information

MDPI

Article

A Record–Replay-Based State Recovery Approach for Variants in an MVX System

Xu Zhong [1], Xinjian Zhao [2], Bo Zhang [3], June Li [1,*], Yifan Wang [1] and Yu Li [4]

[1] Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China; zhongx@whu.edu.cn (X.Z.); wangyif@whu.edu.cn (Y.W.)
[2] State Grid Jiangsu Electric Power Co., Ltd., Information & Telecommunication Branch, Nanjing 210024, China; zhaoxj1@js.sgcc.com.cn
[3] State Grid Laboratory of Power Cyber-Security Protection and Monitoring Technology, China Electric Power Research Institute Co., Ltd., Nanjing 210003, China; zhangbo6@epri.sgcc.com.cn
[4] Purple Mountain Laboratories, Nanjing 211111, China; liyu@pmlabs.com.cn
* Correspondence: jeli@whu.edu.cn

Abstract

Multi-variant execution (MVX) is an active defense technique that can detect unknown attacks by comparing the outputs of redundant program variants. Despite notable progress in MVX techniques in recent years, current approaches for recovery of abnormal variants still face fundamental challenges, including state inconsistency, low recovery efficiency, and service disruption of an MVX system. Therefore, a record–replay-based state recovery approach for variants in MVX systems is proposed in this paper. First, a Syscall Coordinator (SSC), composed of a recording module, a classification module, and a replay module, is designed to enable state recovery of variants. Then, a synchronization and voting algorithm is presented. When an anomaly is identified through voting, the abnormal variant is handed over to the SSC for state recovery, while the Synchronization Queue is updated accordingly. Furthermore, to ensure uninterrupted system service, we introduce a parallel grouped recovery mechanism, which enables the execution of normal variants and the recovery of abnormal variants to run in parallel. Experimental results on SPEC CPU 2006 benchmark and server applications show that the proposed approach achieves low overhead in both the recording and replay phases while maintaining high state recovery accuracy and supports uninterrupted system service.

Keywords: multi-variant execution (MVX); variant state recovery; record–replay; uninterrupted service

check for updates

Academic Editor: Leandros Maglaras

Received: 9 August 2025
Revised: 15 September 2025
Accepted: 22 September 2025
Published: 24 September 2025

Citation: Zhong, X.; Zhao, X.; Zhang, B.; Li, J.; Wang, Y.; Li, Y. A Record–Replay-Based State Recovery Approach for Variants in an MVX System. *Information* **2025**, *16*, 826. https://doi.org/10.3390/info16100826

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Software homogenization is a primary cause of widespread vulnerability propagation [1,2]. To address this issue, multi-variant execution (MVX) technology has been increasingly adopted in the field of cybersecurity. MVX leverages the redundant execution of multiple heterogeneous variants to mitigate the security risks posed by software homogenization, while effectively detecting attacks exploiting zero-day vulnerabilities [3].

However, the implementation of existing approaches in MVX systems still encounters several challenges. A particularly difficult problem is recovering the state of successor variants after compromised ones are removed [4]. Our investigation reveals that existing solutions generally lack effective mechanisms for recovering the state of variants after an MVX system has been attacked.

Research on software fault recovery can serve as a reference for variant state recovery in MVX systems. However, directly applying traditional fault recovery mechanisms to MVX systems presents several limitations. First, the inherent heterogeneity among variants may lead to inconsistencies between pre- and post-recovery states. Second, the requirement for redundant execution of multiple variants in MVX systems can incur significant overhead during state recording and recovery. Finally, when a successor variant undergoes recovery, normal variants often need to wait for the recovery to complete, resulting in potential service disruption.

The challenge of variant state recovery is particularly acute in high-availability and mission-critical systems where service interruptions can lead to significant financial loss or catastrophic failures. For instance, in industrial control systems managing critical infrastructure [5,6] like power grids, in real-time financial trading platforms, or in high-traffic web services, even a brief downtime for recovery is unacceptable. Traditional MVX systems, which often require halting all variants to handle a single failure, are ill-suited for these environments. Therefore, a critical need exists for a recovery mechanism that can restore failed variants without disrupting service continuity.

To overcome this challenge, we propose a record–replay-based state recovery approach for variants in MVX systems. To support variant recovery, we design and implement a Syscall Coordinator (SSC) into the MVX system. During the synchronized execution phase, the SSC records system calls that are validated through voting among the variants. In the state recovery phase, the SSC applies appropriate replay strategies based on the categories of the system calls and deterministically replays them using the recorded data. Experimental results show that the proposed approach enables efficient recovery of successor variants with low overhead, while maintaining uninterrupted system execution.

The main contributions are as follows:

- We design an SSC for an MVX system that supports state recovery of variants through a record–replay mechanism. The SSC consists of a recording module, a classification module, and a replay module. These components are responsible for recording system calls that are validated through a voting process during the synchronized execution phase, and for their deterministic replay during the state recovery phase.
- We design a synchronization and voting algorithm for dynamically managing the synchronized execution queue during the synchronized execution phase. When the voting results are consistent, the algorithm triggers the SSC's recording module to record system calls. In the case of divergence, the abnormal variant is removed from the Synchronization Queue and terminated. A new successor is then instantiated and assigned to the SSC for state recovery.
- We design a parallel grouped recovery mechanism to enable uninterrupted system service. By decoupling the responsibilities of the Monitor and the SSC, this mechanism allows the parallel execution of normal variants and the state recovery for successor variants. As a result, the MVX system is able to continue providing services without waiting for the recovery to complete.

The remainder of this paper is structured as follows: Section 2 reviews related work in multi-variant execution and software recovery technologies. Section 3 details our proposed methodology, including the system architecture and the design of its core components. Section 4 presents the performance, effectiveness, and security evaluation of our system. Finally, Section 5 concludes the paper and discusses future work.

2. Related Work

2.1. Multi-Variant Execution (MVX) Technology

Software vulnerabilities stem from abnormal state transitions caused by design or runtime errors [7], which can be exploited by attackers to gain unauthorized access. Traditional defense mechanisms primarily rely on passive detection based on known vulnerability signatures, resulting in delayed responses and limited effectiveness against newly emerging threats. This drives the rise of MVX techniques, which are characterized by dynamism, randomness, and heterogeneity in program variants that perform the same function [8]. MVX is an active defense approach that overcomes the shortcomings of traditional defense techniques.

A general architecture of MVX systems is illustrated in Figure 1 [9]. It typically consists of four components: a set of variants, an input proxy, an output voter, and a feedback controller.

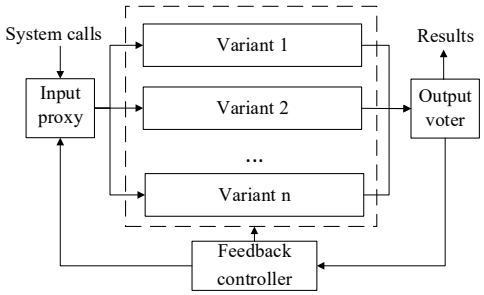


Figure 1. General architecture of multi-variant execution system.

- The variant set consists of multiple variants with the same function but distinct structures. These variants are generated using randomization techniques such as address space layout randomization (ASLR) [10]. Differences in the memory layouts of variants provide the heterogeneity that forms the basis of MVX systems' security guarantees.
- The input proxy replicates incoming inputs and distributes them to the variant set. Each variant in the set executes independently and produces its own output.
- The output voter collects the outputs from all variants, performs voting on the results, and returns the correct result to the runtime environment.
- The feedback controller handles voting discrepancies. Upon identifying voting discrepancies, the feedback controller reconstructs the variant set using a dynamic scheduling algorithm. This process is called the sanitization and recovery of abnormal variants. After an abnormal variant is removed, a successor variant is instantiated. Before this new variant can take over the execution, its state must be restored to match that of the active variants.

Current studies on MVX mainly fall into two domains: system architecture and core techniques [11–13]. This paper focuses on the former and reviews key related works.

