Fast VM Replication on Heterogeneous Hypervisors for Robust Fault Tolerance

Jean-Baptiste Decourcelle, Tu Dinh Ngoc, Boris Teabe, Daniel Hagimont

Fast VM Replication on Heterogeneous Hypervisors for Robust Fault Tolerance

Jean-Baptiste Decourcelle
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3
Toulouse, France

Tu Dinh Ngoc
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3
Toulouse, France

Boris Teabe
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3
Toulouse, France

Daniel Hagimont
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3
Toulouse, France

ABSTRACT

The reliability of virtualization infrastructures in the face of availability issues is a long-standing problem. Current fault tolerance approaches such as live VM replication are effective at addressing external, accidental issues (e.g. hardware failures, power cuts, environmental disasters); however, against an active attacker exploiting zero-day denial-of-service (DoS) vulnerabilities in the hypervisor itself, these approaches do not address the root cause of said vulnerabilities, and therefore cannot protect against these issues. This is made more relevant by the prevalence of DoS vulnerabilities among many widely used hypervisors.

We introduce *heterogeneous replication*, a new solution that enhances live VM replication so that VMs can be replicated across different hypervisors. We show that heterogeneous replication not only mitigates accidental failures from the external operational environment, but also mitigates DoS attacks arising from hypervisor vulnerabilities. We further show that heterogeneous replication can be used to increase the security of virtualized infrastructures without sacrificing availability.

We build *HERE*, our implementation of the heterogeneous replication concept for replicating a protected VM across hypervisor boundaries. We describe the implementation of *HERE*, including details on the necessary VM state replications, as well as a dynamic checkpoint interval adjustment scheme that maximizes VM protection based on load levels. We evaluate *HERE* using various benchmarks to show that *HERE* meets the goal of protecting VMs from availability issues while adapting to the VM's computing load.

CCS CONCEPTS

• **Security and privacy → Virtualization and security**; • **Computer systems organization → Cloud computing**.

KEYWORDS

heterogeneous hypervisors, VM asynchronous state replication, zero-day DoS vulnerabilities, virtualized infrastructures security

Middleware '23, December 11–15, 2023, Bologna, Italy

ACM ISBN 979-8-4007-0177-1/23/12…$15.00
https://doi.org/10.1145/3590140.3592849

ACM Reference Format:
Jean-Baptiste Decourcelle, Tu Dinh Ngoc, Boris Teabe, and Daniel Hagimont. 2023. Fast VM Replication on Heterogeneous Hypervisors for Robust Fault Tolerance. In *24th International Middleware Conference (Middleware '23), December 11–15, 2023, Bologna, Italy*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3590140.3592849

1 INTRODUCTION

As a core component of the modern cloud, hypervisors offer crucial isolation between multiple tenants, allowing various hardware resources to be securely shared and protected. They therefore need to guarantee the confidentiality, integrity and availability of client VMs (also known as the *CIA triad*). Unfortunately, like any other software, hypervisors can suffer from failures from various sources: *internal* failures stemming from implementation bugs and design issues [10, 22, 24]; or *external* failures originating from the operation environment, like hardware vulnerabilities [10, 23], hardware failures, or environmental disasters.

Regardless of the source of failure, cloud providers must propose a strategy to handle these issues. These strategies can generally be categorized into three groups:

a. *Active mitigation of vulnerabilities.* These strategies generally take either one of two forms: by applying live updates to a running system, therefore ensuring protection from a newly-discovered vulnerability as quickly as possible [54]; or by using multiple software implementations to impede exploitation, e.g. by dynamically switching between the two [24, 52];
b. *Fast reboot and restoration.* These strategies either aim to reinitialize a small contained component of the system, or optimize the reboot process such that they can be performed as fast as possible within the maintenance timeframe [7, 16, 32, 55]. They can be used either proactively (i.e. after a system update) or reactively (after a system failure).
c. *Proactive state replication.* These strategies advocate replicating the system's state, allowing the continuing of system operation on a replica in case of external failures [9, 53].

Nevertheless, these approaches all share a common property: *they cannot address internal failures caused by an unknown ("zero-day") DoS vulnerability.* In particular, mitigation-based approaches (a. and b.) require knowledge of the vulnerability-to-be-mitigated, while replication-based approaches (c.) can only recover from accidental failures, and not intentional exploits.

To clarify, let us focus on live VM replication as commonly supplied by hypervisor vendors. Live VM replication, known under various commercial name like vSphere Fault Tolerance (VMWare),

Remus (Xen), or COLO (KVM/Xen), provides a practical manner of addressing availability issues by ensuring continuity of service in the event of a system failure. These solutions generally work by periodically checkpointing the running state of a VM (including its CPU registers, memory, hardware states, etc.) and sending these data over the network to a secondary physical host to be reconstructed as a replica VM. In the event of a failure, the replica VM is quickly activated so as to take over service operation.

It is worth noting that current VM replication solutions require both the primary and secondary hosts to run the same hypervisor. As stated above, this setup is only capable of dealing with external failures, such as a hardware failure causing the primary host to stop working. However, in the case of a targeted attack on the hypervisor, for example by means of a DoS exploit, this approach alone is insufficient for protecting the system. In fact, if the primary host was brought down by a DoS vulnerability, activating the replica does nothing to address the problem; *the same vulnerability remains and can be exploited again on the secondary host.*

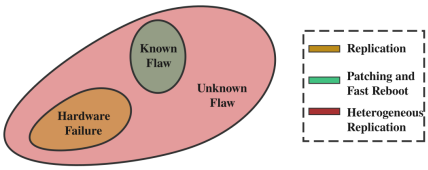

Figure 1: Coverage of various availability issue mitigation strategies.

To address this gap, we propose an extension to live VM replication to handle more than external failures. We introduce the concept of *heterogeneous replication*, which involves replicating VMs in real-time between different hypervisor implementations, combining the fault-tolerant properties of VM replication with the principle of software diversity [33] to account for both internal and external failures. Figure 1 provides a comparison of our solution to other failure mitigation strategies: unlike most existing strategies, heterogeneous replication proposes a defense against both hardware failures and intentional DoS exploits, including zero-day exploits for which a patch is not yet available.

To validate our concept, we implemented *HERE*, a platform for replicating VMs across different hypervisors by implementing an asynchronous state replication model [4], where the VM execution state on the primary host is buffered to the replica, and its activity is released following a checkpointing period. Based on previous studies on heterogeneous migration on hypervisors [21, 24], *HERE* extracts the necessary machine states from the running VM and translates it across heterogeneous hypervisors. Nonetheless, replication requires continuous VM state tracking and transfer, unlike migration. *HERE* further proposes two optimizations on the VM replication process:

Dynamic control of checkpointing period. Live replication often induces significant overhead on applications running in the replicated VMs, a reason why it is mostly used for critical workloads that cannot support a long downtime in the event of a failure. In other words, availability is more important than performance for such workloads. In current implementations of asynchronous state replication, VM execution state is released following a checkpointing period defined at the start of the VM. As such, this value effectively serves as an "average-case" recovery point objective which cannot be changed during the lifetime of the VM, and therefore cannot adapt to changing workloads that demand different checkpoint frequencies. Such a heuristic is not necessarily desirable for mission-critical applications, where as stated above, the goal is often to minimize data loss rather than to maximize application performance. *HERE* proposes a VM replication approach where the checkpointing period is dynamically adapted to the VM's activity based on two configurable parameters: 1) the desired replication overhead D; and 2) the maximum tolerable downtime in case of failure T_{max}. We show that our solution is capable of maximizing the replicated VM's protection while staying within the predefined constraints, regardless of the characteristics of the workload.

Optimized multithreaded replication. Current replication implementations mostly use a single thread to track and send dirty VM pages to perform replication of an entire VM. Such implementations are not capable of efficiently utilizing the full speed of modern network adapters, which can offer hundreds of gigabits of throughput per port. *HERE* proposes a multithreading scheme adapted to each phase of the replication process to reduce its overhead.

In short, we propose the following contributions in our paper:

- We analyze heterogeneous replication in relation to recent hypervisor vulnerabilities, and show how it can be used to harden virtualization infrastructure from exploits;
- We build *HERE*, our implementation of heterogeneous replication on top of two open-source hypervisors, Xen and KVM;
- We show that *HERE* is capable of breaking unknown exploits without causing system interruption by integrating exploit prevention techniques;
- We evaluate *HERE* at a machine scale to investigate its overhead when running typical network, computing and database benchmark workloads.

The rest of our article is organized as follows. Section 2 presents our observations and motivations for the creation of *HERE*. Section 3 presents the necessary background to understand our contribution. Section 4 presents the general overview of *HERE*. Sections 5 to 7 present the design and implementation of *HERE*. Section 8 presents the evaluation results. Section 9 discusses the related works. Section 10 concludes our paper.

