BlueGuard: Accelerated Host and Guest Introspection Using DPUs

Meni Orenbach *NVIDIA*
Rami Ailabouni *NVIDIA*
Nael Masalha *NVIDIA*
Thanh Nguyen *Unaffiliated* *
Ahmad Saleh *NVIDIA*
Frank Block *NVIDIA*
Fritz Alder *NVIDIA*
Ofir Arkin *NVIDIA*
Ahmad Atamli *NVIDIA*

Abstract

Virtual Machine Introspection (VMI) is an essential technique for monitoring the runtime state of a virtual machine. VMI systems are widely used by major cloud providers as they enable a range of applications, such as malware detection. Unfortunately, existing VMI systems suffer from several shortcomings: they either compete with the introspected VMs for shared CPU resources or report poor performance. Further, they cannot introspect hypervisors or bare metal machines.

We propose BlueGuard, a system that leverages the physically isolated Data Processing Unit (DPU) commonly found on data center servers to efficiently run full system introspection by both host and guest introspection (HGI).

BlueGuard facilitates the creation of hardware-accelerated HGI applications and frees the CPU while providing performance isolation. As a beneficial side effect, BlueGuard is capable of introspecting even bare metal servers that are usually out of scope for VMI systems. Furthermore, BlueGuard abstracts the DPU accelerators and provides kernel bypassing, non-blocking memory access, and user-level threading to achieve µs-scale introspection latency. Finally, we introduce *delta introspection* to accelerate the detection of state changes with BlueGuard and demonstrate the ability to isolate infected machines on a network layer.

We implement and extensively evaluate BlueGuard on an NVIDIA BlueField-2 DPU. Our system achieves a 4.3× detection speedup compared to prior work and is capable of monitoring tens of VMs concurrently without hindering the host performance.

1 Introduction

Virtual machine introspection (VMI) is an important tool for identifying advanced threats in VMs, such as rootkits [12, 16, 17, 23, 40, 64, 65, 67, 69], and is commonly deployed in modern data centers [18, 43]. VMI helps detection by bridging the semantic gap [25]: reconstructing high-level information from raw memory images.

VMI was demonstrated in its benefits to detect *stealthy malware* [66], a special type of malware that aims to evade detection employed by traditional security countermeasures. Specifically, stealthy malware can be as sophisticated as to hide its files and processes, disable the detection tools, and use polymorphism to change its appearance on each execution.

Previous work on VMI systems includes different design options. First, Host VMI systems (Fig 1(a)) reside within the hypervisor or the VM [16, 77, 79], and have scalability and resiliency issues [69]. Further, these systems have inadequate isolation and therefore do not allow for a hypervisor or bare metal introspection [39].

Second, Out-of-band (OOB) VMI systems (Fig 1(b)) run on an isolated device [15, 33, 39, 41, 46, 48, 66, 80], and acquire data using Direct Memory Access (DMA) or remote DMA (RDMA). OOB VMI is considered more secure than host VMI due to devices being *physically isolated*. Further, OOB VMI can introspect a full system: *host and guest introspection* (HGI). Notably, prior work focused on introspecting either hypervisors or VMs, overlooking detecting potential threats.

Modern data centers present both new requirements and opportunities for HGI. On the requirements side, server processors can concurrently host dozens of VMs [44], and stealthy malware can conceal itself within milliseconds, limiting the effectiveness of slower HGI systems.

On the opportunity side, data centers now feature increasingly complex servers that host not only conventional resources like CPUs and memory but also accelerators such as Data Processing Units (DPUs) [21, 45, 52]. Note, we use the term DPU interchangeably with SmartNIC. The primary function of DPUs is to offload complex networking tasks like firewalls or packet encryption from the CPU to the DPUs.

To bridge the performance gap of conventional VMI systems in modern data centers, we propose *BlueGuard*, an HGI system that builds on top of a DPU to offload the computation-intensive introspection of tens of VMs concurrently while being physically isolated from the inspected host. BlueGuard

*Work done while at NVIDIA

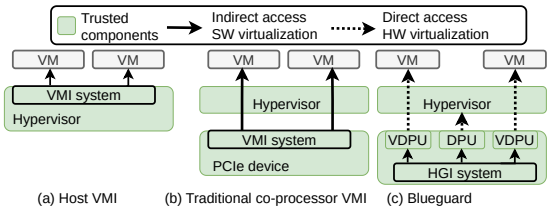


Figure 1: Overview of existing VMI systems and BlueGuard.

is inspired by advancements by NVIDIA for malware detection [53] and executes introspection applications as illustrated in Fig 1(c). BlueGuard utilizes DPUs' SR-IOV capabilities to create a virtual DPU (vDPU) to scalably support HGI.

BlueGuard abstracts the complexity of DPU programming without sacrificing the introspection bandwidth with the following design choices.

DPU tailored introspection. BlueGuard builds on the idea of libOSs used in CPUs for µs-scale [4, 60, 78] for HGI systems. This includes user-level threading, memory management, and I/O management through the vDPUs. Specifically, each introspection application is connected to a single monitored VM or the host through a dedicated vDPU. BlueGuard utilizes the one-sided RDMA interface to program local DMA engines via the vDPU hardware-managed queues enabling kernel-bypass in the DPU. Moreover, BlueGuard reuses known RDMA performance optimizations when programming the DPU's accelerators. This enables BlueGuard to reduce introspection latency by tailoring it to DPUs unique hardware design (§6.3).

Offloading introspection. Introspection applications rely on computing resources to analyze memory for malware, which can create a bottleneck when introspecting tens of VMs with the DPUs resource-constraint general-purpose cores. We demonstrate improved performance when using DPUs hardware for accelerating and offloading these actions from the DPUs' general-purpose cores. Specifically, we show orders-of-magnitude performance improvement for real introspection applications similar to those used by Google's virtual machine threat detection [18]: verifying kernel modules integrity and scanning for malware with YARA rules [74] (§6.5).

Delta Introspection. Not all resource-intensive tasks can be easily accelerated by DPUs. One such example is pointer chasing, which plays a vital role for several VMI applications. Yet, we observe that the only input for VMI applications is the VMs' acquired memory, and the output changes iff the memory changes. Thus, we introduce *delta introspection*, where BlueGuard traces the set of memory accesses and their content, so that upon subsequent invocations it can batch-send the same requests and compare the acquired values. This technique detects changes (deltas) to the VMI application's input data efficiently, without re-running the VMI application.

Ease of introspection. Importantly, BlueGuard allows introspection applications to utilize these accelerators without arduous code changes. Instead, BlueGuard's design provides a plugin system that allows for easy implementation of introspection applications without extensive modifications (§5).

Malware isolation. BlueGuard supports different policies for continuous malware scanning. Upon detection, it can trigger VLAN isolation, segmentation of the infected server or guest in case the hypervisor or tenant are compromised respectively. This isolation is governed by predefined policies controlled by the DPU's programmable switch that dictates different levels of access and containment measures.

Results. We implement a prototype of BlueGuard on top of the NVIDIA BlueField-2 DPU and evaluate the system's performance and scalability using microbenchmarks and real introspection applications such as scanning active processes, active kernel modules, and duplicated credentials checking. We show that BlueGuard enables a 4.3× improvement in detection speed compared to prior HGI systems, while also scaling to monitor up to 64 VMs concurrently. Furthermore, with DPUs' hardware accelerators, BlueGuard accelerates common VMI applications by up to two orders of magnitude.

Further, we show that BlueGuard detects different malware, including stealthy malware which hides its presence in a few milliseconds successfully. Finally, we evaluate BlueGuard's performance impact on introspected VMs by running a bandwidth-demanding introspection application concurrently with SPEC CPU2017 [6] and IOZone [7]. We do not observe a noticeable performance impact indicating that BlueGuard maintains performance isolation.

BlueGuard makes the following contributions:

- Design of an efficient and scalable OOB introspection system using DPUs with SR-IOV, N:M threading, delta introspection, and DPU hardware accelerators.
- Malware detection and effective network containment via VLAN isolation.
- Prototype implementation of BlueGuard on commodity DPUs and evaluation with microbenchmarks and real-world VMI applications.

2 Background

In the following, we introduce machine introspection, common introspection applications, and DPUs.

2.1 Machine introspection

Machine introspection has mostly been studied for the past two decades under the term VMI to gain valuable information on VMs by accessing snapshots of raw memory. Unsurprisingly, VMI was largely considered for security workloads, such as intrusion detection [17], malware analysis [34], and memory forensics [20, 75].

VMI systems execute in isolated environments, most often by the hypervisor [77] or in dedicated VMs [79]. VMI systems bridge the semantic gap and enable seamless execution of VMI applications that periodically acquire guest memory snapshots for security analysis tasks, e.g., reconstructing the process list to identify whether an unknown process exists. Recent systems allow the introspection of the hypervisor itself as well as bare metal machines [29, 39, 48, 66].