Several systems have explored different architectures for MVX. For instance, Orchestra [14] and ReMon [15] pioneered the use of MVX for intrusion detection and secure application monitoring. However, their primary focus is on detecting divergence, and they typically respond by terminating the application, lacking mechanisms for state recovery and continued service. Similarly, MVEE [16] extends MVX to handle parallel programs but also does not address the challenge of recovering a variant after a fault.

Other systems offer partial or limited recovery capabilities. VARAN [17] adopts an event-stream model with a fixed-size ring buffer within a leader–follower architecture. While it can recover follower variants by replaying the leader's event stream, the leader itself cannot be recovered. This design limits its support for full state recovery. MvArmor [18]

utilizes hardware-assisted virtualization to improve MVX performance but, like others, lacks a dedicated recovery mechanism. sMVX [19] focuses on optimizing MVX by applying it only to selected sensitive code paths, but state recovery is not considered in its design.

Recent approaches have also integrated MVX with other defense techniques. MVX-CFI [20] integrates MVX with Control-Flow Integrity (CFI) for proactive defense but lacks feedback handling, terminating execution upon detecting attacks without recovery. Mimic-Box [21] introduces a mimic execution model for control-flow protection but cannot recover from attacks due to the absence of a fault recovery mechanism. Jmvx [22] employs a dual-mode architecture supporting both MVX and record–replay. However, these two modes are designed to be mutually exclusive; the system cannot perform state recovery via replay while operating in MVX mode, leaving a gap for in-service recovery.

In summary, existing work on MVX systems largely overlooks the problem of in-service state recovery for failed variants, highlighting a clear need for research in this area.

2.2. Software Recovery Technology

Software fault recovery has been the subject of extensive research, which has provided valuable insights for this work. The two main approaches are rollback recovery and record–replay.

Rollback recovery relies on checkpointing to periodically save system state (e.g., memory, registers, and process context), allowing the system to quickly roll back to the previous state upon failure. Tools like SCR [23], DMTCP [24], and CRIU [25] offer efficient and scalable solutions for rollback recovery. However, they are ill-suited for MVX systems, where the heterogeneity of variants leads to differences in their memory layout and control flow, making it impossible to directly use checkpoints for state recovery of a new variant.

Record–replay (RR) systems log user inputs, system events, and state information during program execution and deterministically replay them during recovery. Castor [26] is a system for recording and replaying multi-core applications. It achieves low logging overhead through hardware-optimized logging techniques, but it only runs on FreeBSD and requires source code recompilation, which limits its applicability. Scribe [27] introduces rendezvous points and synchronization points to handle interactions, achieving low-overhead replay with user transparency while allowing continued execution after replay. Tsan11rec [28] adopts a "sparse" logging strategy, only capturing the selected events. This improves runtime performance at the cost of reduced fidelity, which may lead to replay failures. Since Tsan11rec was primarily designed for debugging concurrency bugs, it may not guarantee accurate recovery.

Although these RR systems can address the heterogeneity of execution environments, they suffer from three critical limitations when applied to MVX systems:

- Scalability Overhead: Directly integrating RR into MVX significantly increases recording overhead due to multiple concurrent variants.
- State Inconsistency: Without state consistency guarantees, the output of the recovered variants may differ from that of the normal variants, potentially resulting in false positives.
- Service Disruption: The MVX system must suspend the operation of normal variants during variant recovery, leading to service interruptions until recovery completion.

3. Methodology

As discussed above, existing MVX works lack approaches for recovering the states of successor variants, and directly applying traditional fault recovery approaches to MVX systems has inherent limitations. Therefore, we propose a novel system architecture based on the record–replay mechanism to support state recovery for the successor variants in MVX systems.

3.1. Overview

3.1.1. Architecture

The proposed MVX system architecture is shown in Figure 2. It consists of six key components: Monitor, Syscall Coordinator (SSC), Shared Buffer, Synchronization Queue, Recovery Groups, and Thread Scheduler.

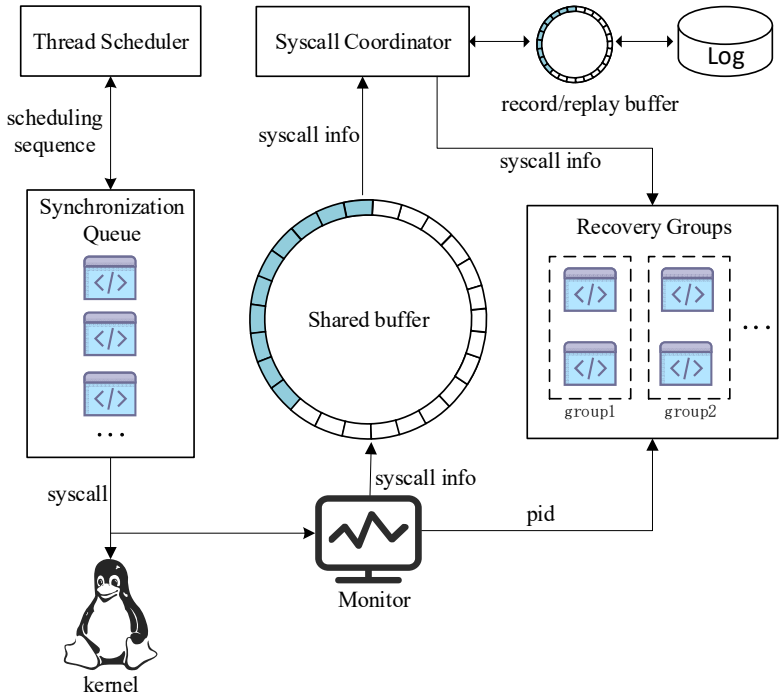


Figure 2. Overall system architecture.

- Monitor: This is the core component of the MVX system. It intercepts system calls issued by variants in the Synchronization Queue and controls both variant synchronization and majority voting.
- Syscall Coordinator (SSC): The SSC is responsible for the state recovery of new variants launched to replace abnormal ones. It records system calls that have been validated through voting and replays them during recovery (details in Section 3.2).
- Shared Buffer: This component temporarily stores system call records from the variants in the Synchronization Queue. The Monitor accesses these records in the Shared Buffer to perform synchronization and voting.
- Synchronization Queue: This queue holds the active variant set, which drives the program's forward execution. At each system call boundary, these variants must synchronize to perform voting. During the voting process, the abnormal variants are removed from this queue and transferred to the Recovery Groups (details in Section 3.3).
- Recovery Groups: They consist of newly instantiated variants, which we term "successor variants", that require state recovery. They are created to replace variants that fail in the voting and are grouped together to be recovered. Each group corresponds to a different recovery timestamp and is handled independently. The SSC coordinates the recovery of variants in each group, which runs concurrently with the forward execution of variants (details in Section 3.4).
- Thread Scheduler: This component supports multi-threaded execution across variants. During synchronized execution, it records the thread scheduling sequences of the leading variants and enforces the same sequences across all other variants to ensure consistency.

3.1.2. Workflow

As described in Section 3.1.1, we divide variants into two distinct sets. Accordingly, each variant runs in one of two phases: (1) synchronized execution and (2) state recovery. Figure 3 shows the workflow of the synchronized execution and recovery phases.

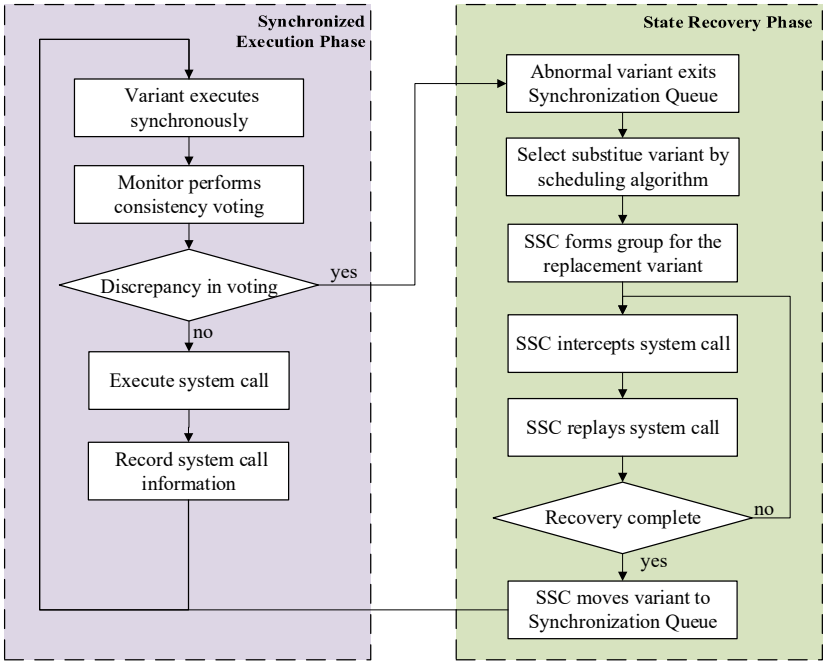


Figure 3. Workflow of the synchronized execution phase and the state recovery phase.