2 MOTIVATION

The prevalence of denial-of-service vulnerabilities. We investigated vulnerabilities in five commonly-used virtualization products (Xen, Linux KVM, QEMU, VMware ESXi and Microsoft Hyper-V) published in an eight-year period from 2013 to 2020 on the NIST National Vulnerability Database [3]. For each product, we counted the total number of CVEs in the given period, the number of CVEs having an impact on availability (CVSS 2.0 Availability impact of Partial or higher), as well as the number of DoS-only CVEs (CVSS 2.0 Confidentiality and Integrity impact equals None).[1] We show

[1]Note that some projects, e.g. Linux, do not normally assign CVEs for security bugs [47]; as a result, this is not an exhaustive listing of all vulnerabilities. Nevertheless, we

our findings in Table 1. We observe that 1) among all hypervisors, most vulnerabilities have an impact on availability; and 2) DoS-specific vulnerabilities make up a substantial proportion of hypervisor vulnerabilities, especially with the open-source products we investigated (Xen, Linux KVM and QEMU). Our claim is corroborated by other works studying hypervisor vulnerabilities [29, 38].

Table 1: DoS vulnerability stats by hypervisor, 2013-2020. "Avail" signifies vulnerabilities *with an availability impact.* "DoS" signifies vulnerabilities that *only impact availability* ("DoS exploits").

Product	CVEs	Avail	Avail%	DoS	DoS%
Xen	312	282	90.4%	152	48.7%
KVM	74	68	91.9%	38	51.4%
QEMU	308	290	94.2%	192	62.3%
ESXi	70	55	78.6%	16	22.9%
Hyper-V	116	95	81.9%	44	37.9%

Little overlap between vulnerabilities of different hypervisors. As shown in other works [24], while individual hypervisors encounter multiple vulnerabilities of different severities every year, different hypervisors tend to share very few vulnerabilities overall (one critical and two medium-severity in a period of seven years). This means a software vulnerability on a hypervisor is unlikely to affect a different hypervisor, a property we utilize in our solution.

The difficulty of mitigating availability issues. Various techniques exist to address unintentional availability issues, e.g. hardware failures or environmental disasters. For example, live VM replication [9, 53] proposes recovery from hardware errors by replicating a VM's execution state onto a remote host, as we presented in Section 1. Nevertheless, live replication alone is not sufficient to address *intentional* exploit attempts, including DoS exploits, as failing over from the primary to the replica does little to address the actual underlying vulnerability.

Denial-of-service vulnerabilities can originate from multiple root causes: memory bugs (use-after-free/null dereference/buffer overflow/etc.) causing crashes upon handling untrusted input; resource exhaustion attacks (e.g. Slowloris, fork bombs); or algorithmic complexity attacks, a subclass of resource exhaustion attacks that focus on algorithmic behaviors (e.g. evil regexes, hash table flooding). DoS issues resulting from this class of attacks are especially difficult to find and fix [30]. The wide spectrum of possible DoS causes makes identifying and stopping them a complex task that requires the use of many different classes of mitigations. Defenses against DoS can take the form of sanitizers (e.g. AddressSanitizer [37] for memory bugs; SlowFuzz [30] and PerfFuzz [19] for algorithmic complexity issues), or active patching techniques to respond to newly-known DoS exploits [32, 54]. While commonly used, patching has the disadvantage of requiring the existence of an update or workaround before the problem could be corrected; the system could have been brought down well before a patch is widely available.

believe that the enumerated CVEs are a representative selection of vulnerabilities in each product.

Exploit mitigation: or how to downgrade exploits to DoS. Current operating systems and applications employ numerous exploit mitigation techniques such as NX bits, address space layout randomization, control-flow integrity [1], checked pointers [11, 20, 51], syscall filters [6, 46], and so on. While an effective barrier against attacks that aim to take control over the target, exploit mitigations do not address the root cause of the vulnerability; upon detecting an active exploitation, the mitigation process cannot easily revert the resulting state corruption, and therefore the most secure response is to force a system crash. This approach essentially turns an exploitable vulnerability into a denial-of-service attack onto the concerned system.

3 BACKGROUND

In this section, we present the general principles of live VM replication-based fault tolerance, as well as its implementation in the Xen hypervisor.

3.1 Live VM Replication

VM replication provides a system-level fault tolerance for critical workloads. It does not require any changes to applications running inside the protected VM. Moreover, as the whole system state is replicated, hardware failures remain fully transparent to users. Nevertheless, the frequent state synchronization between the primary and secondary hosts imposes a significant performance penalty. In general, hypervisors provide two main models of state replication: 1) asynchronous state replication [4] (ASR for short), where the VM execution state on the primary host is buffered to the replica, and its activity released via static periodic checkpoints; or 2) VM lockstepping [9] (LSR for short), where both the primary and replica execute at the same time in order to lower the replication overhead, with their divergence closely controlled via a replication controller. While LSR reduces the overhead on the replicated VM compared to ASR, it necessitates a replication controller that implies significant similarities between the device model implementations of the primary and replica VM to minimize the resulting divergence. As a result, we make use of the ASR replication model, so as to enable more diversity between different implementations.

3.2 Remus: Live VM Replication on Xen

Xen is a type-1 hypervisor that consists of a hypervisor core that ensures isolation between different VMs, a privileged VM ("domain-0") responsible for managing the creation and destruction of other VMs ("domains"), and driver domains that manage and partition device resources. Each virtual device in a Xen domain can run in one of several modes: emulated, where the virtual device approximates a real hardware device; paravirtualized (PV), where the virtual device is exposed through an optimized API shared between the hypervisor and the VM; and passthrough, where the VM is given direct access to real hardware under the isolation of an IOMMU.

Xen provides an ASR-based VM replication solution called Remus [4]. Figure 2 presents the general architecture of Remus. In summary, Remus consists of a state replication engine that sends the VM states to a remote host, which is then responsible for loading this state into the replica VM.

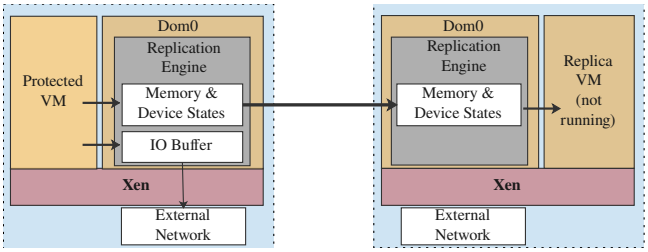

Figure 2: Architecture of Remus VM replication.

Figure 3 illustrates the two main steps of setting up VM replication in Remus. The running VM is first seeded over to the second host using live migration ❷, where the unreplicated memory pages (cf. dirty pages) are copied iteratively until the number of dirty pages goes below a certain threshold, or the maximum iteration count has been reached (5 iterations in the case of Xen). The page copying process uses the same dirty tracking and iteration thresholding algorithm as normal live migration, making use of shadow paging [2] or hardware-assisted dirty page logging (e.g. PML) whenever available. At the end of the seeding step ❸, Xen pauses the VM and transfers over any remaining dirty pages, as well as the vCPU and virtual device states over to the secondary host.

Once the initial state of the VM has been copied, it enters a continuous replication phase where the VM's state is synchronized with its replica using checkpoints at regular intervals noted T. Remus implements this continuous replication via repeated live VM migrations ❹, where in each iteration (1) the VM is paused; (2-3) the dirty memory and device states are copied to the secondary host; (4) the backup host acknowledges the reception of the checkpoint; and finally (5) the primary VM is resumed. We call t the period between the steps (1-5) where the primary VM is paused. To ensure consistency of any network operations between the protected VM and external clients in case of failure of the primary host, all outgoing I/O traffic of the primary VM is buffered during the entire execution period T, and only released once the corresponding checkpoint has completed (step 6). Upon failure of the primary host, the VM's state reverts back to that of the previous checkpoint, and VM activity resumes on the secondary physical host.

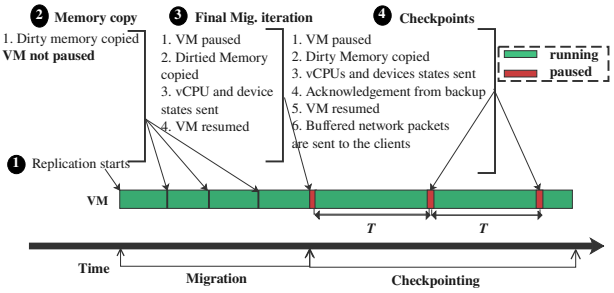

Figure 3: Live VM replication workflow with Remus.

4 *HERE*: HETEROGENEOUS REPLICATION FOR ROBUST FAULT TOLERANCE

In Section 2, we observe that replication/failover techniques are effective against unintentional failures, and patching techniques are effective against known DoS exploits for whom patches and/or workarounds are available. However, neither of these techniques are capable of tackling both types of availability issues at the same time, nor are they capable of addressing unknown DoS exploits (which can be discovered ahead-of-time by adversaries using the static analyzers mentioned above).