2.2 Common VMI applications

In this work, we use the terms VMI application and introspection application interchangeably, as the most common of these applications are originating from the area of VMI-based systems. Since VMI applications are observing the introspected operating system they can access, the applications themselves are not the limiting factor when differentiating between a VMI-, and an HGI-based system, and we thus see any VMI application as a valid HGI application. VMI applications used by prior work and industry tools observably follow a common flow illustrated in Figure 2. In a high-level overview, VMI applications access specific addresses in the introspected's memory based on known symbol offsets to process the acquired data and compute the next set of addresses to access until finally extracting the desired artifact. We find the following common high-level patterns in VMI applications [18, 39, 75, 79]: 1) read memory for a pointer, and read its content; 2) Scan the memory for known malicious patterns, e.g., via YARA rules [74]; 3) Compare the read content to a reference value. In the following, we list a number of common VMI applications that we also support in BlueGuard:

pslist. Lists all processes metadata by traversing over the process list with the list head being the init_task in Linux. The metadata captured includes the process's state, virtual address, name, and associated identifiers: self, parent, user, and group. For simplicity, we use pslist as a running example.

check_creds. Traverses all processes' credentials by traversing the process list and dereferencing the internal cred pointer. Captures metadata on process identifiers that shares the same credential structure.

modules. Lists the metadata for all kernel modules by traversing over the modules list with the list head being pointed to by the symbol modules_addr in Linux. The metadata captured includes the modules' virtual address, name, and size.

modules_integrity. Similar to modules, it iterates over all kernel modules but it captures all modules' virtual memory areas (VMAs) and computes a cryptographic hash per page. Upon every invocation, it validates that the cryptographic hash matches a pre-computed hash, which demonstrates that a kernel module was not compromised by an adversary.

regex_scan. Iterates over all processes, and for each process captures its VMAs and scans them for the given regex.

We demonstrate how BlueGuard utilizes these applications for malware detection in §6.1.

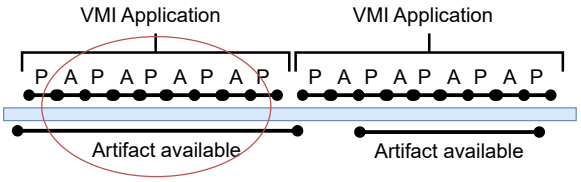


Figure 2: VMI application flow. Artifacts are detected/misdetected. **P**rocessing steps capture data and compute the next accessed address. **A**ccesses fetch the data.

2.3 Data Processing Units (DPUs)

BlueGuard leverages DPUs [24, 42, 45, 47, 51], which extend traditional NICs by offloading various networking tasks from the main CPU, enhancing performance and efficiency in data centers and high-performance computing environments [28, 35, 61]. The reasons include the decline of Moore's law that necessitates better use of CPU cycles [68], stronger security due to physical isolation, and improved performance isolation.

DPUs feature programmable cores that allow for custom processing of network traffic, management cores for overseeing operations, and high-speed memory for quick data storage and processing. Additionally, DPUs are equipped with hardware accelerators for tasks like encryption/decryption, compression, and deep packet inspection, significantly improving overall system performance by offloading these tasks from the DPUs general-purpose cores. DPUs manage both incoming and outgoing network traffic, performing tasks such as load balancing, and routing directly on the embedded NIC.

DPUs that support one-sided RDMA verbs, such as RDMA read and RDMA write, enable direct memory access between a local and a remote node without involving the remote CPU. This capability allows a compute server to directly read from or write to the memory of a remote server by specifying the memory address to be accessed. By bypassing the remote CPU, one-sided RDMA verbs significantly reduce latency and increase bandwidth, making them highly efficient for tasks that require rapid data transfer with low overhead. BlueGuard leverages the one-sided RDMA verbs interface, yet, for local DPU accelerators, e.g., local DMA to read the host's memory into the DPU's local memory. Similarly, DPUs can perform inline acceleration on data read from the host into the DPU's memory, including encryption, decryption, compression, hashing, and regular expression matching.

Finally, DPUs with embedded switch capabilities offer advanced network management directly in hardware. This includes traffic routing, load balancing, and network segmentation. BlueGuard utilizes this to enforce network isolation policies at wire speed when detecting malware.

3 Motivation and objectives

Hypervisor-based virtualization creates a substantial trusted computing base (TCB) consisting of millions of lines of code, which often leads to the discovery of common vulnerabilities and exposures (CVEs). A similar situation exists for bare metal systems. Malware scanning within hypervisors or bare metal systems must be offloaded to custom devices [39, 48]. This limits existing host-based VMI systems and forces the use of OOB introspection within HGI systems.

The downside is that, unlike traditional VMI, OOB introspection cannot affect the introspected VMs, e.g., cannot pause them before accessing the memory, which creates a smearing effect. In turn, it forces HGI systems for efficient memory acquisition. For example, for pslist, malicious processes can be available only for a limited time as they terminate after performing a designated exploit. Notably, detecting the process depends on the processing and memory access latency of the pslist VMI application.

Figure 2 shows two cases where the artifact is only fully available for the left, circled, instance. In the right instance, the artifact becomes only available after the VMI application has already started processing the memory, and the malicious process might be missed. This is the *data smearing* problem and is a known limitation of asynchronous OOB systems. Analysts can reduce the probability of smearing by invoking the VMI application multiple times, e.g., at specific intervals or continuously. In Appendix A we present a statistical model to quantify the detection rate for a VMI application. At a high level, a 100% detection rate with continuous introspection requires the artifact of interest to be in memory at least twice as long as the latency of a full introspection execution.

We propose delta introspection to detect changes much faster than invoking the full VMI application. Its advantage is 100% detection rate in a shorter period, by combining the VMI application's latency with the significantly lower delta introspection latency ($2 \cdot delta\_latency + vmi\_app\_latency$).

Similar to delta introspection, it is critical to accelerate the introspection to increase the detection probability without rewriting the VMI application. Further, public clouds can host tens of VMs that execute concurrently, which necessitates a scalable HGI system.

3.1 Why DPUs?

DPUs improve security over host-based VMI as the interface shared with the host is minimal: reading memory content. Thus, CPU vulnerabilities such as Spectre and Meltdown [30, 38] do not leak DPU-related states.

Moreover, modern processors use IOMMU to protect against malicious devices using DMA to access CPU memory. DPUs are inherently trusted to send and receive data over the network and read and write the content to the hosts' memory. Yet, generic accelerators are likely to be restricted from using DMA through the IOMMU to limit exposure to potential DMA attacks. Note, a stealthy malware attempting to tamper with the IOMMU to avoid BlueGuard 's detection will also tamper with the network access. This effectively limits malware from tampering with BlueGuard and continue leaking secrets or communicate with a control server over the network.

Finally, in addition to the physical isolation, BlueGuard gains additional benefits from residing on the DPU, as the DPU is commonly utilized for networking tasks, which allows an HGI system to autonomously react after malware is detected. For example, BlueGuard could completely isolate an infected VM from the original network, place it in a VLAN dedicated for potentially malicious tenants, and provide notification to an external server all at the same time.

3.2 Advantage over remote introspection

Remote device. Recently, RDMI [39] showcased up to 2Gb/second network bandwidth for introspection as an HGI system by utilizing an RDMA-based approach and transmitting data from a bare metal machine to a programmable switch that performs the introspection. In contrast to prior, host VMI-based systems such as LibVMI [77], RDMI reports better performance even though it only focuses on bare metal servers and does not support the introspection of virtual machines. Crucially, however, RDMI relies on unencrypted RDMA traffic from the introspectee's memory to RDMI. This may not be acceptable for security-conscious users who do not wish to expose the HGI memory reads over untrusted networks.

Further, DPUs contain hardware accelerators such as a regex engine for pattern matching and a hashing unit for computing digests. These accelerators can greatly speed up common introspection tasks. For instance, the regex engine can scan for malware patterns, while the hashing unit can speed up the comparison of known malware digests.

Remote host. Remote introspection with an external CPU acquiring memory through RDMA [31] offers a few advantages: programming flexibility, performance, and improved ecosystem compatibility. Running VMI applications in a remote CPU is close to the DPU model used by BlueGuard. Yet, without sending memory content over the network. Notably, DPUs are prevalent in data centers [21, 45, 52], and HGI offloading to DPUs can reduce the data center's total cost of ownership (TCO) compared to using remote CPUs for HGI. Specifically, DPUs consume less power compared to CPUs, and the saved CPU cores can be rented to tenants.