In the synchronized execution phase, all variants run under the supervision of the Monitor. The Monitor coordinates the synchronous execution of variants in this phase and performs consistency voting. When the voting reaches consensus, the variants proceed normally, and the SSC records the system calls of the validated variants. If a voting discrepancy is identified, the Monitor removes the inconsistent variant from the Synchronization Queue and suspends its execution. A successor variant is then selected according to the scheduling strategy and enters the state recovery phase, during which the SSC performs deterministic recovery based on recorded system calls.

The state recovery phase begins when a successor variant is launched following the removal of a divergent one. The SSC groups variants corresponding to the same voting failure point and initiates their state recovery. For each variant in this phase, the SSC intercepts system calls and deterministically replays them using previously recorded data. After each replaying of the system call, the SSC verifies whether the recovering variant's state aligns with that of the variants in the Synchronization Queue. Once consistency is confirmed, the variant is reintegrated into the Synchronization Queue and transitions back to the synchronized execution phase.

3.1.3. Parallel Execution Mechanism

Separating the variant execution process into two distinct phases is essential for maintaining uninterrupted system operation. This design enables the parallel execution of the forward progress of normal variants and recovery of successor variants.

Figure 4 shows the parallel execution mechanism between the synchronized execution phase and state recovery phases. In this architecture, variants in the Synchronization Queue process live service requests, coordinated by their respective Monitor threads. Meanwhile, successor variants are assigned to separate Recovery Groups (details in Section 3.4), where the SSC replays previously recorded system calls to restore their state.

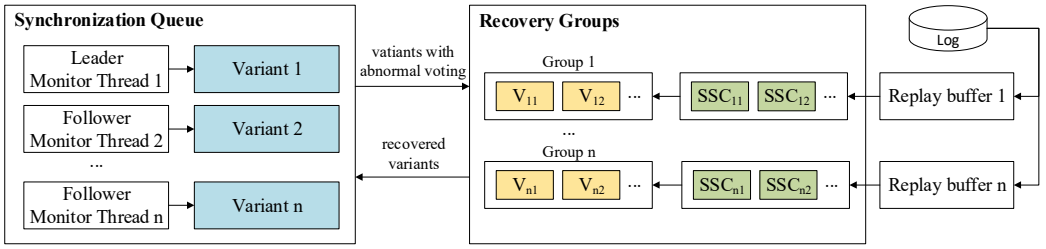


Figure 4. Parallel execution mechanism. The light blue boxes represent active variants in the Synchronization Queue. The yellow boxes denote successor variants undergoing state recovery within the Recovery Groups. The light green boxes are the Syscall Coordinator (SSC) instances responsible for managing the recovery process for each successor variant.

These two sets of variants operate concurrently in isolated control flows, allowing variants in different phases to run in parallel. This parallelism is enabled by the following three core decoupling principles:

Separation of Forward and Recovery Execution: The synchronized execution phase processes real-time system calls to drive application progress, while the recovery phase replays a fixed history of validated calls. Their independence in data flow and control logic eliminates mutual blocking.

Thread-Level Isolation: Each variant in the synchronized execution phase is monitored by a dedicated thread, reducing inter-variant execution dependencies. This design ensures that when a variant is identified as abnormal, it can be gracefully removed from the Synchronization Queue without disrupting the progress of other variants.

Dedicated Replay Buffer: Each recovery group maintains a dedicated replay buffer, allowing multiple recovery tasks to proceed in parallel without interfering with ongoing execution or one another.

3.2. Syscall Coordinator (SSC)

In this work, we design a Syscall Coordinator (SSC) to independently record and replay the four categories of system calls listed in Table 1**,** which are essential for reconstructing program state. The SSC is composed of three main modules: a recording module, a classification module, and a replay module, as shown in Figure 5.

Table 1. Classification of system calls.

Category	Examples
Non-deterministic system calls	getpid, time, random, fstat...
Sensitive system calls	mkdir, write, sendfile...
External input system calls	read, recv, accept...
Process-related system calls	fork, create...

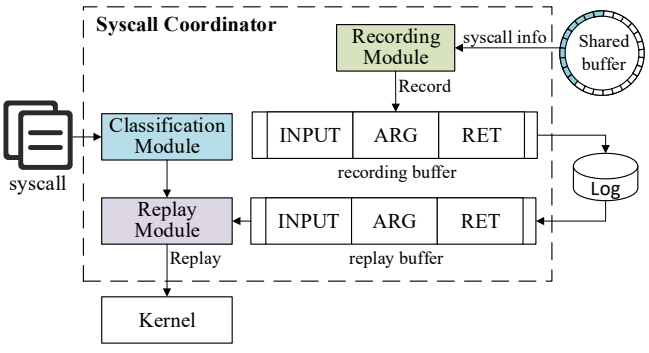


Figure 5. Internal structure of the SSC, composed of recording, classification, and replay modules.

1. Recording Module

The recording module persistently stores critical system call information after synchronization and voting. As shown in Figure 5, the module retrieves relevant data for each system call from a shared buffer and records it accordingly.

A record of a system call is composed of three fields: INPUT, ARG, and RET. The INPUT field stores input data streams from external devices. When an input-related system call is invoked, the SSC buffers the input in execution order to construct an accurate input event sequence. The ARG field stores the arguments of system calls that have passed majority voting. During the state recovery phase, these arguments are used to quickly verify the validity of replayed system calls. The RET field stores the initial return value of non-deterministic system calls. This value is used during recovery to override actual results and maintain consistency between pre- and post-recovery states.

To achieve efficient recording, we implement a two-phase logging mechanism based on the shared buffer: the record of the system call is first cached in a recording buffer. Once the buffer is full, a single I/O operation is used to flush the batch data into the log file. This mechanism converts fragmented high-frequency writes into sequential batch operations, reducing context-switching overhead from log I/O.

2. Classification Module

The classification module is a key component within the Syscall Coordinator (SSC), responsible for ensuring each system call is replayed deterministically and safely. Its primary function is to categorize system calls during the state recovery phase, a process that prevents state inconsistencies and unintended side effects.

As depicted in Figure 5, the classification module operates within the SSC and is invoked for each successor variant undergoing state recovery. This process is governed by the Parallel Grouped Recovery Algorithm (details in Section 3.4). For every system call record retrieved from the replay buffer, the classification module first receives the record containing the syscall number, original arguments (ARG), and return value (RET). It then uses the syscall number (e.g., __NR_read, __NR_mkdir) as a key to identify the call's category from the predefined types listed in Table 1. Finally, based on this category, the module selects the appropriate replay strategy and passes it to the replay module for execution.

As shown in Table 1, system calls are divided into four categories: non-deterministic system calls, sensitive system calls, external input system calls, and process-related system calls. The replay strategies for each category are as follows.

- Non-deterministic system calls (e.g., *getpid*, *random*, *fstat*) may return different values across variants and execution times. During recovery, such calls may lead to inconsistent states. The SSC handles them by skipping actual execution: it modifies the syscall number to an unrelated call (e.g., *getppid*) and replaces the return value with the one recorded in the ret unit.
- Sensitive system calls (e.g., *mkdir*, *write*, *send*) can affect external system states and are assumed to have already been executed correctly before recovery. Re-executing them during recovery could result in duplicated operations or errors. These calls are also skipped during recovery.
- External input system calls (e.g., *read*, *recv*) depend on sender-side input operations, which cannot be re-triggered during recovery. To handle this, the SSC caches the input during synchronous execution in the Input Unit. During recovery, the SSC intercepts the call, fetches the target memory address, writes the cached input content to the variant's memory, modifies the syscall number to a harmless one, and finally returns the input length as the syscall result.

