Symposium on Cloud Computing (SoCC'18)*, pages 199–211, 2018.

[70] H. Yamada and K. Kono. Traveling forward in time to newer operating systems using shadowreboot. In *Proc. of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '13)*, pages 121–130, 2013.

[71] K. Yamakita, H. Yamada, and K. Kono. Phase-based Reboot: Reusing Operating System Execution Phases for Cheap Reboot-based Recovery. In *Proc. of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11)*, pages 169–180, 2011.

[72] Z. Yuan, S. Hong, R. Chang, Y. Zhou, W. Shen, and K. Ren. Vdom: Fast and unlimited virtual domains on multiple architectures. In *Proc. of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)*, pages 905–919, 2023.

[73] G. Zellweger, S. Gerber, K. Kourtis, and T. Roscoe. Decoupling Cores, Kernels, and Operating Systems. In *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 17–31, 2014.

[74] Y. Zhang, J. Crowcroft, D. Li, C. Zhang, H. Li, Y. Wang, K. Yu, Y. Xiong, and G. Chen. KylinX: A dynamic library operating system for simplified and efficient cloud virtualization. In *Proc. of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pages 173–186, 2018.