With *HERE*, we make the key observation that replication techniques can be combined with the principle of software diversity to cover all three aforementioned issue categories. Software diversity has been the target of previous research [17, 33] as a method for increasing the resiliency of a software system to attacks.

4.1 Threat Model

As stated above, the goal of *HERE* is to provide a virtualized infrastructure that is protected from accidental failures of the underlying software/hardware, and is resistant to active exploitation of DoS vulnerabilities. We assume that besides accidental system failures, a malicious user can use a hypervisor vulnerability to launch DoS attacks against the hypervisor host, or against other protected VMs running on the same host. These attacks can originate from a guest VM controlled by the guest attacker, or any external services with a connection to the hypervisor hosts. Figure 1 and Table 2 summarize the general coverage of our solution.

Table 2: *HERE*'s coverage of DoS issues from various sources.

Source	Guest failure	Host failure
Accidents; HW/SW errors	Yes	Yes
Guest user	No	Yes
Guest kernel	No	Yes
Other guests	Yes	Yes
Other services	Yes	Yes

4.2 Overview of Heterogeneous Replication

The main idea of *HERE* is to extend live VM replication so that protected VMs can be replicated from one hypervisor to a different hypervisor. We present below how *HERE* can enhance the robustness of a virtualized system:

HERE *against accidental failures.* Being based on live VM replication, *HERE* provides protection against unintentional failure of the primary virtualization host. In case of a hardware failure, power loss or a similar incident, the replica VM automatically activates, restoring access to any service running inside the protected VM.

HERE *against DoS exploits.* Once the primary host is brought down by a DoS exploit on its hypervisor H_A, whether originating from the guest or an external source, the replica VM is then activated on a secondary host running a completely different hypervisor H_B. As stated in Section 2, two hypervisors with differing

implementations are not likely to be affected by the same software vulnerability. Therefore, the secondary host is not likely to be vulnerable to the same issue, rendering said DoS exploit unusable.

5 ARCHITECTURE OF *HERE*

Figure 4 shows the general architecture of the platform in a simple VM replication scenario. *HERE* is installed as a user-mode component of both hypervisors to replicate selected VMs. *HERE* consists of the following core components: a *state manager* that manages VM execution and extracts VM execution states to be replicated; a *device manager* that handles virtual device-specific states; a *state translator* that supports replication between heterogeneous hypervisors; and finally, a *dynamic checkpoint period manager*.

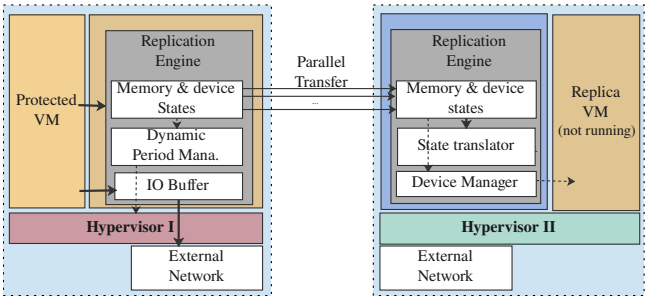

Figure 4: Architecture of *HERE*.

5.1 State Manager

This component is responsible for controlling the VM's execution throughout the replication process, and for replicating the VM's virtual CPU and memory states. Following the VM replication workflow presented in Section 3.2, the state manager makes use of the source hypervisor's memory tracking facilities throughout the initial migration and subsequent periodic checkpoints. To optimize the replication process, the state manager uses a multithreaded implementation adapted to each individual phase of the VM replication process. We present further details on the implementation of the state manager in Section 7.

5.2 Device Manager

This component handles I/O-level replication of networking states, by buffering outgoing traffic and releasing buffered packets when each checkpoint is acknowledged by the remote replica. We opted for a heterogeneous device model strategy, i.e. the primary and secondary hypervisors provide different device models to the protected VMs. The secondary hypervisor's device manager is responsible for switching the VM from the primary hypervisor's device model to the secondary's. Such a design avoids sharing the same vulnerabilities when using the same device models between the primary and secondary VMs.

5.3 State Translator

A prerequisite of heterogeneous replication is the ability to translate VM states from one hypervisor to another. The state translator component is responsible for translating the virtual CPU, memory and device states of the VM into a format that can be loaded onto the replica VM. Following previous studies on live migration between heterogeneous hypervisors [21, 24], the state translation is done by copying the contents of vCPU registers into a common format, then restoring the corresponding data into the secondary hypervisor's format. To make sure that the protected VM can be safely resumed on the secondary hypervisor, *HERE* ensures virtualization compatibility between both hypervisors by adjusting platform features as necessary.

5.4 Dynamic Checkpoint Period Manager

In Section 3.2, we established that Remus's fixed-frequency checkpointing necessarily creates a delay between the primary and replica VM. Lockstepping approaches like COLO significantly reduce the delay by simultaneously executing the primary and replica, however they require substantial similarities between the primary and secondary hypervisor implementations to keep replica divergence under control. As a result, these approaches are not appropriate in a heterogeneous environment using multiple hypervisors.

In checkpointed replication solutions like Remus, all outgoing traffic must be buffered during the entire duration of the checkpoint. The amount of incoming traffic lost in case of a failover event is therefore proportional to the checkpoint interval. However, Remus uses a fixed checkpoint interval that cannot adapt to the workload being protected.

We define the performance degradation caused by a checkpoint for a given period T in Equation 1, with t being how long the VM is paused during that checkpoint (see Figure 3). We will see that t depends on T. We chose the real checkpoint duration rather than the replication traffic's packet count to account for variations in the replication network interface's performance, for example due to network congestion.

$$D_T = \frac{t}{t+T} \tag{1}$$

As stated in Section 1, we focus our attention on critical workloads where availability is preferable over performance. We therefore aim not to keep the same checkpointing interval T, but to reduce the checkpointing interval (i.e. checkpointing more frequently) so as to reduce the amount of potential computation loss in the event of a host failure. To this end, the dynamic checkpointing manager controls the checkpointing interval under the constraints of two parameters: the desired performance degradation denoted D (which is a *soft limit* that can be exceeded at high loads), and the maximum tolerated checkpointing interval T_{max} (which is a *hard limit* that must be followed in all cases). The main task of this component can therefore be modeled in Equation 2.

$$\text{Find } T \text{ such that } \begin{cases} D_T \approx D & \text{(soft)} \\ T \leq T_{max} & \text{(hard)} \end{cases} \tag{2}$$

Analogous to previous work like Adaptive Remus [5], HERE allows the checkpoint period to vary, but the comparison stops there. Contrary to HERE, Adaptive Remus targets IO applications in particular and provides only two period settings: a default setting, and a lower checkpointing period setting enabled when IO activity is detected in the VM. The key idea is that reducing the checkpoint

period decreases the buffering time of outgoing traffic, which speeds up communication and thus service delivery.

T varies over time depending on the activity of the workload. In other words, the dynamic checkpoint interval calculation is done depending on the VM's memory activity. As presented in Section 3, each checkpoint's pause period involves copying any dirtied memory pages to the replica, followed by transfer of vCPU states before the primary VM is resumed. The pause/resume and state transfer steps take a constant amount of time regardless of VM activity; however, the memory copying step might take a variable amount of time based on how many pages need to be copied. In other words, the pause duration t is a combination of the variable cost $f(N)$ for copying N memory pages divided by the parallelism factor P plus an amortized constant cost C (Equation 3):

$$t = \frac{f(N)}{P} + C \tag{3}$$

We use a step-based approach to find an interval containing the target value T, i.e. for every checkpoint, we adjust T by fixed increments σ of the period length. Algorithm 1 shows the recalculation of the period T after every checkpoint. At the beginning of the continuous replication process, we set $T = T_{max}$ to avoid exceeding the replication interval constraint. For every subsequent checkpoint, we measure the known pause duration t_{curr} and calculate the associated performance degradation D_{curr}. In case we have a degradation budget (i.e. $D_{curr} \leq D$), we adjust the interval by one step σ, preserving the last-known good interval as T_{prev} (lines 7 and 8). If we exceeded the degradation budget (i.e. $D_{curr} > D$), we walk back the interval to the previously-known good value (line 10). If restoring the interval T_{prev} is not effective, we directly adjust T to the midpoint between the current T and T_{max} (line 13).

Algorithm 1 Checkpoint interval adjustment algorithm.

```
1:  T ← T_max
2:  D_prev ← D
3:  while perform checkpoint do
4:      t_curr ← measured pause duration
5:      D_curr ← t_curr / (t_curr + T)
6:      if D_curr ≤ D then
7:          T_prev ← T
8:          T ← T − σ
9:      else if D_prev ≤ D then
10:         T ← T_prev
11:     else
12:         T_prev ← T
13:         T ← round((T + T_max)/2, σ)
14:     end if
15:     D_prev ← D_curr
16: end while
```

The last question that remains in our algorithm is the computation of the pause duration t at each checkpoint. There is a linear relationship between the transfer duration and the number of pages, as illustrated in Figure 5. In fact, as we demonstrate later in our evaluations, the higher the number of pages to be transferred, the longer the page transmission time. We therefore model t as the linear Equation 4.