- Process-related system calls, such as *fork*, pose a unique non-deterministic challenge because each variant receives a different process/thread ID (PID). Unlike other non-deterministic values (e.g., timestamps), these PIDs are crucial for subsequent control operations like *wait*(PID). Consequently, a naive approach of simply rewriting the PIDs to a common value to pass voting would break program logic, as later operations would reference an invalid PID.

To address this, we introduce virtual process identifiers (VPIDs), as shown in Figure 6. When a variant executes a *fork*, the SSC intercepts the real PID returned by the OS. It then maps this real PID to a canonical VPID that is consistent for this event across all variants, storing this mapping. During voting, the Monitor uses the mapping to translate each variant's real PID back to the common VPID, thus avoiding a false positive without altering the real PID used by the variant. When replaying this call, the SSC receives a new real PID (PID_new) from the OS. It simply updates the mapping to associate the original VPID with this PID_new.

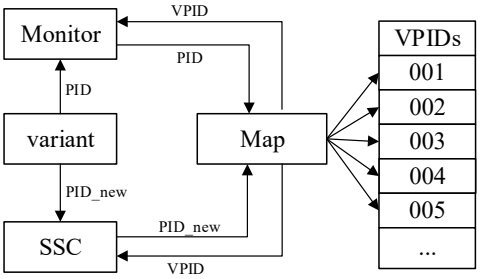


Figure 6. The virtual process identifier mechanism.

For other system calls not included in the above categories, such as *uname* and *getuid*, which do not modify the system state and return fixed values, no special handling is required during recovery, and they are executed normally.

3. Replay Module

The replay module is responsible for executing deterministic replays of system calls. As illustrated in Figure 5, the classification module determines the appropriate strategy based on system call type and passes it to the replay module. The replay module then reads the relevant system call data from external log files in batches into the replay buffer. Based on the buffered data and the chosen replay strategy, the module replays each system call deterministically.

3.3. Synchronization and Voting

As described in Section 3.1.2, this work maintains a set of healthy variants in the Synchronization Queue, which enables continuous service even in the presence of failures. This section details the mechanism by which the Monitor performs synchronization and consistency voting among these variants. The Monitor intercepts system calls, aligns the execution points across variants, and conducts majority voting to detect behavioral discrepancies. Variants identified as abnormal are immediately removed from the queue for recovery.

To support concurrent execution, the Monitor is implemented as a multi-threaded component, where each variant is managed by a dedicated Monitor thread. These threads coordinate through a shared buffer to ensure consistent synchronization and voting across all active variants. The complete procedure is presented in Algorithm 1.

Algorithm 1 Synchronization and Voting Algorithm

Input: Synchronization Queue Q_{sync}, Variant Process ID *pid*, *shared_buffer*, Primary Monitor Process *leader*

Output: *result*

syn_count ← 0 // Initialize the sync counter
syscallInfo ← SyscallGet(*pid*) // Intercept and fetch the syscall data
syn_count ← *syn_count* + 1
while *syn_count* < GetQueueNumber(Q_{sync}) **do**
Wait for other variants to sync // Wait all variants are synchronized
end while
shared_buffer ← *syscallInfo* // Write to shared buffer for voting
voteRes ← Voting(*shared_buffer*)
if *voteRes* =−1 *then*
result ← *valid*
if getpid() = *leader* **then**
RecordSyscall(*syscallInfo*) // Only the leader calls record module
end if
else
result ← *invalid*
if getpid() = *leader* **then**
RemoveFromQueue(*voteRes*, Q_{sync})
Pid_new ← scheduling()
Recovery(*Pid_new*) *// Trigger recovery for the successor variant*
while GetQueueNumber(Q_{sync}) = 0 **do**
Waiting for recovery to complete
end while
end if
end if

The *leader* refers to the ID of the primary Monitor thread. By designing a primary Monitor, we avoid redundant execution of related operations. *syn_count* is a shared variable initialized to 0 at the beginning of each synchronization round. It is used to track the number of variants that have completed synchronization in the current round. Monitor threads of different variants use this variable to collaboratively synchronize the variants.

Synchronization is considered complete when *syn_count* equals the number of variants in the Synchronization Queue (Q_{sync}). Monitor threads intercept system calls and read the relevant data via SyscallGet(*pid*). After synchronization, the data is written into the shared buffer for consistency voting. Voting(*shared_buffer*) performs voting based on the content in the *shared_buffer*. When the voting results are consistent, it returns -1; otherwise, it returns the process ID of the abnormal variant. If the voting result is valid, the primary Monitor calls the recording module in the SSC to record the system call data. Otherwise, the primary Monitor removes the abnormal variant from the Q_{sync}. A new successor variant is instantiated by scheduling(). This variant then transitions into the state recovery phase.

When the number of variants in Q_{sync} drops to zero, it indicates that all variants have been removed due to voting failures. In this case, the system is temporarily suspended until one or more recovering variants complete their recovery and rejoin the Synchronization Queue.

3.4. State Recovery for Variants

This work proposes a parallel grouped recovery mechanism for the state recovery phase, enabling successor variants to be restored in isolation. As illustrated in Figure 4, successor variants are divided into independent Recovery Groups, each of which is assigned a dedicated replay buffer. Variants within a recovery group are managed by their corresponding Syscall Coordinator (SSC) instances. For example, in Group 1, the variant labeled V_{11} is paired with its respective coordinator SSC_{11}, which is responsible for its recovery process. Recovery Groups operate in parallel with the Synchronization Queue, enabling non-disruptive fault handling without interfering with the execution of healthy variants.

This grouping strategy is specifically designed to avoid resource contention that may arise when multiple variants enter the state recovery phase at different times. Upon detection of inconsistency, the Monitor terminates the abnormal variant, after which a newly instantiated successor is assigned to the SSC for state recovery. Instead of using a single recovery pipeline, the SSC assigns these variants from different synchronization points to separate Recovery Groups for independent processing. Each group is allocated a dedicated replay buffer, ensuring that recovery operations are isolated and free of mutual interference. Once all variants in a group have completed recovery, the associated buffer is released to conserve system resources.

The core recovery logic within each group involves deterministically replaying previously recorded system calls until the recovered variant reaches the state of the Synchronization Queue. This process is orchestrated by the classification and replay modules of the SSC and is detailed in Algorithm 2.

Algorithm 2 Parallel Grouped Recovery Algorithm

Input: Process ID of successor variant *pid*, Synchronization Queue Q_{sync}

Output: Number of replayed system calls *count*

```
count ← 0                                  // Initialize replayed system call counter
G_i ← NewRecoveryGroup(pid)
buffer ← InitReplayBuffer(G_i)     // Initialize the replay buffer
while count < GetState(Q_sync) do  // Check the recovery is complete
UpdateBuffer(buffer)               // Update buffer contents if needed
syscallInfo ← ReadCall(count,buffer)
replayStrategy ← Classify(syscallInfo)
ReplayCall(pid, syscallInfo, replayStrategy)
count ← count + 1
end while
AddToQueue(Q_sync,pid)       //Add variant back to the queue
ReleaseGroup(G_i)
return count
```

First, the SSC updates the buffer via UpdateBuffer(*buffer*). If all contents in the buffer have been accessed, new system call records are batch-loaded from the log into the buffer. ReadCall(*count*, *buffer*) retrieves the relevant system call information from the buffer, and the classification module determines the *replayStrategy* based on the system call number in *syscallInfo*. The replay module then replays the system call according to the obtained information and strategy. Finally, the SSC checks whether the current variant's state matches that of the variants in the Synchronization Queue Q_{sync}. This check is implemented by comparing the number of replayed system calls with the number of executed system calls in the queue GetState(Q_{sync}). If the states match, the variant is considered successfully

recovered and is re-added to the queue, transitioning back to the synchronized execution phase. Detailed design and handling of system call categories used during replay are described in Section 3.3.