3.3 System model

We consider a platform with a multi-core CPU under the management of a cloud service provider (CSP). Multiple VMs execute on the same platform and are managed by a hypervisor. Both the VMs as well as the hypervisor are the target

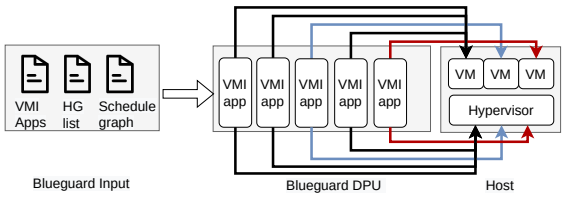


Figure 3: BlueGuard overview. Arrows depict which host or guest each VMI application is monitoring.

of introspection. If no VMs are present, the whole bare metal machine as a whole is the sole target of introspection. The platform contains a DPU connected through PCIe to the CPU. BlueGuard executes on the DPU and runs introspection applications to detect malware on the system.These introspection applications monitor live kernel and user application states without modifying or suspending the guest system.

BlueGuard targets the system organization shown in Figure 3. Using BlueGuard, security analysts can deploy VMI applications such as pslist for malware detection together with a security policy that contains the requested HG list: which VMs and hypervisor to introspect with the deployed VMI applications and the introspection schedule.

3.4 Threat model

The adversary is malware in the host that may possess root privileges that aim to compromise the confidentiality and integrity of services executed in the introspected system. We assume the hardware and firmware in both, the host's CPU and DPU, are trusted, and the hypervisor and any running VM boot securely to enable DPU access to their memory. Both the DPU itself as well as software running on the DPU are inside the trusted computing base and cannot be compromised by the adversary.

After boot and BlueGuard initialization, the adversary may infect a target during runtime by launching various software attacks, e.g., by exploiting a buffer overflow vulnerability [70] or by taking over the host from an infected VM. However, a core assumption is the visibility of any malicious modification in the main memory. Thus, we assume that the adversary can be detected by applications that are introspecting the memory, and attackers avoiding memory modifications are out of scope.

Next, we distinguish between the threat models for the two deployment variations of BlueGuard: Bare metal introspection and VM introspection.

Bare metal threat model. An adversary compromising the bare metal machine and gaining kernel-level access may tamper with the page table or kernel data structures to avoid introspection and malware detection [3, 26]. Specifically, by forcing the VMI systems to introspect a memory region that appears benign while malware executes elsewhere in memory. These address translation redirection attacks are a known limitation shared by all OOB VMI systems [26, 33, 48, 66] and is out of scope. **VM-specific threat model.** In the VM-specific deployment, a compromised hypervisor can alter IOMMU translations to avoid VM introspection by BlueGuard. This attack is a form of the previously mentioned translation redirection attacks and is equally out of the scope due to the same set of limitations. Finally, DoS attacks are out of scope for both types of deployment.

3.5 Objectives and non-goals

Our goal with BlueGuard is to provide a configurable system with improved introspection efficiency compared to existing OOB HGI systems. Specifically, security analysts can opt to continuously invoke introspection applications for an optimal detection rate, or employ heuristics to invoke specific applications for certain VMs on the system. Yet, proposing new VMI applications that are more proficient at detecting malware, or a specific invocation heuristic to improve the detection accuracy and DPU resource utilization is out of the scope of this paper. Instead, in § 6 we evaluate VMI applications that were used by prior work [39, 79] and a continuous invocation heuristic for optimal detection rate.

4 BlueGuard design

We first explain the high-level design of BlueGuard and then describe the main components in detail.

Figure 4 shows the overall architecture of BlueGuard. BlueGuard depends on DPUs with support for virtualization (SR-IOV), such that the DPU can present itself as a virtual DPU (vDPU) that connects to a VM or the hypervisor. BlueGuard implements a mechanism to transfer live memory content from the VMs to the VMI applications via the vDPU's queues abstracted as *VMI contexts*. Each VMI context contains the VMI application's metadata, similar to a process in a traditional OS, which includes vDPU hardware queues: send, receive, and completion queues for the DMA, regex, and hashing accelerators, the physical memory regions, an intermediate symbol table (IST) for the kernel, and internal caches.

VMI applications do not use VMI contexts directly. Instead, BlueGuard's interface abstracts memory accesses to either a DPU *local address space (LAS)* or the introspected's *remote address space(RAS)*.

On the first boot of the VM/hypervisor, BlueGuard creates an underlying VMI context for it, and communicates with the introspected to set up the control plane to enable access to its memory: a set of send/receive queues and completion queue per acceleration engine and memory registration for its RAS and a DPU LAS. Next, VMI applications can submit either memory access or inline computation requests to buffers in LAS or RAS. The VMI applications are executed by BlueGuard's scheduler.

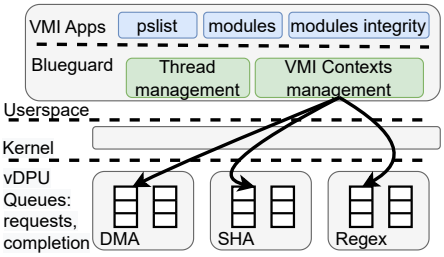


Figure 4: BlueGuard manages VMI applications in userspace.

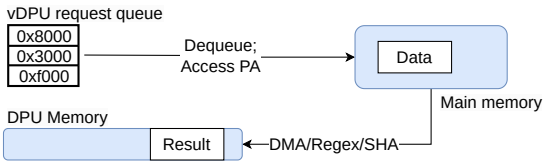


Figure 5: BlueGuard data plane flow with inline computation. vDPU queues are simplified to only contain the VM PAs.

4.1 Memory management

In BlueGuard, the DPU's general-purpose cores execute the control plane, and the embedded NIC zero-copies data from the host memory to the DPU's local memory. Under the hood, BlueGuard uses the traditional RDMA interface to access the NIC's queues. Yet, unlike traditional RDMA the data is not transferred over the network and hence sensitive VM data is not exposed to adversaries that may snoop on the network. Moreover, BlueGuard use of DPUs with SR-IOV support sidesteps the need to virtualize access to the DMA engine, enabling kernel-bypass [27].

Initialization. BlueGuard relies on a PCIe feature called Single Root Input/Output Virtualization (SR-IOV) that employs so-called virtual functions (VFs) to virtualize a PCIe device [59]. Each VF is directly assignable to a VM or the hypervisor, granting access to unique DPU hardware resources, thereby acting as a vDPU. The VFs are used by BlueGuard's initialization to set up DPU access to the VM as described next.

To initiate memory access, BlueGuard first registers two memory regions: one in the target: either the VM/hypervisor for RAS access or the DPU for LAS access, and the other in the DPU. Then, BlueGuard creates and exchanges control data to set up the queue pairs to a ready state for all the acceleration engines: DMA, regex, and hashing. Note, BlueGuard intends this process to be performed by agents in the DPU and the introspected VM/hypervisor, which are both trusted at initialization time while exchanging the data over a secure channel, e.g., a UNIX socket guarded with TLS. This secure channel is closed after the data exchange is finished and before any external client can connect. Specifically, we assume the introspection begins in a valid state before it can be compromised by an adversary.

Protect VM control structures. After the initialization completes, the agent terminates but does not free the resources allocated. Thus, access by the DPU is maintained throughout the VM/hypervisor lifetime. Later, when the boot completes, an adversary may gain root privileges and attempt to limit or tamper with the control structures, e.g., deregister the registered RAS memory region, or modify it to limit introspection to memory regions that fake benign activity. However, such attempts would render the existing RAS object invalid and would thus be detected by BlueGuard. Note, DoS attacks are out of scope for BlueGuard (§3.3).

RAS access. DPUs contain address translation capabilities to access virtual addresses. However, prior work observed that limited NIC resources can limit scalability when accessing large virtual memory regions [73]. Furthermore, our threat model assumes the VM or hypervisor may be compromised after BlueGuard's initialization, which makes it challenging to validate in runtime the security posture of newly registered memory regions. Instead, we use the `pa-mr` verb that enables registering the entire physical address space [39, 55, 73]. However, this design choice forces BlueGuard to perform software translations of RAS memory. Specifically, virtual address (VA) to physical address (PA) and guest VA (GVA) to guest PA (GPA) translations for the hypervisor and VMs, respectively (§4.3).

Once the control plane is set up, BlueGuard internally invokes the acceleration engines by sending a work request element via the queue pair with the target address in either RAS or LAS along with the size for each access and the target local memory to store the result (Figure 5). After the request completes, the DPU places a work completion element in its completion queue to asynchronously notify that the data fetch was completed successfully, or indicate an error if occurred.

Shared address space. BlueGuard enables the sharing of common information, such as the GVA-GPA translation cache of introspected VMs/hypervisor across different VMI applications as they all share the same address space. This is useful when multiple VMI applications are configured to monitor the same VM/hypervisor.