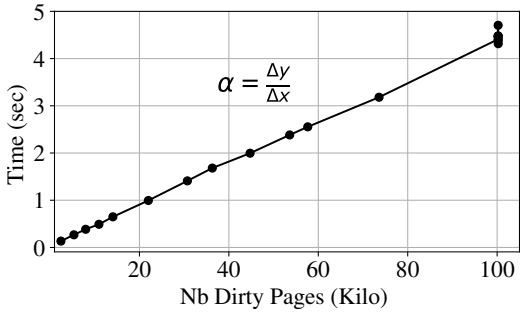

Figure 5: The linear relationship between number of dirty pages and page sending time $f(N) = \alpha N$.

$$t = \frac{\alpha N}{P} + C \tag{4}$$

6 SECURITY PROPERTIES OF *HERE*

Effectiveness of HERE *against DoS exploits.* One can argue that once the replica VM has switched onto the secondary hypervisor H_B, the system remains vulnerable to another DoS attack. However, in such cases, *HERE* effectively doubles the effort needed to completely bring down the protected infrastructure. Rather than only having to craft one DoS exploit, the attacker now needs to use two different exploits at the same time, as failure to bring down both systems means any remaining system can continue running the protected services.

Increasing security with exploit mitigation without sacrificing availability. Another benefit of *HERE* is that it can be combined with exploit mitigation technologies to maintain system availability while increasing security at the same time. As stated above, an exploit mitigated by one of these technologies might end up crashing the system. However, the live VM replication of *HERE* helps ensure that even in case of such a crash, the protected VM keeps operating thanks to the secondary hypervisor.

7 IMPLEMENTATION

We organize this section as follows: Section 7.1 describes our experimental environment; the latter sections present the implementation of *HERE* in detail.

7.1 Environment

We implemented *HERE* on two exemplary hypervisors: Xen 4.12 and KVM (using `kvmtool` as the userspace component) in order to replicate a running VM from Xen to KVM. This implementation choice is motivated by Xen and KVM being commonly-used, open-source hypervisors with different architectures.

7.2 State Manager

We modified Xen's built-in migration code in two ways. Firstly, we extended Xen's dirty page tracking code to allow independent tracking of dirty pages for each vCPU. To this end, we implemented a mechanism in the Xen kernel that gathers dirty page information

from each vCPU using Intel PML[2] into a ringbuffer that could be read independently for that vCPU without having to interrupt other vCPUs. Secondly, we implemented support for multithreaded migration to speed up the live VM replication process. Multithreading in *HERE* is done in two different ways:

(1) For the seeding phase of VM replication (see Section 3.2), we set up individual migrator threads for each vCPU of the VM. Each migrator thread then sends its own dirty pages to the second host following Xen's migration iteration algorithm. As the VM stays running during the entire first migration phase, the same memory page could be modified by multiple vCPUs, leading to consistency problems on that memory page. To account for this issue, the migration process keeps track of "problematic" pages that are potentially impacted by consistency problems, and resends these pages during the final stop-and-copy step. A page is considered "problematic" if it has been transferred several times by different threads, meaning that it was modified by different vCPUs. This method is appropriate when the VM's utilization level is low (e.g. when VM replication is being initialized after booting and before the VM goes into service).
(2) During the continuous replication phase, we again start multiple migrator threads, where each thread is responsible for sending chunks of the VM's memory. Specifically, we split the VM's memory ranges into disjoint regions of uniform size (2 MB in our implementation). These regions are then assigned to the migrator threads in a round-robin fashion. During each checkpoint operation, each thread reads the shared dirty bitmap, finds pages it needs to migrate using the above assignment, then copies it to the remote host.

7.3 Device Manager

As we opted for a heterogeneous device model strategy, this component implements the configuration of devices on the replica VM. Contrasting with heterogeneous migration [21, 24] where the failover is triggered by the administrator, failover can happen anytime in a replication system. Therefore, HERE cannot prepare for it using the same strategy as for example unplugging devices on the primary VM and replugging them on the secondary VM. Instead, we use the following approach. Recall that I/O operations on the replica VM are only performed when it is activated following a failure on the primary host; upon a failure event, we instruct the guest VM to safely unplug its old I/O devices, followed by installing new devices that are compatible with the secondary hypervisor. Replication does not work on passthrough devices because backtracking is usually necessary and only provided by para-virtualized devices. As a consequence, HERE only handles PV devices.

7.4 State Translator

This component is inspired by previous work on migration on heterogeneous hypervisors [21, 24]. We implemented the necessary translation of VM migration states from Xen to KVM, including CPU registers, memory pages, timers, interrupt controllers, and any other emulated features. Additionally, we adjusted CPU features of the protected VM exposed by the CPUID instruction on both Xen and KVM to make sure that the protected VM can safely resume on the secondary hypervisor.

[2]Intel Page Modification Logging is a hardware feature that allows for tracking modified memory pages of VMs per vCPU.

7.5 Dynamic Checkpoint Period Manager

This component is responsible for controlling the VM checkpointing period as described in Section 5. We again modified Xen's migration algorithm to measure the last checkpoint duration and calculate the resulting checkpoint frequency to be applied in the next iteration. As stated previously, we used checkpoint duration as our main metric rather than relying on the number of sent packets because the resulting degradation can vary depending on existing traffic on the network adapter.

7.6 Implementation Size

We implemented *HERE* in approximately 9000 lines of code, of which 800 lines of code belong to the Xen kernel, and 8000 lines belong to Xen and KVM's userspace components (`xl`, `libxl`, `libxc` and `kvmtool`). We additionally inserted a minimal Linux kernel module of 150 lines of code into the guest in order to receive migration events from the device manager. No changes were made to the host's Linux kernel (Dom0). From the minimal changes needed to the host's OS kernel (Xen), *HERE* should have minimal impact on the host hypervisor's security properties.

7.7 Integrating *HERE* in Data Centers

HERE forces the usage of multiple hypervisors in a data center. As previous work has demonstrated [24], some data centers already manage multiple hypervisors. Thus, the heterogeneity of virtualization systems in data centers is a concept already popular. *HERE* advocates to use this heterogeneity for a robust replication. In addition, virtualization systems are very often administered by tools such as Openstack which is based on standard libraries such as libvirt which interfaces with all hypervisors.

8 EVALUATION

In this section, we present our evaluation results of *HERE*. With these evaluations, we plan to answer the following questions:

- How does *HERE* ensure continuation of service when confronted with a denial of service only attack on the primary hypervisor?
- How does *HERE*'s migration phase perform compared to the default VM migration?
- How does *HERE*'s checkpointing phase perform compared to Remus?
- How accurate is *HERE*'s dynamic period manager at detecting workload variations and defining an appropriate checkpointing period?
- What is the overhead of *HERE* on protected VMs, and how does that impact application performance?

8.1 Experimental Setup

Hardware. We evaluated *HERE* on two servers with their characteristics described in Table 3. HERE does not implement migration and replication on heterogeneous hardware yet. This limitation is

the subject of future work. We reserved 10 GB of memory for the privileged domain Dom0 in Xen. For all of our evaluations, we used the Ethernet adapter exclusively for VM communications, while the Omni-Path interconnect is reserved for migration and replication.

Table 3: Hardware configurations.

Component	Characteristics
CPU	2x Intel Xeon Gold 6130
Memory	192 GB (96 GB per node)
Ethernet	Intel X710 10GbE (NUMA node 0)
Interconnect	Intel Omni-Path HFI 100 (NUMA node 0)
Storage	1.0 TB HDD
OS	Debian 10

Applications. We evaluated *HERE* using several types of applications, including CPU-, memory- and I/O-intensive applications. Table 4 provides a list of benchmarks used in our experiments.

8.2 Analyzing *HERE*'s Security Properties

In this section, we analyze *HERE*'s ability to protect guest OSes from DoS attacks.

The benefits of heterogeneity. As stated in section 4.2, it is unlikely for two different hypervisors to be subject to the same software vulnerability unless they share a substantial part of their source code. For example, since both Xen and QEMU-KVM hypervisors use QEMU to emulate their device models, implementing *HERE* on them would not have protected the guest from QEMU vulnerabilities (e.g. CVE-2015-3456). We avoided this scenario by implementing our solution on top of Xen and Linux KVM/`kvmtool`.

Vulnerability analysis. Table 1 shows that 80% to 95% of hypervisor vulnerabilities documented between 2013–2020 have an effect on availability. Among these vulnerabilities, approximately half solely impact availability without affecting the integrity of the hypervisor or its VMs; we therefore consider these to be DoS-only vulnerabilities. For instance, 49% of Xen's vulnerabilities fall into this category; 41% of vulnerabilities affect both availability and integrity; and only 10% do not affect Xen's availability.