3.5. Comparison with Representative MVX Systems

To highlight the novelty and effectiveness of our approach, we compare it with several representative MVX systems in terms of their ability to recover variant states and support for parallelism during recovery. Table 2 summarizes these comparisons, followed by further discussion of the key differences.

Table 2. Comparison with representative MVX systems.

Approach	Recovery Support	Parallelism During Recovery	Notes
Ours	Yes	Yes	Enables Parallel Recovery Without Halting Normal Execution
Orchestra [14]	Full restart only	No	Stop running when an anomaly is detected and restart the variants
ReMon [15]	No	No	Implements record–replay for multi-thread parallelism
MVEE [16]	No	No	Similar to ReMon
Varan [17]	Partial	No	Restricts recovery only to slave variants via leader–follower
sMVX [19]	No	No	Recovery is not considered

Variant state recovery is lacking in existing systems but is provided in ours. Most existing MVX systems, such as Orchestra, sMVX, and MVEE, respond to anomalies by directly terminating or restarting the affected variant, without attempting any form of state recovery. This approach discards the variant's progress and offers no mechanism for state recovery. In contrast, we introduce a Syscall Coordinator (SSC) that records validated system calls during synchronized execution. When a new variant is launched, the SSC deterministically replays these logs to reconstruct the variant's state.

Support for variant recovery by record–replay mechanisms is absent in existing systems but is present in ours. Some MVX systems, such as ReMon and MVEE, incorporate record–replay mechanisms, but they are designed only to enforce deterministic execution of multi-threaded programs, ensuring that variants follow the same thread interleaving during synchronized execution. Once a variant diverges or fails, it cannot be restored or reintegrated into the system.

Varan implements record–replay functionality through an event-stream mechanism combined with a leader–follower architecture. In this design, only the leader variant executes independently and generates a stream of events, which follower variants replay to maintain consistency. This architecture enables state recovery for follower variants by reissuing the recorded system calls from the leader. However, due to the inherent constraints of the leader–follower model, only the leader's execution can be recorded, and only follower variants can be restored. The leader lacks a replay mechanism and cannot be recovered once it fails. Furthermore, Varan's components are tightly coupled: when a follower enters recovery, the leader must pause its execution and wait for the recovery process to complete. As a result, the system cannot support uninterrupted execution during variant failures.

In our system, state recovery is handled independently by the Syscall Coordinator (SSC), separate from the execution flow of Synchronization Queue. Failed variants are assigned to independent Recovery Groups, and recovery is performed in parallel with ongoing execution. This decoupled design allows failed variants to undergo recovery without interfering with ongoing execution.

4. Evaluation

To evaluate the proposed approach, we implemented an MVX system based on the approach described above. Our experiments were conducted on a server equipped with an Intel Xeon Platinum 8255C processor and 32 GB of memory, running Ubuntu 20.04 LTS. The system was implemented in C17 and compiled using GCC 9.4.0.

Our system uses ptrace to intercept system calls in both the Monitor and the SSC. Common MVX interception methods include loaded kernel modules [29], binary rewriting [17], and ptrace API [30]. While kernel modules and binary rewriting can reduce context-switching overhead, they introduce deployment and security challenges. In particular, binary rewriting is more susceptible to evasion. In contrast, ptrace, as a native UNIX interface, offers better security and deployment flexibility, making it the most widely adopted approach.

This section will evaluate the proposed system in terms of its performance, effectiveness, and security.

4.1. Performance

In this work, we introduce a record–replay mechanism into MVX systems to enable variant state recovery. Therefore, this section evaluates the performance of the system's recording and recovery processes using the SPEC CPU 2006 benchmark and server applications.

1. Performance on Microbenchmark

The SPEC CPU 2006 is an industry-standardized benchmark suite designed to measure the compute-intensive performance of a system. It consists of a set of real-world programs that are computationally demanding, making it suitable for evaluating the fundamental overhead of our system. To assess our system's performance on the SPEC CPU 2006 benchmark suite, we measured four metrics for each program: native execution time (no MVX) T_{native}, MVX execution time without recording T_{mvx}, MVX execution time with recording enabled T_{record}, and time spent replaying system calls during recovery $T_{recovery}$. All experiments used a redundancy degree of three variants. The results are plotted in Figure 7. The y-axis represents the normalized execution time, where the performance of the original, unmodified application (Native) is the baseline at 100%. Therefore, any value above 100% indicates the relative increase in execution time. For instance, a Record value of 150% signifies that the execution under our system took 1.25 times longer than the native execution, corresponding to a 25% time overhead.

We compare our system against two representative record–replay systems, Scribe and Jmvx. Since neither Scribe nor Jmvx performs synchronization and voting, we normalize our recording overhead C_{record} to T_{mvx} (Equation (1)). Since variants in our system do not require synchronization or coordination during the recovery phase, the recovery overhead $C_{recovery}$ is normalized to the native baseline T_{native} (Equation (2)). As shown in Figure 7, our system incurs an average recording overhead of 6.96% and a replay overhead of 8.98%.

$$C_{record} = (T_{record} - T_{MVX})/T_{MVX} \times 100\% \quad (1)$$

$$C_{recovery} = \left(T_{recovery} - T_{native}\right)/T_{native} \times 100\% \quad (2)$$

Jmvx integrates multi-variant execution and record–replay into a dual-mode system. However, its architecture lacks proper logical isolation between the two modes, preventing coordinated execution and making it incapable of recovering successor variants within a multi-variant execution system. As a result, only Jmvx's record–replay mode is included in the comparison.

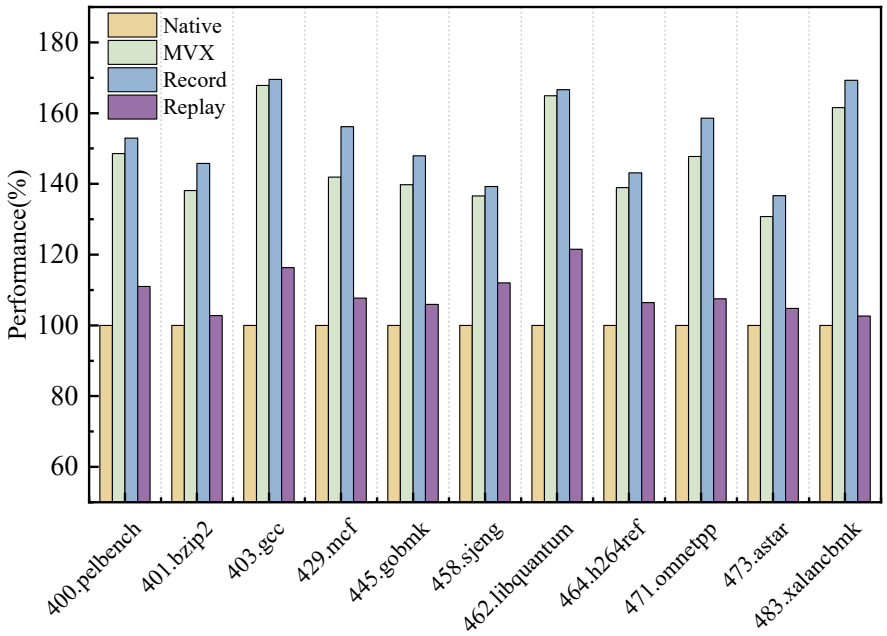


Figure 7. Performance overhead on SPEC CPU 2006.

According to the results in Table 3, Scribe demonstrates the lowest additional overhead, but it relies on modifications to the Linux kernel, limiting its deployment flexibility. In contrast, our system is entirely implemented in user space. Although it introduces slightly higher overhead (6.96% vs. 5%), it offers significant advantages in terms of portability and deployment convenience. Compared to Jmvx, our approach achieves lower overheads in both the recording and replay. This performance improvement is attributed to three key design optimizations: First, during recording, the SSC directly retrieves information from a ring-shaped shared buffer, avoiding interference with the Monitor's synchronization and voting operations. Second, a batch-based log management strategy in the shared buffer reduces the frequency of I/O operations and minimizes read/write overhead. Finally, in the state recovery phase, the replay strategy intelligently skips system calls that do not require re-execution, further reducing the cost of recovery.