4.2 BlueGuard scheduler

BlueGuard builds on DPUs, which have limited computing resources. Therefore, instead of continuously polling the completion queue to validate memory access requests completed successfully, BlueGuard performs a single polling operation in a non-blocking manner. Specifically, if the poll indicates the access was completed successfully, the VMI application can continue to execute. Otherwise, BlueGuard context switches to a different VMI application. Once the VMI application

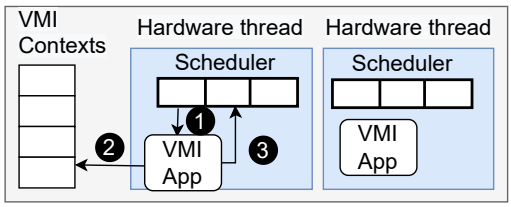


Figure 6: BlueGuard threading model. ❶ Schedule VMI application. ❷ Post read request to VM's memory. ❸ Yield to the scheduler while the request is being processed.

is scheduled back, BlueGuard checks whether the memory access is completed or if another context switch is required.

Threading model. BlueGuard uses an N:M threading model with cooperative multitasking to avoid costly hardware thread context switches during non-blocking memory accesses. Figure 6 illustrates how M VMI application threads are multiplexed across N OS threads. In BlueGuard, each thread executes a userspace scheduler that chooses a VMI application registered to a hardware thread to be executed.

Once a VMI application is scheduled to run it executes uninterruptedly until posting a request on the VMI context's queue and then yields its execution back to the scheduler. The scheduler checks whether another VMI application thread can wake up due to an expired timeout or the arrival of a completion element. If no VMI application can be executed, the scheduler backs off to save the DPU compute resources. We evaluate this design decision in § 6.3.

4.3 Address translation

Virtual memory translation in VMs involves nested page tables that translate GVA to GPA and GPA to Host PA (HPA).

IOMMU considerations. BlueGuard sets the IOMMU to passthrough mode to prevent the hypervisor from manipulating with introspection through DMA remapping when it is being introspected. Thus hypervisor and bare metal introspection use the PA directly. Yet, for VM introspection, BlueGuard uses PCIe passthrough configuration, where the IOMMU translates GPAs to HPAs. While IOMMU configuration and address remapping attacks are possible by the hypervisor we currently exclude them from BlueGuard's threat model similar to prior HGI systems. We discuss mitigations and their performance impact in §7.

Software address translation. BlueGuard detects the root of the VM's/hypervisor's kernel page table. Next, for each VA, BlueGuard traverses the page table with DMA accesses and uses the extracted data to compute the PA with a software-based page table walk. Our prototype currently targets the x86 architecture with four-level page tables. Thus, BlueGuard sends four DMA requests to translate a VA into a PA.

BlueGuard cannot access the CPU control registers, including CR3. Instead, it performs an exhaustive memory search to find the PA of the VM's/hypervisor's kernel root page table. The scan uses a known memory forensic approach based on an IST: a structure containing offsets and sizes for kernel symbols in the virtual address space. The scan takes a known VA that points to a page with known contents, e.g., the init_task in the Linux kernel, and performs software address translation on it. The candidate that has a valid translation and points to a page with the expected content is deemed to be the root of the kernel's page table. The process page table physical addresses are stored in Linux in each task_struct. Thus, BlueGuard can recover them as well.

GPA-HPA translation. With VMs, a second level translation from GPA to HPA exists. The second level of translation can be performed by BlueGuard similarly to GVA to GPA by finding the physical address of the root of the extended page table. This allows BlueGuard to directly translate GPAs to HPAs akin to the method of Graziano et al. [22]. However, BlueGuard can offload the GPA to HPA translation to the IOMMU with PCIe passthrough of vDPUs to improve the introspection performance. Note, that the IOMMU may be tampered with by a compromised hypervisor, which does not introduce a new attack vector as translation redirection attacks are out of scope for BlueGuard and OOB introspection systems in general [26] (§ 3.4).

4.4 Inline computation

Managing the accelerators for VMI applications requires special care as unfortunately, VMs/hypervisor states exist in virtual memory, and the matching PAs are unlikely to be contiguous. Thus, invoking the accelerators on RAS buffers could yield an incorrect computation if the buffer crosses two pages. Note, LAS accesses are registered with DPU virtual memory so they do not share this issue. To overcome this, BlueGuard automatically detects cross-page access. Next, for each access to the regex or hashing engines, BlueGuard first copies the requested buffers to local memory and then triggers a subsequent request for the respected engine on the local memory that is contiguous. For non-cross page accesses BlueGuard performs the computation directly on the VM's memory. Finally, to reduce memory utilization, BlueGuard partitions the copies to 2 pages at a time for engines that do not maintain an internal state such as the regex. However, for engines such as SHA that do maintain a state all the pages are copied at once.

4.5 Delta introspection

We observe that the only input to VMI applications is the data accessed through the accelerators. Thus, to validate there are no changes in the introspected VMs' state, e.g., a new process spawned or terminated by the pslist VMI application, BlueGuard tracks all the accessed RAS pointers together with their respective size and last known values. Next, instead of re-running the VMI application, BlueGuard issues a batch DMA

request for all the RAS addresses and sizes as was traced by the original execution. Detecting a change is then based on a comparison check. To reduce the check's latency, a digest (SHA-256) of the original content and that of the latest invocation are matched. In case of a mismatch, the VMI application is executed traditionally to capture the new artifacts. Delta introspection facilitates a known RDMA programming optimization to accelerate memory accesses: doorbell batching [27], which is impossible for traditional VMI applications.

Finally, we note that hashing or pattern-matching changes only with content modifications. Therefore, BlueGuard can trace the original content using DMA requests for delta introspection.

VA-PA translation cache. Similar to existing VMI systems, BlueGuard maintains a translation cache to reduce the number of DMA accesses. However, BlueGuard is able to utilize delta introspection to identify when invalidations are required. Specifically, when sending the batched DMA request, BlueGuard tags the page table walk-related requests, and upon a delta being detected BlueGuard then checks for changes in the page table fetched content. Any change in the page table levels invalidates the corresponding GVA-GPA entry from the translation cache, and the next invocations of the VMI applications would ensure correct execution.

Note, this improves upon existing approaches that blindly trust the cache does not contain stale translations [77]. While it may still be possible to encounter stale translations as BlueGuard cannot introspect the TLB, in practice we did not encounter such a case. It does, however, come with a performance impact, as the inclusion of page table walk-related requests increases the number of DMA requests. We evaluate both cases.

4.6 Malware Isolation

The fact that BlueGuard runs on a DPU, which connects the host and guests to the network, allows controlling every network access by leveraging a virtual switch: Open vSwitch (OVS) [54]. Specifically, the NVIDIA BlueField DPU supports offloading VLAN management to OVS, which adds a VLAN tag to all packets sent and strips the tag for all packets received. Note, the host/VM interface is unaware of the VLAN tagging as the configuration is made on the DPU for netdev representors [37].

BlueGuard utilizes this to enforce network access restrictions and to contain a VM or the hypervisor as soon as it is considered infected. Specifically, BlueGuard employs network isolation policies, such that a potentially infected system can be placed in VLANs with only limited, up to no connectivity to other machines at all. This prevents attackers from spreading to other systems (lateral movement) and since this enforcement is done in the DPU, the attacker cannot easily revert a containment, in contrast to a compromised hypervisor for example.

5 Implementation

Our prototype of BlueGuard is implemented in C/C++ and closely follows the description in § 4. The only exception is our prototype does not currently implement malware isolation policies. Instead, our prototype configures OVS on the DPU to isolate VMs/hypervisor in pre-configured VLANs. BlueGuard consists of a VM agent used to set up the control plane with BlueGuard that is deployed in the NVIDIA Bluefield-2 DPU [51].

The implementation leverages the NVIDIA DevX interface and the DOCA SDK [50], which enables submitting requests to the DPU's accelerators.

BlueGuard threading model. We implement the N:M threading model using C++ coroutines and force yielding to a FIFO scheduler when VM access is performed. Note, we statically balance the task load per OS thread and observe our threading model maintains fairness.

Profiles. BlueGuard uses Volatility IST [75] to provide VMI applications with access to symbols offsets and sizes in the introspected guest virtual address space. Specifically, the IST profile is constructed with existing tools that utilize the kernel image (with debug symbols) and the system map [75]. Furthermore, for each VM and hypervisor, we extract the available physical memory regions to validate we only serve memory access requests to system memory.

VMI applications interface. BlueGuard's interface towards the VMI applications includes a VMI request queue: process_jobs(vmi_ctx, job_ctx[], callback), that appends single or multiple job contexts to the introspected VM/hypervisor. Each job context includes job_type: dma, regex, or sha, a PA to access, the size in bytes to access, and a destination buffer in the DPU memory to store the result into. The callback function is invoked by the scheduler once all jobs are completed with the job contexts and a status indicating success or failure. Thus, VMI application developers can send a single job or batch jobs with a single call. DPUs manage jobs in order so batching enables polling only the last submitted job to infer all jobs completed successfully.