To explore the impact of *HERE* in the event of an attack, we further studied the 49% DoS-only attacks on Xen following the same approach used by Diego et al. [29]. In other words, we investigated each vulnerability's root cause, attack vectors, application targets, the post-attack outcome of the hypervisor and running VMs, and finally the applicability of *HERE* to these vulnerabilities. We ignored vulnerabilities that also affect integrity because unless some mitigation technique can downgrade them into availability only vulnerabilities, replication is useless against them. Table 5 summarizes our findings.

Table 4: Description of benchmarks.

Benchmark (metric)	Description
Memory benchmark	Write-intensive benchmark using a defined memory percentage
YCSB RocksDB (ops/sec)	Database benchmark suite
SPEC CPU 2006 (ops/sec)	CPU/memory intensive programs
Sockperf (latency)	Network-intensive benchmark

Regarding the attack vectors of these vulnerabilities, we observed the following partition: 25% originate from virtual device management (whether emulated, paravirtualized or passthrough); 20% originate from hypercall processing; 12% originate from vCPU management; 7% originate from shadow paging; 2% originate from VM exit handling; and the remaining 34% originate from other Xen components. Finally, regarding the target components of each vulnerability, we find that 84.5% target the Xen hypervisor core, Dom0 and tools; 12.5% target the guest OS; and the remaining 3% target other Xen software (e.g. Xenstore). Finally, our findings show that more than half of DoS-only vulnerabilities are launched from a *guest user-space* process; the remaining half require *ring-0 kernel* privileges to be executed.

While investigating the post-attack outcome of each vulnerability, we defined three possibilities: (1) where the targeted component *crashes* and is completely shut down; (2) where the target *hangs* (i.e. stops responding to all requests); and (3) where the target malfunctions so as to cause *starvation* of certain resources. We observe that a large majority of DoS-only vulnerabilities (79%) caused its target to crash; 13% caused a hang; and only 8% caused a resource starvation. Nevertheless, we note that regardless of a vulnerability's outcome, *HERE* remains applicable as a countermeasure when an attack is detected.

In the current implementation of *HERE*, we rely on a periodic heartbeat between the primary and replica hosts to ensure that the hypervisors are functioning normally. Furthermore, it is possible to couple *HERE* with works such as [25, 31] that focus on detecting attacks on hypervisors. Once an attack is detected, the affected hypervisor can safely crash; control of the VM is then handed over to the second hypervisor, and the VM continues to function.

Table 5: Distribution of DoS-only vulnerabilities by target, post-attack outcomes, and applicability of *HERE*.

Target		Outcome		*HERE*
84.5%	Xen, Dom0, Tools	66.0%	Crash	Applicable
		13.0%	Hang	Applicable
		5.5%	Starvation	Applicable
12.5%	Guest OS	10.0%	Crash	Applicable
		2.5%	Starvation	Applicable
3.0%	Other software	3.0%	Crash	Applicable

8.3 Evaluating VM Migration Optimizations

In this section, we evaluate the effectiveness of the optimizations to the VM migration process implemented in *HERE*.

We compared a VM migration implementation with the optimizations proposed in *HERE* to Xen's default implementation using two different scenarios: (1) migrating an idle VM; and (2) migrating a VM running our memory microbenchmark described above.

Migrating idle VMs. We evaluated the migration time of an idle VM equipped with 4 vCPUs and various memory sizes ranging from 1 to 20 GB using both *HERE* and Xen. The resulting migration times are shown on the left side of Figure 6. We note that despite our

migration being slightly slower for small VM sizes (1-2 GB RAM), we gain increasing migration performance of up to 25% for larger VMs (8-20 GB RAM). This is explained by the VM migration time being dependent on memory size: the more memory the VM has, the more time it takes to copy it to the remote host, and therefore the bigger the impact of *HERE*. According to [24], for any number of vCPUs or memory size, migration time is the same for both HyperTP and Xen's default implementation. We conclude that HERE provides the same gain of performance compared to HyperTP.

Migrating VMs under memory load. We implemented a benchmark that performs random memory operations to artificially load the migration process. We ran the benchmark on a VM with 4 vCPUs and 20 GB of RAM, while varying the amount of memory allocated to the benchmark between 10% and 80% of total memory (i.e. 2 GB to 16 GB). The right side of Figure 6 shows the migration times under each condition. We observe that migration speed was significantly impacted even at 10% memory consumption; however, the optimizations of *HERE* improved migration time by nearly 49% compared to Xen live migration (and thus HyperTP).

8.4 Evaluating VM Replication Optimizations

In this section, we aim to show the performance improvements brought by our VM replication optimizations. To this end, we configured both *HERE* and Remus with a fixed replication period of 8 seconds, then compared the memory transfer time per checkpoint of each solution under various VM memory sizes using a 30% memory load for our benchmark. We also measured the time needed to resume the protected VM, which corresponds with the period from when the secondary host is aware of a primary failure to when the replica VM resumes operation.

Checkpoint memory transfer time comparisons. In this evaluation, we focus on the memory transfer time for each checkpoint in order to calculate the performance degradation following Equation 1. Figures 8a and 8b show respectively the checkpoint transfer times for the idle VM and memory microbenchmark VM; Figures 8c and 8d present the calculated degradations. First of all, we note that the memory microbenchmark causes a significant increase in checkpoint transfer time, as is expected from a memory write-intensive benchmark. Nevertheless, we observe that *HERE*'s optimizations provide a checkpoint transfer time up to 70% lower than that of Remus on idle VMs and 49% lower on loaded VMs. This points to our multithreaded checkpoint transferring algorithm being more performant than that of Xen's. Concerning the calculated performance degradation, we observe that our solution brings little improvement on idle VMs (considering that the total calculated degradation of Remus is less than 1%); however on loaded VMs our solution shows a clear improvement over Remus, since the checkpoint transfer time starts becoming more significant at higher memory loads.

Replica VM resumption time. We also evaluated the time required to resume the replica VM with *HERE* in the event of a failure. Using the same configuration as the previous experiments, we stopped the primary VM and counted the time from the beginning of the failover process to when the replica restarts its operation. Figure 7 presents the obtained results. We observe that the VM resumption time on *HERE* is in the order of 10 milliseconds. The short length of this

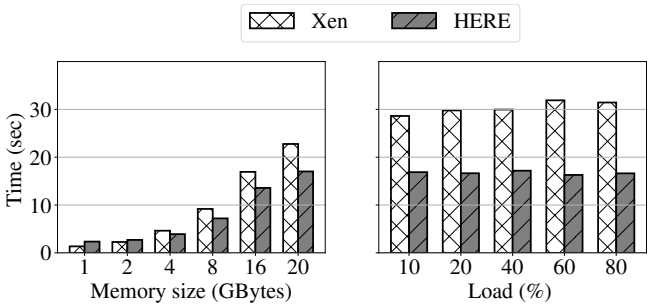

Figure 6: Migration times of idle VM (left) and memory benchmark VM (right).

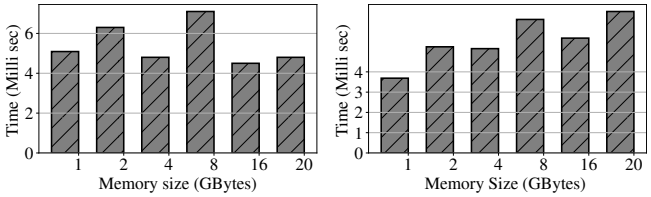

Figure 7: Replica resumption times for idle VM (left) and microbenchmark VM (right).

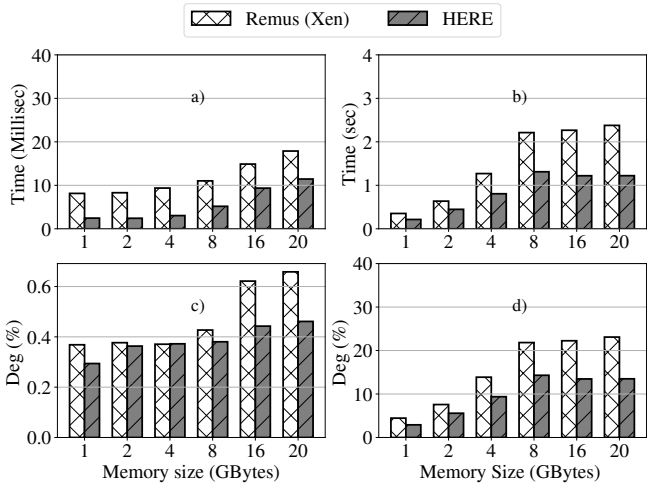

Figure 8: Checkpoint transfer times of idle (a) and loaded VMs (b); Perf. degradations of idle (c) and loaded VMs (d).

duration is mostly due to the more efficient userspace component `kvmtool`. We also note that this resumption time does not increase with the VM's memory size or load level. Note that the total time until the VM continues its operation is also dependent on other factors (checkpoint frequency, heartbeat frequency, etc.)