Table 3. Comparison of performance overhead on SPEC CPU 2006.

	Performance Overhead	
	Recording	**Recovery**
Ours	6.96%	8.98%
Scribe [27]	5%	/
Jmvx [22]	8%	13%

2. Performance on Server Application

To further evaluate the practicality of our system, we tested its performance under several widely used server applications, including *lighttpd*, *nginx*, and *redis*. These applications represent typical long-running, high-throughput services that are sensitive to performance overhead and recovery latency.

To better simulate real-world network conditions, the server applications were deployed on a cloud node located in Shanghai, China, while client requests originated from a university in Central China. This setting introduced realistic WAN latency and network variability into the evaluation. Each service was tested in three modes: Native: native execution baseline, MVX (synchronization and recording): our system running with synchronization and syscall recording, and Recovery (replay): our system during the recovery phase, replaying system calls in a detached thread.

For *nginx* and *lighttpd*, we used ApacheBench (ab) with 10,000 total requests and 100 concurrent connections. For *redis*, we used redis-benchmark with 10,000 SET/GET operations and 10 concurrent connections.

Figure 8 presents the performance overhead of our system across three server applications. Unlike traditional record–replay systems, which operate on a single execution stream, our system performs recording within an MVX framework, where variants must synchronize and vote at system call boundaries. As a result, the overhead observed during the recording phase in our system reflects both the MVX synchronization cost and the recording overhead. It is therefore inappropriate to directly compare recording overhead between our system and classical record–replay systems, since the synchronization delay is unique to MVX. Instead, we evaluate the synchronization + recording performance of our system against representative MVX systems to assess whether our design introduces any additional performance burden during synchronized execution.

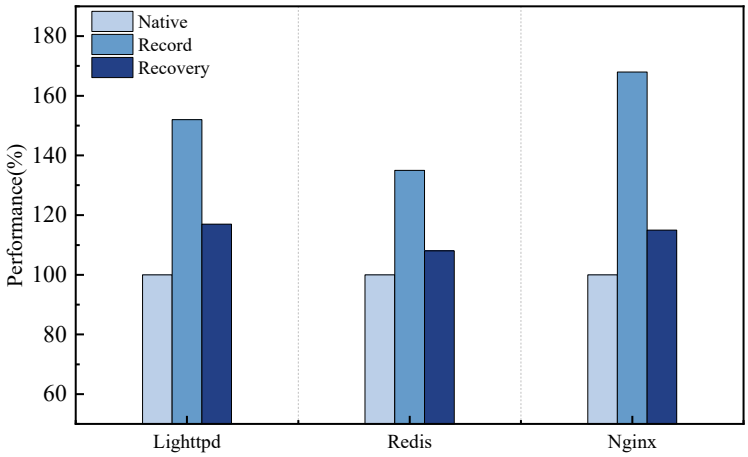


Figure 8. Performance overhead on server applications.

In contrast, the recovery phase in our system operates independently: successor variants undergoing state recovery do not participate in voting or synchronization. In this regard, our replay model is comparable to traditional record–replay systems, as it executes a single-threaded deterministic re-execution.

To further contextualize these results, we compare our system with representative MVX and record–replay frameworks in Table 4. The comparison shows that our system introduces performance overheads comparable to those of existing MVX and record–replay systems, and even demonstrates competitive results in some cases (e.g., 1.52× for lighttpd vs. 2.23× in sMVX, and 1.68× for nginx vs. 2.94× in ReMon). These results indicate that our approach successfully extends record–replay capabilities within MVX systems without incurring significant additional time overhead. Moreover, the entire design operates in user space, enhancing portability for real-world deployment.

Table 4. Comparison of performance overhead on server applications.

System	Record			Replay		
	Lighttpd	**Redis**	**Nginx**	**Lighttpd**	**Redis**	**Nginx**
Ours	1.52×	1.35×	1.68×	1.17×	1.08×	1.15×
MvArmor [18]	1.77×	/	1.47×	/	/	/
ReMon [15]	1.55×	1.45×	2.94×	/	/	/
Tachyon [31]	1.48×	/	/	/	/	/
sMVX [19]	2.23×	/	2.66×	/	/	/
Castor [26]	/	/	/	1.13×	/	1.09×
HyCoR [32]	/	/	/	1.05×	1.21×	/

4.2. Effectiveness

To evaluate the effectiveness of our proposed record–replay mechanism for variant state recovery, we focus on two aspects: (1) ensuring uninterrupted execution, verifying that the system continues running correctly and smoothly even when variants experience faults, and (2) verifying recovery correctness, measuring the consistency of system call re-execution compared to the original execution.

4.2.1. Uninterrupted Execution Verification

A key objective of our design is to allow normal variants to continue execution while successor variants undergo state recovery, thereby avoiding service interruption. To verify this property, we conducted controlled fault injection experiments on the SPEC CPU 2006 benchmark.

For each benchmark, we inject a voting failure at three execution checkpoints: 25%, 50%, and 75% of the total runtime. Upon detection, the affected variant is transferred to the SSC for recovery while the remaining variants continue execution. We define a three-level availability scale to assess uninterrupted execution: Level 0 indicates complete service interruption, Level 1 indicates that the system continues running but the successor variant fails to recover, and Level 2 indicates that the system operates normally and the successor variant is fully recovered and reintegrated into synchronous execution.

The results of our uninterrupted execution tests are shown in Table 5. The columns labeled “25%”, “50%”, and “75%” indicate the point during the program’s total execution time at which we simulated a voting failure to trigger the recovery process. The numerical values in the table correspond to the availability levels defined previously: Level 2 means the system continued service without interruption and the successor variant was fully recovered and reintegrated into the Synchronization Queue, while Level 1 means the system continued service, but the recovery process did not complete before the benchmark program terminated.

Table 5. Results of effectiveness testing on SPEC CPU 2006.

Program	Correctness	Uninterrupted Operation		
		25%	**50%**	**75%**
pelbench	√	2	2	1
bzip2	√	2	2	2
gcc	√	2	2	2
mcf	√	2	2	2
gobmk	√	2	2	2
sjeng	√	2	2	2

Table 5. *Cont.*

Program	Correctness	Uninterrupted Operation		
		25%	50%	75%
libquantum	√	2	2	2
h264ref	√	2	2	1
omnetpp	√	2	2	1
astar	√	2	2	2
xalancbmk	√	2	2	2

As shown in Table 5, our system achieves Level 2 availability in the vast majority of test cases, confirming its ability to handle variant failures seamlessly. The few instances of Level 1 outcomes occurred when a fault was injected late in the program's execution (at the 75% mark). In these cases, the benchmark program terminated before the recovery process, which restarts from the initial state. This is an expected artifact of testing with short-running benchmark programs and does not represent a failure of the recovery mechanism. For long-running server applications, this scenario is highly unlikely, as the recovery process would have ample time to complete. These results strongly validate our design goal of providing uninterrupted service during variant recovery.

4.2.2. Recovery Correctness Verification

We further evaluate the correctness of the recovery process by comparing the replayed system calls against the original system call log. Inconsistencies may occur due to non-determinism, external inputs, or timing-related behavior, which are inherent challenges in practical replay systems. Additionally, during the recovery process, a variant may execute certain system calls that are not recorded by the SSC, typically those that do not affect program state (e.g., *epoll_wait*, *poll*). These benign mismatches can lead to inconsistencies when compared with the recorded log, similar to false positives in multi-variant execution systems.

We define replay *Accuracy* (Equation(3))as the percentage of system calls whose arguments and return values match those recorded during the original execution.

$$Accuracy = \frac{correctly\ replayed\ syscalls}{Total\ syscalls} \tag{3}$$

All SPEC CPU benchmarks are deterministic and single-threaded, making them ideal for verifying functional correctness. For each benchmark, we compare the syscall arguments and return values after replay. As shown in Table 5, our system achieves 100% replay *Accuracy* across all programs, confirming that the SSC can deterministically restore variant states in controlled environments.