Implemented VMI applications. VMI applications can be developed easily with BlueGuard's interface. We demonstrate this by implementing several VMI applications with similar functionality as used in prior work [18, 79], and by supporting all VMI applications listed in § 2.2. Note that all VMI applications use the IST to extract symbols' offsets and sizes and access only the minimum set of addresses to extract the artifacts.

Accelerated VMI applications. Notably, VMI application development is simple as it is programmed on general-purpose processors with known programming languages while BlueGuard abstracts the DPU hardware interfaces. For the two VMI applications modules_integrity and regex_scan, we each implement two versions. First, a standard, ARM-optimized version in software via the OpenSSL library [56] and the

POSIX.2 regex library, respectively. Second, a hardware alternative that makes use of the DPU accelerators, with the hashing accelerator for the modules_integrity application, and the regex accelerator for the regex_scan application.

For modules_integrity, our implementation batches hashing requests to the accelerator to improve performance by first capturing all the pages in the kernel modules' VMAs (§ 6.5).

Finally, both software implementations regex_scan_sw and modules_integrity_sw first issue a DMA request to a staging buffer and then issue the respective operation on it.

6 Evaluation

Hardware setup. The platform we use to evaluate BlueGuard is a ProLiant DL380 Gen10 server with two Intel(R) Xeon(R) Gold 6130 CPUs, 204 GB RAM, and 1.8 TB NVMe. The server is connected to an NVIDIA BlueField-2 DPU and uses its Ethernet adapters to connect to a switch through a trunk port. For the malware evaluation, we configured two VLANs: VLAN 10 for unrestricted internet access and VLAN 99 for containment and running malware. The VLAN assignment is enforced on the DPU through OVS configuration. Furthermore, we use a dedicated VM with a clean base snapshot for every malware execution. The host CPU runs Ubuntu Linux with a custom kernel compiled to support LibVMI with KVM based on version 5.4.

Note, unlike LibVMI, BlueGuard does not necessitate a custom kernel. Yet, to remove bias in experiments we opt to use the same kernel throughout all the experiments. The DPU runs the off-the-shelf NVIDIA Linux kernel 5.4.0-1035-bluefield. We install 16 VMs on the host using KVM and QEMU. Each VM has 4 vCPUs, 8 GB RAM, and 100 GB of storage while running Linux kernel 5.4.0-125. We use KVM and QEMU to manage the virtualization stack on the host and configure SR-IOV with the IOMMU set to passthrough mode.

Methodology. We run each experiment measuring latency 10 times and measuring throughput for at least 30 seconds. We report the mean and normalized values for latency and throughput tests respectively and validate that the standard deviation is within an acceptable range (5%). The introspection details for each evaluated VMI application are depicted in Table 1. Finally, when delta introspection is used, it is triggered continuously unless specified otherwise.

Table 1: Number of accesses and size per VMI application.

VMI application	# of accesses	Total memory accessed
pslist	1,236	9.6 KB
modules	643	8.6 KB
check_creds	962	7 KB
regex_scan	1,227	9.5 KB
modules_integrity	8,055	302.9 KB

6.1 Malware detection and isolation

In this section, we evaluate BlueGuard's effectiveness for malware detection.

Open source rootkits. We evaluate Diamorphine [11] and Reptile [63], which are based on a loadable kernel module using the modules VMI application. Both rootkits offer several functionalities, but their hiding capabilities are most important for this evaluation: both hide themselves automatically during initialization. We run the rootkits 100 times at an arbitrary time in a single VM, out of 16 VMs concurrently monitored by BlueGuard. We employ delta introspection to detect changes in the modules list followed by the modules VMI application to detect the malicious kernel module. BlueGuard achieves 91% and 100% detection rates for Diamorphine and Reptile respectively before they can hide themselves.

To understand the detection rate achieved, we consider the probability detection model we define in Appendix A. In short: If the time window of an artifact in memory is as long as it takes to execute delta introspection twice and the plugin once, we expect a 100% detection rate. As we measure the average latency for delta introspection and the modules plugin to be 108.2$usec$ and 1.38$msec$ per VM, we expect 100% for a time window of at least 1.59$msec$ ($2 \cdot 108.2usec + 1.38msec$). During tests, we conservatively measure the time window in which the rootkits are available in the kernel modules list to be 13$msec$ and 120$msec$ for Diamorphine and Reptile respectively. It is thus inline with our model to achieve 100% detection rate for Reptile. Yet, we also expect to observe a 100% detection rate for Diamorphine as the time window is a multiple of the aggregated latencies. We suspect the difference is due to P90 latency, which causes the aggregated latency of delta introspection and modules to be larger than 13$msec$, or the time window in which the module is available in the modules list to be significantly smaller. Yet, we did not investigate it further.

Real-life malware. Next, we evaluate malware samples of the families Cl0p, Gafgyt, Mirai, XMRig, DinodasRAT, AresRAT, Babuk and Kaiji (see Appendix C). We use the regex_scan and pslist VMI applications to scan for indicators of compromise (IoCs) in memory: onion addresses, and suspicious parent/child relationships (sh child processes), respectively. We measure the time windows of sh child processes to be as low as 1$msec$, and the onion addresses to exist for at least 1 second. In both cases we monitor a single VM. We measure the average latency for delta introspection and pslist for a single system to be 214.7$usec$ and 2.49$msec$, so we expect a detection rate of 100% for a time window of at least 2.92$msec$ ($2 \cdot 214.7usec + 2.49msec$). The ground truth IoCs for each malware is gathered with forkstat, execsnoop–bpfcc and from online sandbox systems. We detect 108/132 sh child processes

Table 2: Memory access latency comparing BlueGuard to LibVMI.

# of Bytes	LibVMI	BlueGuard VM (Bare Metal)
4	11.34 usec	2.7 usec (2.7 usec)
8	11.04 usec	2.7 usec (2.7 usec)
16	11.08 usec	2.7 usec (2.7 usec)
32	11.07 usec	2.7 usec (2.7 usec)
64	11.87 usec	2.7 usec (2.7 usec)
1K	11.03 usec	3.3 usec (3.3 usec)
4K	11.04 usec	3.6 usec (3.4 usec)
1M	3 msec	108.1 usec (82.2 usec)
2M	6.1 msec	180.6 usec (160.7 usec)

(82%), and observe a total of 34 sh processes had a time window between 1*msec* to 3*msec*. Also, we identify the onion addresses IoCs for every executed malware based on the online sandbox results. To conclude, BlueGuard successfully detects temporary IoCs for real-life malware.

Malware isolation. BlueGuard monitors a dedicated VM, initially assigned to VLAN 10, and waits for suspicious IoCs. Once detected, BlueGuard isolates this VM into VLAN 99 by invoking OVS. To evaluate the latency for isolating a VM, we consider ping as a malicious process, which is configured to continuously send ICMP packets every 1*nsec* to an external server on the internet, accessible via VLAN 10 but not VLAN 99. As ping sends its packets sequentially, we conservatively calculate the time until isolation occurs with the aggregated RTT times of all successfully received packets. The process initialization time for ping until it starts sending packets is neglected for this experiment. For the detection and containment we created a modified version of the pslist VMI application that immediately isolates the VM via OVS. We run 100 iterations, and observe ping is isolated after 27.7*msec* on average with a minimum of 12.19*msec* and maximum of 83.04*msec*.

6.2 Introspection efficiency

Host VMI comparison. First, we compare BlueGuard to the popular host-based LibVMI framework [77]. We implement a simple VMI application that reads a physical address of configurable size into a local buffer for 1 million iterations to remove noise. We execute it on a single VM and disable all caches in LibVMI and BlueGuard to make a fair comparison.

The results are reported in Table 2. First, we observe BlueGuard is much faster compared to LibVMI: about 3× for small buffers up to 4 KB, and 27× to 33× for larger buffers: 1 MB and 2 MB respectively. For small buffers, we attribute the speedup to BlueGuard kernel-bypass and zero-copy approach, whereas LibVMI relies on UNIX sockets. The speedup grows with larger buffers due to the DMA engines.

Note, this showcases the benefits of DPUs: by using SR-IOV, BlueGuard enables direct access to the vDPU hardware.

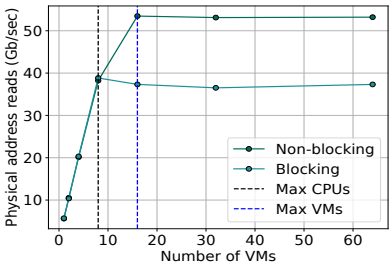


Figure 7: Memory access bandwidth while reading physical pages for a different number of VMs using BlueGuard.