8.5 Evaluating the Dynamic Checkpoint Period Manager

In this section, we evaluate *HERE*'s dynamic checkpoint period manager, responsible for adapting the checkpoint period to the

workload. In a VM with 4 vCPUs and 8 GB of memory, we configured our memory microbenchmark to execute a workload that varies with time; the benchmark is configured to use 20% of the memory at first, increasing to 80% afterwards and falling back to 5% at the end. We configured *HERE* with a desired degradation $D = 0.3$ (i.e. 30%) and $T_{max} = 25$ seconds. We monitored the selected checkpoint period T and the instantaneous calculated degradations D_T over the lifetime of the benchmark. The obtained results are presented in Figure 9, in which the benchmark's load level is denoted by the green line named *Load*.

We observe that following an increase in workload from 20% to 80% around the 20th second, *HERE* automatically adapts to this load and increases the checkpoint period at the 40th second (as shown by the blue line named *Period*). Correspondingly, when the load level falls back down to 5% at the 125th second, the checkpoint period is also automatically adjusted shortly after. The bottom graph of Figure 9 shows how *HERE* adapts the checkpointing parameters following the user's given constraints. Except for a short adjustment period after each change in load levels, the resulting performance overhead (shown by the black line denoted *Overhead*) follows the preset overhead value of 30% (denoted by the red line named *Set Overhead*). We conclude that *HERE*'s dynamic checkpoint period manager respects the user's provided VM replication parameters.

To validate this conclusion, we carried out the same evaluation with the *Workload A* from YCSB benchmark suite (more details on the benchmark configuration is provided in the next section). The obtained results are presented in Fig 10. The top figure shows the period variation, while the bottom presents the calculated degradation. We note that during the execution of the benchmark, *HERE* enforces a degradation close to the set value of 30%. This is also observable on the application performance; the workload on *HERE* shows a performance of 28406 ops/sec, a slowdown of approximately 33.6% compared to the baseline performance of 42779 ops/sec.

8.6 Database, CPU-Intensive and I/O Benchmarks

In this section, we test the performance and adaptability of *HERE* on three different benchmark suites: YCSB, SPEC CPU 2006 and Sockperf. Each benchmark was evaluated under 3 conditions with *HERE*: (1) an execution using *HERE* with a defined T_{max} and degradation set to 0% to enforce T = T_{max}; (2) *HERE* with a given degradation and no T_{max} ($T_{max} = \infty$); and finally (3) *HERE* with a defined degradation and T_{max}. Table 6 presents a summary of these configurations. On all figures in this section, the numbers above the bars represent the performance degradation in percentage.

Evaluations with YCSB.. We executed YCSB benchmark suite running on a single VM using the 6 included workloads A to F, each with 1 million records and 4 million operations. The results obtained for all scenarios are presented in Figs 11 12 13. More precisely, Fig 11 presents the results of *HERE* with a fixed T = T_{max}. The figure also contains results with Remus configured with the same periods. We can observe that *HERE*'s optimizations provide a replication with less overhead than Remus. For instance, with Workload A, Remus provides a 52% and 45% respectively with periods of 3 and 5 seconds, while *HERE* induces 32% and 28% for the same periods.

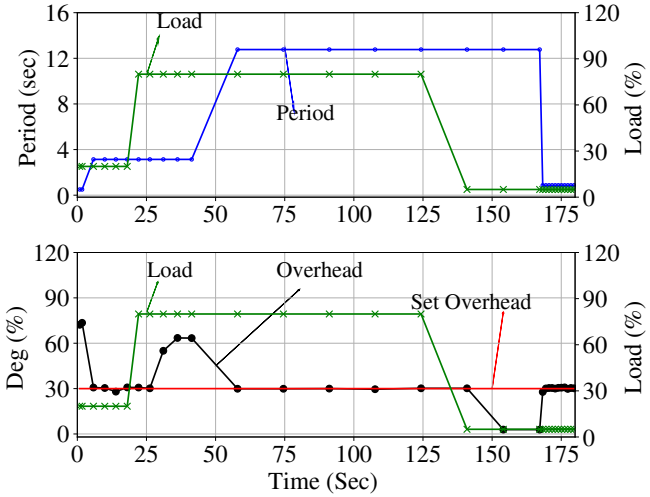

Figure 9: Dynamic checkpoint period versus load level (top); overhead versus load level (bottom).

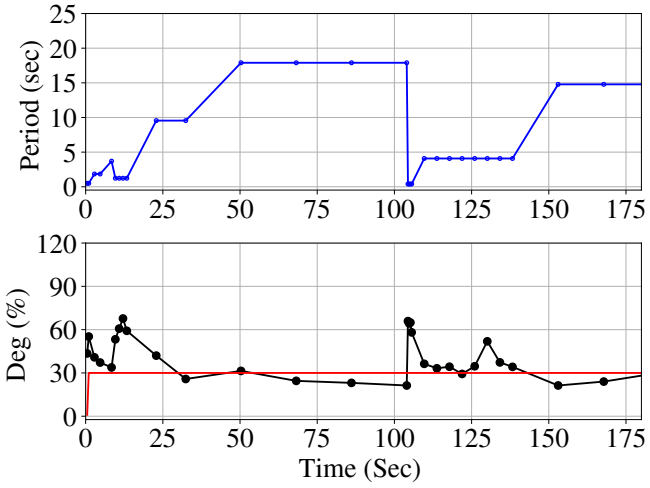

Figure 10: Dynamic checkpoint period for YCSB workload A.

Fig 12 presents the results of *HERE* with a set value for degradation. We note that for smaller degradations (20 and 30%), *HERE* successfully respects the defined value. However, this is more difficult for a higher degradation value of 40%. This is simply due to the software and hardware cost of the stop-and-go replication process. Indeed, with a high degradation value, checkpoints are more frequent, therefore hardware overheads such as cache misses, TLB misses and software overheads for scheduling the VM are increased. All of this impacts the application's observed degradation level.

Finally, Fig 13 presents the results of *HERE* with defined degradation and T_{max} values. We note that the observed degradation numbers stay near the defined values, ranging from 48-53% for $HERE_{(3Sec,40\%)}$ and 33-38% with $HERE_{(5Sec,30\%)}$. This is explained by the fact that *HERE* forces the protected VM to follow the desired degradation. This conclusion is confirmed on Fig 11 because we can note that with periods of 3 and 5 seconds, the observed degradations on YCSB are below 40% and 30% respectively.

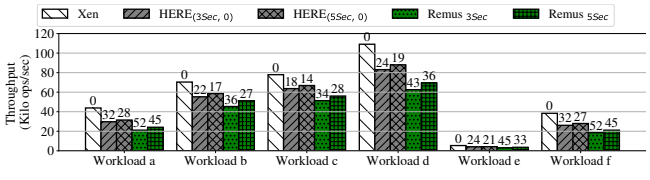

Figure 11: Evaluations of Remus and *HERE* using YCSB with the same checkpoint period.

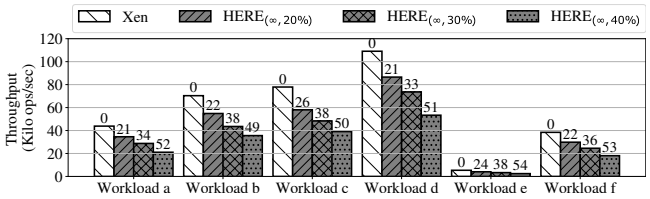

Figure 12: Evaluations of *HERE* using YCSB with defined degradation.

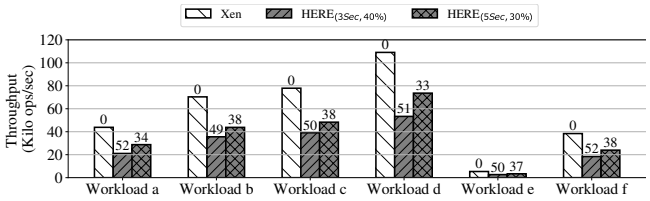

Figure 13: Evaluations of *HERE* using YCSB with defined degradation and T_{max}.

Table 6: YCSB and Sockperf evaluation configurations.