We further tested *nginx*, *lighttpd*, and *redis* under realistic workloads, executing 10,000 requests to each service, with one variant undergoing recovery. During replay, each system call is compared against the original log to detect divergence.

As shown in Figure 9, after an initial warm-up period, the replay *Accuracy* for all server applications stabilizes above 95%, corresponding to a replay inconsistency rate of less than 5%. The initial slight variations in *Accuracy* are attributed to the non-deterministic system calls common during server initialization and process setup. As the server transitions into a steady state, it begins to handle client requests using highly repetitive and predictable sequences of system calls. This operational consistency leads to the *Accuracy* curve plateauing, demonstrating the reliability of our replay mechanism for long-running services. We compare the replay inconsistency of our recovery mechanism with the false positive rates observed in existing multi-variant execution systems. Specifically, ReMon [15]

reports a false positive rate of 9.1%, and Mimic-box [21] reports 13.6%, whereas our system achieves a significantly lower rate of mismatch during recovery. These results validate the practicality of our approach in handling real-world workloads with high consistency.

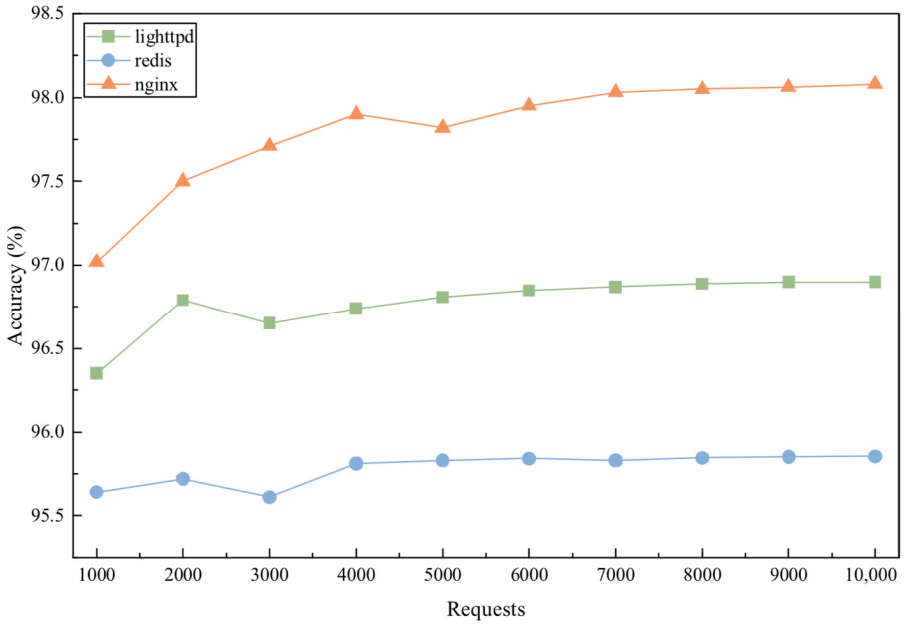


Figure 9. Recovery accuracy over time during 10,000 requests in server applications.

4.3. Security

MVX systems are primarily designed to defend against memory-corruption-based attacks, particularly those that rely on code reuse, control-flow hijacking, and memory layout exploitation. These include, but are not limited to, stack overflows, heap overflows, format string vulnerabilities, and use-after-free errors. Our proposed system builds on the same security foundations as prior MVX designs, such as N-Variant Systems [3], ReMon [15], and MvArmor [18], and adopts the same core defense principle: runtime behavioral divergence detection.

Although our system introduces a state recovery mechanism, it does not alter or weaken the core security model of MVX. All variants are still diversified and monitored in lockstep (in the Synchronization Queue), and any abnormal variant is promptly isolated and sanitized before being recovered. Therefore, the types of attacks our system can defend against remain consistent with those covered by prior MVX systems.

To validate the system's defense capabilities, we test it against representative CVE vulnerabilities, focusing on stack overflows, heap overflows, and integer overflows. These are among the most common memory corruption attacks. To simulate these attacks, we used publicly available proof-of-concept (PoC) exploits. For CVE-2013-2028 (Nginx stack overflow), we sent a specially crafted HTTP request with an oversized URL to trigger the buffer overflow during URL parsing. For CVE-2014-0160 (OpenSSL "Heartbleed"), we sent a malicious heartbeat request to the vulnerable server, causing it to read and return private memory contents. For CVE-2021-4790 (Apache HTTP Server), we sent a crafted request that leads to a null pointer dereference. In each case, the exploit attempts to alter the control flow or read unauthorized memory, leading to divergent behavior between variants. The CVE-2017-13089 vulnerability in Wget was triggered by having the client download a file from a malicious server that uses chunked transfer encoding to cause a stack overflow. The experiments use dual-variant redundancy to represent a worst-case deployment scenario. The results are shown in Table 6.

Table 6. Results of security testing.

CVE	Threat	Software Version	Defense Success
CVE-2013-2028	Stack overflow	Nginx 1.3.9	√
CVE-2014-0160	Heap overflow	OpenSSL 1.01	√
CVE-2021-4790	Stack overflow	Apache HTTP Server $\leq$ 2.4.51	√
CVE-2017-13089	Integer overflow	Wget < 1.19.2	√

We take CVE-2013-2028 as a representative example. This vulnerability is a stack overflow found in the Nginx server. By sending an HTTP request with an excessively long URL, an attacker can trigger a buffer overflow during URL parsing in Nginx, leading to stack memory corruption and leakage of the return address. This enables the attacker to construct an ROP chain to launch an attack against the server, causing the program to jump and execute malicious code embedded in the overflow data. In our system, due to address space layout randomization (ASLR) and variant diversification, the two variants jump to different addresses when an attack is launched. As a result, when the Monitor intercepts and votes on system calls, it identifies the inconsistency as illegal behavior, subsequently blocking the execution of the affected variant and triggering the sanitization and recovery mechanism. The experimental results confirm that the Monitor effectively detects and mitigates malicious actions resulting from these buffer overflow attacks.

In conclusion, the multi-variant execution system implemented with our proposed approach can not only support state recovery and uninterrupted service but also effectively defend against overflow-based attacks.

5. Conclusions

To address the issues of state inconsistency, low recovery efficiency, and service interruption in existing MVX systems, this paper proposes a record–replay-based state recovery approach for variants. The approach leverages a Syscall Coordinator (SSC) to record validated system calls during the synchronized execution phase and deterministically replay them for recovering the state of failed variants. The SSC is functionally decoupled from the Monitor, allowing it to record and replay system calls without interfering with the execution of normal variants. In addition, a parallel grouped recovery mechanism is introduced to isolate recovery tasks and avoid resource contention among concurrently recovering variants. The combination of this decoupled design and the parallel grouped recovery mechanism enables the system to maintain uninterrupted execution of normal variants during the recovery of successor variants.

We implement a system based on our approach and evaluate it in terms of performance, effectiveness, and security. Experimental results validate that our approach preserves the core security guarantees of the MVX architecture while achieving highly efficient state recovery for successor variants. It also maintains uninterrupted service during the state recovery phase. As a result, the proposed approach effectively addresses the limitations of existing methods, including state inconsistency, low recovery efficiency, and service disruption.

Future work will focus on enhancing the security of the Monitor, a critical component in any multi-variant execution system, to ensure its integrity during runtime, further improving the overall security of the system.

Author Contributions: Conceptualization, X.Z. (Xu Zhong), X.Z. (Xinjian Zhao), B.Z., J.L., Y.W., and Y.L.; methodology, X.Z. (Xu Zhong) and J.L.; software, X.Z. (Xu Zhong); validation, X.Z. (Xu Zhong) and Y.W.; investigation, X.Z. (Xu Zhong) and Y.L.; resources, B.Z. and X.Z. (Xu Zhong); writing—original draft preparation, X.Z. (Xu Zhong); writing—review and editing, X.Z. (Xu Zhong)

and J.L.; supervision, J.L.; project administration, J.L.; funding acquisition, X.Z. (Xinjian Zhao) and B.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by State Grid Corporation of China: Research on the Theory and Technology of Endogenous Security and Reliable Operation of Power Information System Against Unknown Attacks, grant number 5700202458225A-1-1-ZN.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Acknowledgments: We gratefully acknowledge the support of State Grid Jiangsu Electric Power Co., Ltd., Information & Telecommunication Branch and China Electric Power Research Institute Co., Ltd. for their constructive feedback during project discussions and for providing critical data for this study.