This is unlike a traditional OOB system design that requires costly kernel mode transitions to virtualize access to DMA engines across HGI applications.

Bare metal introspection comparison. To understand the performance differences between VM and bare metal introspection we use the aforementioned VMI application. Yet, instead of accessing a VM's memory, we configure BlueGuard to access the host's memory. We observe that bare metal introspection is slightly faster compared to VMs. We attribute this to the additional GPA-HPA translations performed by the IOMMU when introspecting VMs. Specifically, as this VMI application reuses the same buffers, for small buffers the translation fits in the IOTLB thereby hiding the added latency. However, larger buffers oversubscribe the IOTLB resulting in increased translation latency.

HGI systems comparison. To the best of our knowledge, the highest reported introspection bandwidth of prior HGI systems is 250 MB/second [15, 32, 66]. To compare with BlueGuard we create a simple VMI application that reads the entire VM memory to a local DPU buffer 2 MB at a time while, and discarding old read data. We observe a bandwidth of 53 Gb/second, which is about 27× higher compared to prior OOB approaches, e.g., PCIleech [15]. We compare BlueGuard and RDMI introspection latency with VMI applications next.

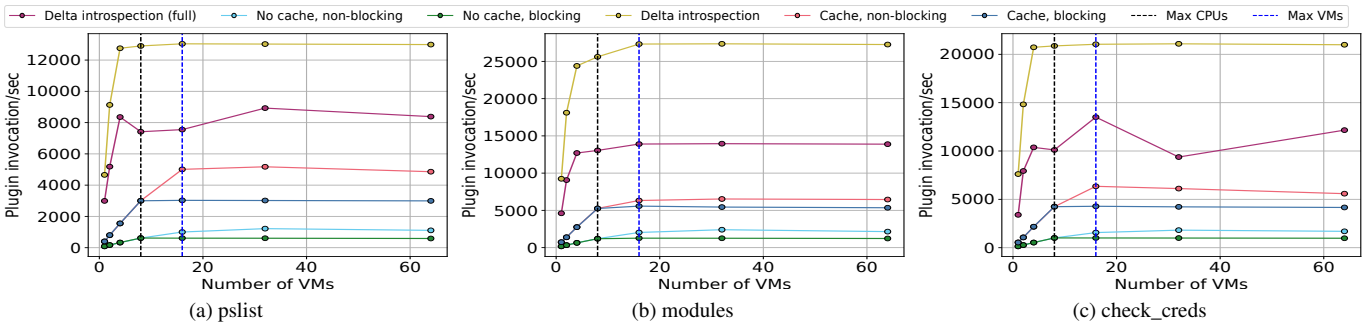
6.3 BlueGuard introspection scalability

As server processors host tens of VMs [44], we evaluate BlueGuard when introspecting multiple VMs in parallel.

Memory access. We implement a VMI application that reads a 4 KB buffer from a varying number of VMs. We choose 4 KB as we observe many VMI applications perform reads of 4 KB or less, and Table 2 demonstrates the access latency of 4 KB is similar to that of smaller access sizes.

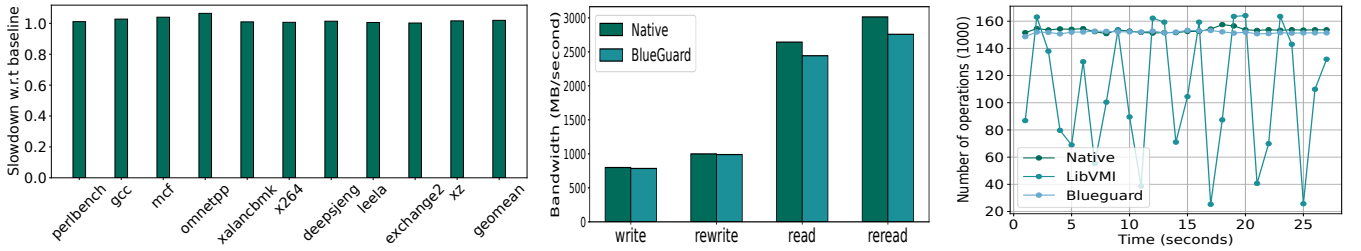
We run the VMI application and compute a normalized memory accesses/second value. To evaluate BlueGuard capability to introspect multiple VMs, we vary the number of CPU cores used from 1 to the maximum available in our platform: 8, with each hardware thread introspecting a VM.

Context switch overhead. To evaluate BlueGuard's thread-



(a) pslist (b) modules (c) check_creds

Figure 8: Throughput of VMI applications while varying the number of introspected VMs.



(a) SPEC CPU2017 normalized slowdown while introspected by BlueGuard.

(b) IOZone bandwidth with/without BlueGuard introspection.

(c) Redis throughput during introspection: native vs. LibVMI vs. BlueGuard.

Figure 9: Performance impact of VMI applications on the introspected VMs.

ing model, we issue *non-blocking memory accesses* with BlueGuard's user threads that yield after issuing requests, and *blocking memory accesses* that use kernel threads and busy-wait for completion while relying on the OS for context switches. For example, introspecting 16 VMs uses 16 and 8 kernel threads for blocking and non-blocking configurations respectively.

The results are reported in Figure 7. Once BlueGuard accesses more than 8 VMs the available compute resources of the DPU are oversubscribed, and kernel threads suffer from context switch overheads. However, BlueGuard threading model scales better as it can overlap memory accesses with compute, which improves the bandwidth by up to 40%.

VM scalability. Note, due to the limited resources in the host CPU used for evaluation, we can only execute up to 16 VMs concurrently. To understand BlueGuard scalability after this point we simulate up to 64 VMs by creating multiple VMI contexts in BlueGuard, with each context containing its own set of resources. We compare the throughput of accessing 4 KB buffers with 16 VMI contexts introspecting a single VM to 16 VMI contexts each introspecting a different VM. We observe the same throughput is achieved in both experiments, which boosts our confidence that simulated VMs' can be representative when evaluating BlueGuard's scalability beyond 16 VMs. Figure 7 shows that BlueGuard reaches a saturation point for 16 VMs. We note that while the introspection bandwidth is saturated when introspecting 16 VMs, a larger number of introspected VMs is possible but will experience reduced performance due to a bottleneck of the underlying BlueField-2 device. This bottleneck stems from the DPU resources that are shared between each introspection context, e.g., the DPU hardware queues and work elements. BlueGuard introspection exhibits graceful degradation beyond 16 VMs which results in a fair sharing of the introspection bandwidth between VMs. Figure 7 demonstrates BlueGuard's capability to concurrently introspect up to 64 VMs with state-of-the-art introspection performance. To the best of our knowledge, this result shows that BlueGuard can scale to match modern data center virtualization requirements.

VMI applications performance. We evaluate the pslist, modules, and check_creds VMI applications with BlueGuard in four configurations: with and without address translation caching, and with and without blocking memory accesses. Note, we configure BlueGuard with every VMI context having its own dedicated translation cache to correctly simulate more than 16 VMs. Similar to the memory access experiment, we vary the number of CPU cores used. Blocking accesses use hardware thread per VM, and the non-blocking accesses use BlueGuard's user threads.

We report the number of end-to-end executions of each VMI application/second in Figure 8. Similarly to prior work, we observe the effectiveness of the address translation cache in improving the throughput. Also, non-blocking memory

accesses improve the throughput and with translation caching disabled, it partially hides the translation latency. Finally, we observe that with BlueGuard VMI applications can introspect up to 64 VMs without noticeable performance degradation.

Delta introspection. Next, we evaluate the impact of doorbell batching via delta introspection with BlueGuard. For this setup we only run the corresponding VMI application once to fill the delta introspection cache and from that point on only check the cache for any deltas. Whether or not a delta is recognized does not change the execution behavior, in order to get unadulterated results. We are focusing on two different flavors: Delta Introspection (full), which covers all DMA requests including page table translations (§4.4), and Delta Introspection without the translations. Figure 8 shows that when batching 64 requests all the VMI applications enjoy about an order-of-magnitude acceleration for a single thread/VM. However, the benefits diminish due to in-NIC contention with a large number of concurrent requests. Additionally, the cost for covering the page table translations with Delta Introspection (full) decreases the throughput by nearly 50%.

Comparison with RDMI. While not focusing on scalability, RDMI involves in-network traversal that increases the introspection latency. Specifically for check_creds RDMI reports a latency of 4.47 msec, which results in an execution time of 8.94*msec* ($2 \cdot 4.47msec$) for a 100% detection rate (§3, §A). As we measure the average latency for delta introspection and the check_creds plugin to be 131.3*usec* and 1.82*msec* respectively for a single VM, BlueGuard's latency for a 100% detection rate is 2.08*msec* ($2 \cdot 131.3usec + 1.82msec$), a 4.3× speedup. Note, using the DPU accelerators further improves the introspection efficiency (§ 6.5).