Acronym	Configuration
Xen	Xen without replication
$HERE_{(3Sec,0\%)}$	*HERE* with D = 0% and T_{max} = 3 seconds
$HERE_{(5Sec,0\%)}$	*HERE* with D = 0% and T_{max} = 5 seconds
$HERE_{(\infty,20\%)}$	*HERE* with D = 20% and T_{max} = ∞
$HERE_{(\infty,30\%)}$	*HERE* with D = 30% and T_{max} = ∞
$HERE_{(\infty,40\%)}$	*HERE* with D = 40% and T_{max} = ∞
$HERE_{(5sec,30\%)}$	*HERE* with D = 30% and T_{max} = 5 seconds
$HERE_{(3sec,40\%)}$	*HERE* with D = 40% and T_{max} = 3 seconds
$Remus_{3Sec}$	Remus with T = 3 seconds
$Remus_{5Sec}$	Remus with T = 5 seconds

Evaluations with SPEC CPU 2006. We selected 4 benchmarks from SPEC CPU 2006: `gcc`, `cactuBSSN`, `namd` and `lbm`. We then executed these benchmarks with the configurations presented in Table 6. Figs 14 15 16 highlight the obtained results. The observations are similar to those of YCSB: on Fig 14, we can observe that *HERE* leads to a smaller replication overhead than Remus for the same checkpointing period. Regarding fixed-degradation replication, we can note on Fig 15 that *HERE* manages to respect lower desired degradations better than higher ones, again due to VM scheduling effects. Finally, Fig 16 presents the evaluation with a defined degradation and T_{max}. As with YCSB, we can notice that the desired degradation prevails over T_{max} because with 3 and 5 seconds as period, the degradation is lower than 40 and 30% respectively. This is why the degradation is around 30-38% with $HERE_{(5Sec,30\%)}$ and 43 to 51% with $HERE_{(3Sec,40\%)}$ for all benchmarks.

Evaluations with Sockperf. We also evaluated networking latency caused by VM replication using the Sockperf benchmark. We set up Sockperf in "under-load" mode, where the VM replies to a percentage of incoming packets from a remote server. We used 3 different configurations of Sockperf: using 64-byte packets ("load *a*"), using 1400-byte packets ("load *b*"), and using 8900-byte packets ("load *c*")

Fig 17 presents the obtained latency with each workload (presented in log scale). On baseline Xen, we observe the expected behavior, where latency mostly scales with packet size. With VM replication, latency increases massively independent of packet size, with $Remus_{3Sec}$ and $Remus_{5Sec}$ taking 845 msec and 1332 msec per packet on average. This is caused by ASR approaches needing to buffer each packet onto the replica before it's released to the outside world, leading to network latency being mostly dependent on checkpoint frequency. *HERE*, thanks to its dynamic checkpoint control, manages to limit the network latency to lower numbers than that of Remus, at an average of 129 msec and 148 msec for $HERE_{(3Sec,40\%)}$ and $HERE_{(5Sec,30\%)}$ respectively.

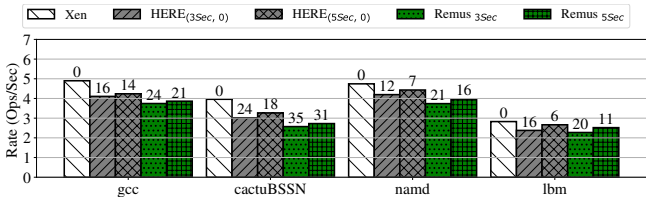

Figure 14: Evaluations of Remus and *HERE* using SPEC CPU 2006.

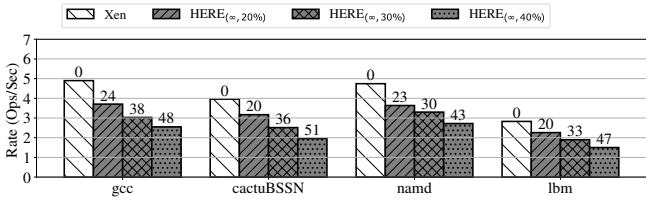

Figure 15: Evaluations of *HERE* using SPEC CPU 2006 with defined degradation.

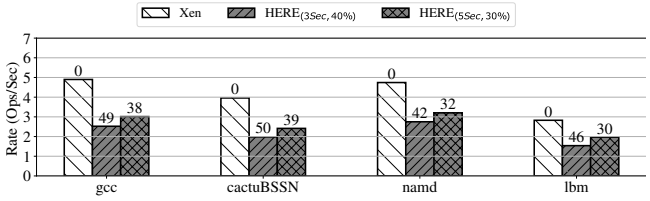

Figure 16: Evaluations of *HERE* using SPEC CPU 2006 with defined degradation and T_{max}.

8.7 Overhead of *HERE*

As we mentioned in Section 5.1, VM replication is multithreaded depending on the number of vCPUs. As these threads consume

extra resources, in this evaluation, we aim to evaluate the amount of CPU and memory resources consumed by *HERE*.

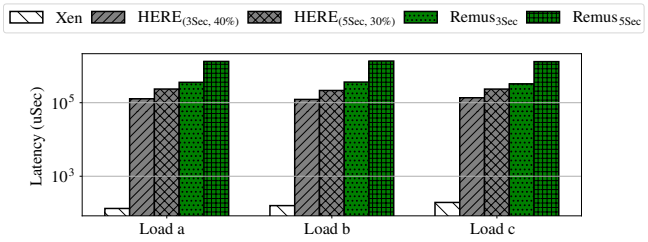

Figure 17: Evaluations of network latencies of Remus and *HERE* using Sockperf (log scale).

Note that this overhead mostly depends on the number of replication threads and not the checkpoint period, i.e. *HERE*'s overhead with a period of 1 second will be the same as with a period of 5 seconds. We measured the CPU and memory consumption of *HERE* when replicating a VM having 4 vCPUs and 16 GB of memory while running our microbenchmark, with a fixed replication period of 1 second. We observe that despite the multithreaded implementation, *HERE* only uses 62% CPU (with 100% CPU usage being one fully-loaded CPU core) and a memory consumption of 314 MB (counted as resident set size), which is comparable to that of existing solutions like Remus.

9 RELATED WORK

Automated analysis of hypervisors. Like other types of software, automated analysis tools [18, 39] are commonly used on hypervisors to detect and remedy potential security issues. VCC [18] uses the Z3 theorem prover to analyze the safety of hypervisor code using embedded annotations. Microkernels like seL4 [15], NOVA [40] and Phidias/Schism [26] use formal methods to prove their various security properties. The Linux kernel comes with numerous compile-time and runtime tools for detecting kernel bugs [42–45, 48]. Note that runtime analysis solutions can be combined with *HERE* as described in Section 6 to enhance the security of the hypervisor without inadvertently introducing DoS issues.

Fuzzing. Numerous works propose fuzzing of API surfaces to detect software vulnerabilities [13, 28, 41]. HYPER-CUBE [34] implements a custom guest OS for high-throughput hypervisor and virtual device fuzzing. Nyx [35] uses coverage-guided fuzzing on nested VMs to improve vulnerability detection. SlowFuzz and PerfFuzz [19, 30] focus on the particular class of algorithmic complexity vulnerabilities. Fuzzing can be combined with compile-time or runtime analysis for enhanced coverage [14, 27].

Hypervisor live updating. HyperFresh [8] uses nested virtualization to upgrade a running hypervisor without disrupting running VMs. Orthus [54] proposes a thin KVM API wrapper for quickly replacing the KVM hypervisor module. VM-PHU and Hy-FiX [32, 36] use fast soft-reboot techniques to replace a running hypervisor with minimal downtime. While these solutions support moving between different versions of the same hypervisor, they do not support moving between different hypervisors. In addition, fast reboot-based methods interrupt the VM by a non-negligible time period of several seconds while the hypervisor is rebooted.

Heterogeneous hypervisor support. These solutions propose supporting running VMs on multiple different hypervisors, and therefore are the most closely related to *HERE*. HyperTP [24] combines fast soft-reboot and heterogeneous VM migration into a solution called *hypervisor transplant* to allow switching between different hypervisors; however, it does not propose a fault-tolerant replication solution and therefore can only be used once a vulnerability is already known. The techniques used by HyperTP are similar to that of Vagrant [21], which implements a solution that bridges different hypervisors together in one VM migration framework, but does not explicitly address security problems.

Live VM replication. Remus [4] proposes a checkpointing and I/O-buffering algorithm for asynchronous VM replication. COLO [9] executes the active and passive replicas in lockstep, comparing their I/O outcomes to maintain consistency between the different replicas. Adaptive Remus [5] modifies Remus to adapt the checkpointing frequency based on network activity. rRVM [12] and Gannet [50] make use of secondary-side I/O buffering to reduce replication overhead. PLOVER [49] uses a paravirtualized approach for applying state machine replication techniques to VM replication. To our best knowledge, *HERE* is the first solution that proposes VM replication across multiple different hypervisors.

10 CONCLUSION

In this article, we introduced *HERE*, a solution for enhancing the security and availability of hypervisor infrastructure. We introduced the unique protective strategy of heterogeneous replication to protect a VM from multiple classes of failures, and showed its applicability in light of our investigation on hypervisor vulnerabilities. We proposed optimizations of the replication process to reduce replication overhead; furthermore, we implemented additional logic to control the performance impact of VM replication on running applications. We discussed various concerns related to securely implementing *HERE* in virtualized systems, including showing how *HERE* combined with exploit mitigation can safeguard the system against multiple classes of vulnerabilities without sacrificing availability or increasing its attack surface. Finally, we evaluated the performance and adaptability of *HERE* under multiple different scenarios, and showed how *HERE* could adapt its performance profile following the user's predetermined constraints.