Conflicts of Interest: Author Xinjian Zhao is employed by State Grid Jiangsu Electric Power Co., Ltd. Author Bo Zhang is employed by China Electric Power Research Institute Co., Ltd. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

MVX	Multi-Variant Execution
SSC	Syscall Coordinator
RR	Record–Replay

References

1. Temizkan, O.; Park, S.; Saydam, C. Software diversity for improved network security: Optimal distribution of software-based shared vulnerabilities. *Inf. Syst. Res.* **2017**, *28*, 828–849. [CrossRef]
2. Albusays, K.; Bjorn, P.; Dabbish, L.; Ford, D.; Murphy-Hill, E.; Serebrenik, A.; Storey, M.-A. The diversity crisis in software development. *IEEE Softw.* **2021**, *38*, 19–25. [CrossRef]
3. Cox, B.; Evans, D.; Filipi, A.; Rowanhill, J.; Hu, W.; Davidson, J.; Knight, J.; Nguyen-Tuong, A.; Hiser, J. N-Variant Systems: A Secretless Framework for Security through Diversity. In Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, 31 July 2006; pp. 105–120.
4. Yao, D.; Zhang, Z.; Zhang, G.; Liu, H.; Pan, C.; Wu, J. A Survey on Multi-Variant Execution Security Defense Technology. *J. Cyber Secur.* **2020**, *5*, 77–94.
5. Bhamare, D.; Zolanvari, M.; Erbad, A.; Jain, R.; Khan, K.; Meskin, N. Cybersecurity for industrial control systems: A survey. *Comput. Secur.* **2020**, *89*, 101677. [CrossRef]
6. Min, B.H.; Borch, C. Systemic failures and organizational risk management in algorithmic trading: Normal accidents and high reliability in financial markets. *Soc. Stud. Sci.* **2022**, *52*, 277–302. [CrossRef] [PubMed]
7. Ruohonen, J.; Rauti, S.; Hyrynsalmi, S.; Leppänen, V. A case study on software vulnerability coordination. *Inf. Softw. Technol.* **2018**, *103*, 239–257. [CrossRef]
8. Wu, J. Development paradigms of cyberspace endogenous safety and security. *Sci. Sin. Informationis* **2022**, *52*, 189–204. [CrossRef]
9. Chen, N.; Jiang, Y.; Hu, A.Q. An Attack Feedback Dynamic Scheduling Strategy Based on Endogenous Security. *J. Inf. Secur. Res.* **2023**, *9*, 2–12.
10. Goktas, E.; Kollenda, B.; Koppe, P.; Bosman, E.; Portokalidis, G.; Holz, T.; Bos, H.; Giuffrida, C. Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In Proceedings of the 2018 IEEE European Symposium on Security and Privacy (EuroS&P), London, UK, 24–26 April 2018; pp. 227–242.
11. Chen, Y.; Wang, J.; Pang, J.M.; Yue, F. Diversified Compilation Method Based on LLVM. *Comput. Eng.* **2025**, *51*, 275–283.
12. Chen, Z.; Lu, Y.; Qin, J.; Cheng, Z. An optimal seed scheduling strategy algorithm applied to cyberspace mimic defense. *IEEE Access* **2021**, *9*, 129032–129050. [CrossRef]

13. Wei, S.; Zhang, H.; Zhang, W.; Yu, H. Conditional Probability Voting Algorithm Based on Heterogeneity of Mimic Defense System. *IEEE Access* **2020**, *8*, 188760–188770. [CrossRef]
14. Salamat, B.; Jackson, T.; Gal, A.; Franz, M. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In Proceedings of the 4th ACM European Conference on Computer Systems, Nuremberg, Germany, 1–3 April 2009; pp. 33–46.
15. Volckaert, S.; Coppens, B.; Voulimeneas, A.; Homescu, A.; Larsen, P.; De Sutter, B.; Franz, M. Secure and efficient application monitoring and replication. In Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC 16), Santa Clara, CA, USA, 12–14 July 2017; pp. 167–179.
16. Volckaert, S.; Coppens, B.; De Sutter, B.; De Bosschere, K.; Larsen, P.; Franz, M. Taming parallelism in a multi-variant execution environment. In Proceedings of the Twelfth European Conference on Computer Systems, Belgrade, Serbia, 23–26 April 2017; pp. 270–285.
17. Hosek, P.; Cadar, C. Varan the unbelievable: An efficient n-version execution framework. *ACM SIGARCH Comput. Archit. News* **2015**, *43*, 339–353. [CrossRef]
18. Koning, K.; Bos, H.; Giuffrida, C. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In Proceedings of the 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Toulouse, France, 28 June–1 July 2016; pp. 431–442.
19. Yeoh, S.; Wang, X.; Jang, J.-W.; Ravindran, B. sMVX: Multi-Variant Execution on Selected Code Paths. In Proceedings of the 25th International Middleware Conference, Hong Kong, China, 2–6 December 2024; pp. 62–73.
20. Yao, D.; Zhang, Z.; Zhang, G.; Wu, J. MVX-CFI: A practical active defense framework for software security. *J. Cyber Secur.* **2020**, *5*, 44–54.
21. Pan, C.; Zhang, Z.; Ma, B.; Yao, Y.; Ji, X. Method against process control-flow hijacking based on mimic defense. *J. Commun.* **2021**, *42*, 37–47.
22. Schwartz, D.; Kowshik, A.; Pina, L. Jmvx: Fast Multi-threaded Multi-version Execution and Record-Replay for Managed Languages. *Proc. ACM Program. Lang.* **2024**, *8*, 1641–1669. [CrossRef]
23. Cao, J.; Arya, K.; Garg, R.; Matott, S.; Panda, D.K.; Subramoni, H.; Vienne, J.; Cooperman, G. System-level scalable checkpoint-restart for petascale computing. In Proceedings of the 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS), Wuhan, China, 13–16 December 2016; pp. 932–941.
24. Savin, G.I.; Shabanov, B.M.; Fedorov, R.S.; Baranov, A.V.; Telegin, P.N. Checkpointing Tools in a Supercomputer Center. *Lobachevskii J. Math.* **2020**, *41*, 2603–2613. [CrossRef]
25. CRIU. Available online: https://criu.org/Main_Page (accessed on 8 May 2025).
26. Mashtizadeh, A.J.; Garfinkel, T.; Terei, D.; Mazieres, D.; Rosenblum, M. Towards practical default-on multi-core record/replay. *ACM SIGPLAN Not.* **2017**, *52*, 693–708. [CrossRef]
27. Laadan, O.; Viennot, N.; Nieh, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In Proceedings of the 2010 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, New York, NY, USA, 14–18 June 2010; pp. 155–166.
28. Lidbury, C.; Donaldson, A.F. Sparse record and replay with controlled scheduling. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, Phoenix, AZ, USA, 22–26 June 2019; pp. 576–593.
29. Lu, K.; Xu, M.; Song, C.; Kim, T.; Lee, W. Stopping memory disclosures via diversification and replicated execution. *IEEE Trans. Dependable Secur. Comput.* **2018**, *18*, 160–173. [CrossRef]
30. Volckaert, S.; Coppens, B.; de SutterMember, B. Cloning your Gadgets:Complete ROP Attack Immunity with Multi-Variant Execution. *IEEE Trans. Dependable Secur. Comput.* **2016**, *13*, 437–450. [CrossRef]
31. Matthew, M.; David, B. TACHYON: Tandem execution for efficient live patch testing. In Proceedings of the 21st USENIX Conference on Security Symposium (Security'12), Bellevue, WA, USA, 8–10 August 2010; pp. 617–639.
32. Zhou, D.; Tamir, Y. Hycor: Fault-tolerant replicated containers based on checkpoint and replay. *arXiv* **2021**, arXiv:2101.09584.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.