6.4 VMs performance impact

We evaluate BlueGuard's performance impact on the VM workloads using SPEC CPU2017 [6], and IOzone [7] to simulate compute-sensitive and I/O-sensitive workloads respectively.

Compute. We run the SPEC CPU2017 benchmark suite on a single VM with and without BlueGuard executing in the background and measure the latency of the applications. With BlueGuard, we execute the VMI application that accesses the memory of all 16 of the VMs (§6.3), which maximizes BlueGuard's stress on the host's memory subsystem.

We report the normalized slowdown BlueGuard incurs on the applications compared to the non-introspected native execution in Figure 9a. We do not observe a noticeable slowdown due to the increased memory utilization by BlueGuard.

Storage. As storage devices share the PCIe with DPUs, we investigate whether high-bandwidth introspection impacts workloads that are I/O-sensitive. To that end, we run the IOzone benchmark suite on a single VM, with and without BlueGuard using the aforementioned memory access VMI application in the same configuration as before. IOzone measures the bandwidth of different file access patterns and we configure it to run with all vCPUs in the VM and access files that are twice as large as the available VM's memory to ensure storage accesses. We report the observed bandwidth achieved with and without BlueGuard in Figure 9b.

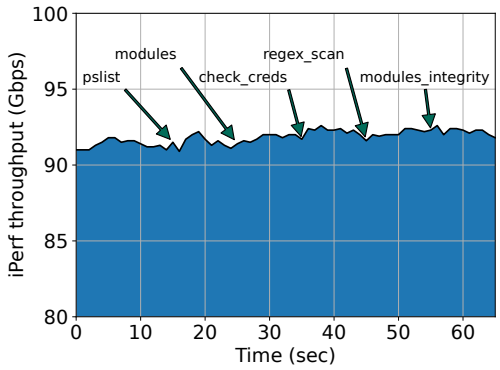


Figure 10: Performance isolation with networking and BlueGuard. Arrows mark points in time that VMI applications are executed on the DPU.

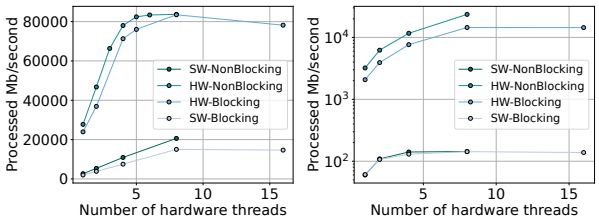
We observe a small decrease in throughput for read and re-read access patterns. We attribute this to the kernel's page cache and prefetching mechanisms that mask storage accesses. Note, to the best of our knowledge, even without a page cache a performance degradation for storage access cannot be used to detect BlueGuard as the storage is shared across tenants without fairness guarantees.

Performance isolation. To demonstrate how VMI applications sharing the physical machine with VMs create the noisy neighbor effect we measure the throughput of Redis [62], a popular in-memory data structure store using the workload generator memtier benchmark. First, we run Redis and the workload generator in a single VM, capturing the number of observed operations every second for 30 seconds. To reduce noise we pin Redis to CPU 0. Next, we run the pslist VMI application using LibVMI [77], which can be used to detect potential malware that gained execution privileges.

We continuously monitor the VM and run the VMI application in a loop. To get an indication of the lack of performance isolation we also pin pslist to the same CPU 0 as Redis. We report the values observed in Figure 9c. It is clear that the lack of performance isolation causes a significant drop in throughput, which harms the user experience. For comparison, we run pslist with BlueGuard. As can be seen in Figure 9c, it achieves complete performance isolation while offloading the CPU.

Note, dedicating hypervisor cores for VMI limits the scalability of monitoring VMs and takes away cores that can be rented by CSPs for tenants. Unlike them, DPU cores are not rented to tenants but offload infrastructure workloads from the hypervisor and are managed by CSPs.

Networking. To measure the impact of BlueGuard on the



(a) modules_integrity (b) regex_scan

Figure 11: Number of bytes processed using a different number of threads with the DPU's accelerators (HW) and with the DPU's processor (SW).

network performance, we conduct an experiment similar to prior work [39]. Specifically, we run iperf on the host and execute different VMI applications concurrently with BlueGuard on the DPU. The network bandwidth achieved by iperf without BlueGuard averages at roughly 93 Gb/second. While iperf is executing, we execute each VMI application once. Figure 10 illustrates the results of this experiment. We observe that BlueGuard incurs a negligible impact on the network.

6.5 Hardware acceleration

Accelerated kernel modules integrity verification. We evaluate BlueGuard with the modules_integrity VMI application with and without the DPU's hashing engine, and with and without blocking memory accesses. We execute each configuration with address translation caching enabled and with a different number of hardware threads. For the non-blocking configuration, we introspect 2 VMs per thread, which we find provides the highest throughput. We report our findings in Figure 11a.

We observe a 12× speedup comparing software-based hashing to hardware acceleration. This is because the hardware accelerator computes the hash with zero copies. Also, we observe that BlueGuard is able to reach 80 Gb/second bandwidth when monitoring kernel module integrity. Surprisingly, we observe that the hash engine reaches a higher introspection bandwidth compared to regular memory accesses via the DMA engine. We attribute this to our implementation that batches requests. Finally, we observe that with non-blocking memory accesses 5 threads are sufficient to reach the maximum bandwidth for this VMI application compared to 8 threads that are needed with the blocking approach.

Accelerated Regular Expression matching. We evaluate BlueGuard with the regex_scan VMI application with and without the DPU's regex engine, and with and without blocking memory accesses. For that purpose we utilize a YARA rule for the Linux/Moose malware [5] from the YARA rules project [74] and convert it into a regular expression. YARA rules themselves support textual, binary and regular expression patterns that are used to identify specific binary streams, such as in malware, and hence are utilized to identify certain malware families and samples. Since the YARA project supports regular expressions, it can benefit too from performance improvements by the DPU's regex engine. In our test, regex_scan is configured to scan all VMAs with the generated regex for a single process: Redis [62] (more details are in Appendix B).

We execute both the hardware and software configurations with a different number of hardware threads while introspecting a single VM for setup simplicity. For the non-blocking configuration, we introspect 4 instances of the VM per thread, which we find provides the peak throughput. Finally, we enable the translation cache and pre-compute all VMAs' pages with valid mappings and scan them for matching patterns.

We report the results in Figure 11b, and observe that hardware acceleration improves the performance by two orders of magnitude compared to software implementation. To the best of our knowledge, the DPU's CPUs do not contain regex acceleration instructions resulting in a larger gain compared to hashing. Also, we observe the staging buffer acts as a bottleneck, as each invocation of the VMI application must first copy data to the buffer and then perform the regex operations on it. However, the non-blocking design of BlueGuard minimizes the bottleneck since both the DMA and regex invocations can be overlapped with compute.

We consider that some VMI applications may opt to detect newly installed page mappings for improved security. Thus, we evaluate the staging buffer construction without precomputing page mappings in the VMAs. We observe a 2.5× slowdown for the maximal reported throughput in Figure 11b.

7 Discussion

Additional VMI accelerators. DPUs are undergoing active development and may introduce new hardware accelerators that can benefit VMI applications. For example, the NVIDIA Bluefield-2X DPU connects a DPU with a GPU over a PCIe switch and enables efficient communication via GPUDirect [49]. In future work, we consider using BlueGuard to further accelerate malware detection via accelerated machine learning inference.

Integration with existing VMI tools. We do not consider it challenging to retrofit existing VMI applications to utilize BlueGuard similarly to prior VMI frameworks. We take Volatility [75] as a case example. Volatility relies on a LibVMI interface to perform live introspection by providing access to the VMs' memory. This interface includes creating and destroying a VMI context, reading and writing from a VM's memory with and without zero-padding, and finally retrieving all the available addresses and checking whether an address is valid for the VM. All of this is supported by BlueGuard and can be exposed externally, except for writing to the VMs' memory, as remediation is out of BlueGuard's scope. Such

integration would enable executing Volatility plugins with BlueGuard.

Invalid page translation. While rare, it is possible to encounter an address translation failure in BlueGuard due to absent pages' mappings in the target VM. While stealthy malware may attempt to evade detection by invalidating specific pages, once they are needed for correct malware execution they will have to be installed in the VM's page table, which will make them visible to BlueGuard. Further, with delta introspection, BlueGuard can detect them efficiently.