ACKNOWLEDGEMENTS

This work was supported by the French *Agence nationale de la recherche* under the projects ANR WalkIn (ANR-20-CE25-0005), ANR PicNic (ANR-20-CE25-0013) and LabEx CIMI (11-LABX-0040). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). We would like to thank our shepherd Redha Gouicem and the anonymous reviewers for their helpful feedbacks.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40.

[2] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2012. Revisiting hardware-assisted page walks for virtualized systems. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 476–487. https://doi.org/10.1109/ISCA.2012.6237041

[3] Harold Booth, Doug Rike, Gregory A Witte, et al. 2013. The national vulnerability database (nvd): Overview. *ITL Bulletin, National Institute of Standards and Technology, Gaithersburg, MD* (2013).

[4] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. 2008. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX symposium on networked systems design and implementation*. San Francisco, 161–174.

[5] Marcelo Pereira Da Silva, Rafael Rodrigues Obelheiro, and Guilherme Piegas Koslovski. 2017. Adaptive Remus: adaptive checkpointing for Xen-based virtual machine replication. *International Journal of Parallel, Emergent and Distributed Systems* 32, 4 (2017), 348–367.

[6] Theo de Raadt. 2015. pledge() - a new mitigation mechanism. https://www.openbsd.org/papers/hackfest2015-pledge/mgp00001.html.

[7] Alex Depoutovitch and Michael Stumm. 2010. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 181–194. https://doi.org/10.1145/1755913.1755933

[8] Spoorti Doddamani, Piush Sinha, Hui Lu, Tsu-Hsiang K Cheng, Hardik H Bagdi, and Kartik Gopalan. 2019. Fast and live hypervisor replacement. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 45–58.

[9] YaoZu Dong, Wei Ye, YunHong Jiang, Ian Pratt, ShiQing Ma, Jian Li, and HaiBing Guan. 2013. Colo: Coarse-grained lock-stepping virtual machines for non-stop service. In *Proceedings of the 4th annual Symposium on Cloud Computing*. 1–16.

[10] A. Gkortzis, S. Rizou, and D. Spinellis. 2016. An Empirical Analysis of Vulnerabilities in Virtualization Technologies. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 533–538.

[11] Google Inc. [n. d.]. MiraclePtr<T> One Pager. https://docs.google.com/document/d/1pnnOAIz_DMWDI4oIOFoMAqLnf_MZ2GsrJNb_dbQ3ZBg/edit?usp=sharing.

[12] Muyang He, Shaoning Pang, Denis Lavrov, Ding Lu, Yuan Zhang, and Abdolhossein Sarrafzadeh. 2016. Reverse replication of virtual machines (rrvm) for low latency and high availability services. In *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 118–127.

[13] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. 2017. Vdf: Targeted evolutionary fuzz testing of virtual devices. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 3–25.

[14] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. 2020. FuZZan: Efficient sanitizer metadata design for fuzzing. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 249–263.

[15] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 207–220.

[16] K. Kourai and S. Chiba. 2011. Fast Software Rejuvenation of Virtual Machine Monitors. *IEEE Transactions on Dependable and Secure Computing* 8, 6 (2011), 839–851.

[17] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 276–291.

[18] Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM 2009: Formal Methods*, Ana Cavalcanti and Dennis R. Dams (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 806–809.

[19] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 254–265.

[20] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. 2019. {PAC} it up: Towards Pointer Integrity using {ARM} Pointer Authentication. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 177–194.

[21] Pengcheng Liu, Ziye Yang, Xiang Song, Yixun Zhou, Haibo Chen, and Binyu Zang. 2008. Heterogeneous live migration of virtual machines. In *In International Workshop on Virtualization Technology (IWVT'08)*.

[22] Zeyu Mi, Dingji Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2020. (Mostly) Exitless VM Protection from Untrusted Hypervisor through Disaggregated Nested Virtualization. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1695–1712. https://www.usenix.org/conference/usenixsecurity20/presentation/mi

[23] Onur Mutlu and Jeremie S. Kim. 2020. RowHammer: A Retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 8 (2020), 1555–1571. https://doi.org/10.1109/TCAD.2019.2915318

[24] Tu Dinh Ngoc, Boris Teabe, Alain Tchana, Gilles Muller, and Daniel Hagimont. 2021. Mitigating Vulnerability Windows with Hypervisor Transplant. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 162–177. https://doi.org/10.1145/3447786.3456235

[25] Jason Nikolai and Yong Wang. 2014. Hypervisor-based cloud intrusion detection system. In *2014 International Conference on Computing, Networking and Communications (ICNC)*. 989–993. https://doi.org/10.1109/ICCNC.2014.6785472

[26] Jan Nordholz. 2017. Design and provability of a statically configurable hypervisor. *Technische Universität Berlin* (2017).

[27] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. Parmesan: Sanitizer-guided greybox fuzzing. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2289–2306.

[28] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. 2021. V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*.

[29] Diego Perez-Botero, Jakub Szefer, and Ruby B Lee. 2013. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 international workshop on Security in cloud computing*. 3–10.

[30] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2155–2168.

[31] Sundaresan Rajasekaran, Harpreet Singh Chawla, Zhen Ni, Neel Shah, Emery Berger, and Timothy Wood. 2018. CRIMES: Using Evidence to Secure the Cloud. In *Proceedings of the 19th International Middleware Conference* (Rennes, France) *(Middleware '18)*. Association for Computing Machinery, New York, NY, USA, 40–52. https://doi.org/10.1145/3274808.3274812

[32] Mark Russinovich, Naga Govindaraju, Melur Raghuraman, David Hepkin, Jamie Schwartz, and Arun Kishan. 2021. Virtual Machine Preserving Host Updates for Zero Day Patching in Public Cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 114–129. https://doi.org/10.1145/3447786.3456232

[33] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software diversity: state of the art and perspectives. *STTT* 14, 5 (2012), 477–495. https://doi.org/10.1007/s10009-012-0253-y

[34] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2020. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing.. In *NDSS*.

[35] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

[36] Andrea Segalini, Dino Lopez-Pacheco, Guillaume Urvoy-Keller, Fabien Hermenier, and Quentin Jaquemart. 2021. Hy-FiX: Fast In-place Upgrades of KVM Hypervisors. *IEEE Transactions on Cloud Computing* (2021).

[37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. Addresssanitizer: A fast address sanity checker. In *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*. 309–318.

[38] Le Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. 2017. Deconstructing Xen. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/deconstructing-xen/

[39] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1275–1295.

[40] Udo Steinberg and Bernhard Kauer. 2010. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*. 209–222.

[41] syzkaller authors. [n. d.]. syzkaller - kernel fuzzer. https://github.com/google/syzkaller.

[42] The kernel development community. [n. d.]. The Kernel Address Sanitizer (KASAN). https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[43] The kernel development community. [n. d.]. The Kernel Concurrency Sanitizer (KCSAN). https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.

[44] The kernel development community. [n. d.]. Kernel Electric-Fence (KFENCE). https://www.kernel.org/doc/html/latest/dev-tools/kfence.html.

[45] The kernel development community. [n. d.]. Kernel Memory Leak Detector. https://www.kernel.org/doc/html/latest/dev-tools/kmemleak.html.

[46] The kernel development community. [n. d.]. Seccomp BPF (SECure COMputing with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[47] The kernel development community. [n. d.]. Security bugs - The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/admin-guide/security-bugs.html.

[48] The kernel development community. [n. d.]. The Undefined Behavior Sanitizer - UBSAN. https://www.kernel.org/doc/html/latest/dev-tools/ubsan.html.

[49] Cheng Wang, Xusheng Chen, Weiwei Jia, Boxuan Li, Haoran Qiu, Shixiong Zhao, and Heming Cui. 2018. {PLOVER}: Fast, Multi-core Scalable Virtual Machine Fault-tolerance. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 483–489.

[50] Cheng Wang, Xusheng Chen, Zixu Wang, Youwei Zhu, and Heming Cui. 2017. A fast, general storage replication protocol for active-active virtual machine fault tolerance. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 151–160.

[51] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. 2015. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 20–37.

[52] Idris Winarno, Takeshi Okamoto, Yoshikazu Hata, and Yoshiteru Ishida. 2016. Increasing the Diversity of Resilient Server Using Multiple Virtualization Engines. *Procedia Computer Science* 96 (2016), 1701–1709.

[53] Xin Xu and H. Howie Huang. 2015. DualVisor: Redundant Hypervisor Execution for Achieving Hardware Error Resilience in Datacenters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 485–494. https://doi.org/10.1109/CCGrid.2015.30

[54] Xiantao Zhang, Xiao Zheng, Zhi Wang, Qi Li, Junkang Fu, Yang Zhang, and Yibin Shen. 2019. Fast and scalable VMM live upgrade in large cloud infrastructure. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 93–105.

[55] Diyu Zhou and Yuval Tamir. 2018. Fast hypervisor recovery without reboot. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 115–126.