Cache coherence and internal CPU states. BlueGuard shares similar limitations with prior OOB VMI systems. First, BlueGuard uses the DPU's accelerators, which are not cache-coherent with the host CPU. However, recent advancement in CXL [10] is likely to get adopted by future DPUs and BlueGuard can enjoy them similarly to SR-IOV. Second, BlueGuard cannot introspect CPU internal states such as registers and internal memory. While this can lead to address translation manipulation attacks, e.g., manipulating the CR3 register in x86 [26], this is both a shared limitation with prior OOB systems [26, 66] and VMI applications built with BlueGuard can utilize delta introspection and regex acceleration to detect page table structures faster than existing OOB systems.

Introspection consistency. Unlike host VMI systems, BlueGuard does not pause VMs during introspection as this leads to major overheads. We validated this in an earlier prototype that used a cooperative protocol between a DPU agent and the hypervisor. The hypervisor trapped writes to CR3 and sent the new values to the DPU. This is similar to IMEE [79] approach ensuring BlueGuard always uses the correct page table.

Unfortunately, the recent Meltdown mitigation: kernel page table isolation (KPTI) [38] involves switching the page table on every kernel crossing. Thus, trapping the CR3 updates resulted in VM exits per kernel crossing and reduced the throughput of Redis [62] (same setup as described in § 6.4) by 15×. This result convinced us to pursue a non-cooperative design that maintains performance isolation. Furthermore, it is unclear whether the performance guarantees of systems such as IMEE that rely on trapping are kept with KPTI.

IOMMU configuration attacks. Since BlueGuard excludes the hypervisor from the TCB, the hypervisor may be compromised and misconfigure the IOMMU to hide introspection. Like RDMI's [39] proposal, BlueGuard can use the PCIe ATS feature to tag all DMA requests as translated, bypassing IOMMU translations. This applies to both hypervisor and VM introspection. For hypervisor introspection, BlueGuard directly translates GPAs to HPAs. For VM introspection, BlueGuard can employ direct GVA to HPA translation (§4.3). BlueGuard using ATS translated requests will force access to the correct HPA regardless of IOMMU configuration.

8 Related Work

VMI systems. VMI systems have been extensively studied and analyzed for security, usability, and performance [25, 69].

Host-based VMI systems [2, 8, 12, 16, 23, 40, 64, 65, 67, 76, 77, 79] either cooperate with the introspected VMs' kernel or use virtualization to introspect VMs. However, they lack the capability of introspecting a bare metal machine or the hypervisor and compete for CPU resources. BlueGuard does not use host resources and allows HGI.

OOB VMI systems [19, 31, 32, 36, 46, 48, 58, 66, 80] perform introspection from an external device similar to BlueGuard. These systems focused, however, mostly on the introspection of the host [29, 48, 66] while BlueGuard introspects the host and any guest.

Recently, NVIDIA published a blog on OOB malware detection using DPUs [53]. However, the blog post lacks details on the performance and scalability of their system, which is the focus of BlueGuard.

Finally, RDMI [39] uses programmable switches and RDMA NICs to provide OOB memory introspection. Unlike RDMI, BlueGuard does not pass any memory over RDMA, which is known to be vulnerable to attacks by network adversaries [71], and further improves introspection performance over RDMI through delta introspection, DPU accelerators and caching. Additionally, BlueGuard isolates infected hosts and guests from the network.

DPU offload. Many approaches offload host tasks to DPUs. Lynx [72] offloads server data and control planes, LineFS [28] offloads distributed file systems, AccelNet [13] offloads TCP and SDN stacks, and Floem [61] and ClickNP [35] offload network functions. BlueGuard shares the same vision of offloading infrastructure services from the host CPU and accelerating them using DPUs. However, these papers do not address the unique challenges of VMI applications.

LibOSs. BlueGuard is inspired by libOSs that accelerate I/O [4, 9, 14, 60, 78]. Similarly, BlueGuard bypasses the kernel and utilizes hardware virtualization to accelerate introspection I/O. However, BlueGuard focuses on VMI applications' requirements and DPUs' hardware accelerators.

9 Conclusion

BlueGuard demonstrates the applicability of DPUs to efficiently and scalably enable HGI in data centers. BlueGuard considers VMI applications' requirements and the DPU hardware resources to provide 4.3× faster detection compared to prior HGI systems, without hindering VMs' performance. Finally, BlueGuard also enables malware isolation upon detection by utilizing DPUs' network offload capabilities.

Acknowledgments

We thank the anonymous reviewers and the shepherd for their help in improving the manuscript.

Ethics considerations

This paper contains no survey with human involvement or an evaluation with live systems in the real world. All provided evaluations were performed in a lab environment that is strictly isolated from the authors' systems as well as from the internet. A list of all used malware samples can be found in Appendix C, and the isolated environment ensured that this paper did not assist in accidentally spreading these samples to our own systems or the wider world (see § 6).

Open science

Due to licensing reasons, we are unable to provide the full artifact of BlueGuard. Yet, we follow the spirit of open science in our best effort by providing the following pieces in [57]:

1. Proof of concept code of an earlier version of BlueGuard to allow future researchers to have a second source of reference when wishing to reproduce our results. This code is not a fully reproducible artifact but we see it as a valid source of ground truth to accompany the paper text.
2. Code of the VMI applications as they are used in BlueGuard, to allow reviewing of our claim that VMI application writing needs no extensive modifications to be used in BlueGuard.
3. Raw log data as well as the scripts used to reproduce the graphs and evaluation data presented in the paper.

References

[1] *MalwareBazaar*. abuse.ch, 2024. https://bazaar.abuse.ch/.

[2] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. HyperSentry: Enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, page 38–49, New York, NY, USA, 2010. Association for Computing Machinery. doi: 10.1145/1866307.1866313.

[3] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. DKSM: Subverting virtual machine introspection for fun and profit. In *2010 29th IEEE Symposium on Reliable Distributed Systems*, pages 82–91, 2010. doi: 10.1109/SRDS.2010.39.

[4] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay.

[5] Olivier Bilodeau and Thomas Dupuy. *Dissecting Linux/Moose*, 2022. http://www.welivesecurity.com/wp-content/uploads/2015/05/Dissecting-LinuxMoose.pdf.

[6] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery. doi: 10.1145/3185768.3185771.

[7] Don Capps and William Norcott. IOzone filesystem benchmark, 2008.

[8] Martim Carbone, Matthew Conover, Bruce Montague, and Wenke Lee. Secure and robust monitoring of virtual machines through guest-assisted introspection. In Davide Balzarotti, Salvatore J. Stolfo, and Marco Cova, editors, *Research in Attacks, Intrusions, and Defenses*, pages 22–41, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[9] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for ms-scale RPCs with breakwater. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[10] *Compute Express Link*. Compute Express Link Consortium, 2023. https://www.computeexpresslink.org.

[11] *LKM rootkit for Linux Kernels*. Diamorphine, 2023. https://github.com/m0nad/Diamorphine.

[12] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *2011 IEEE Symposium on Security and Privacy*, pages 297–312, 2011. doi: 10.1109/SP.2011.11.

[13] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen

Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, April 2018. USENIX Association.

[14] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[15] Ulf Frisk. *PCIleech*, 2022. https://github.com/ufrisk/pcileech.

[16] Yangchun Fu and Zhiqiang Lin. Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *2012 IEEE Symposium on Security and Privacy*, pages 586–600, 2012. doi: 10.1109/SP.2012.40.

[17] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, volume 3, pages 191–206. San Diega, CA, 2003.

[18] *Virtual Machine Threat Detection overview*. Google, 2022. https://cloud.google.com/security-command-center/docs/concepts-vm-threat-detection-overview.

[19] *GRR Rapid Response: remote live forensics for incident response*. Google, 2022. https://github.com/google/grr.

[20] *Rekall Forensics*. Google, 2022. http://www.rekall-forensic.com.

[21] *The next wave of Google Cloud infrastructure innovation: New C3 VM and Hyperdisk*. Google, 2023. https://cloud.google.com/blog/products/compute/introducing-c3-machines-with-googles-custom-intel-ipu.

[22] Mariano Graziano, Andrea Lanzi, and Davide Balzarotti. Hypervisor memory forensics. In *Research in Attacks, Intrusions, and Defenses*, pages 21–40, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[23] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process implanting: A new active introspection framework for virtualization. In *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, pages 147–156, 2011. doi: 10.1109/SRDS.2011.26.

[24] *Intel Infrastructure Processing Unit (Intel IPU)*. Intel, 2022. https://www.intel.com/content/www/us/en/products/network-io/smartnic.html.

[25] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion. SoK: Introspections on trust and the semantic gap. In *IEEE Symposium on Security and Privacy*, pages 605–620, 2014. doi: 10.1109/SP.2014.45.

[26] Daehee Jang, Hojoon Lee, Minsu Kim, Daehyeok Kim, Daegyeong Kim, and Brent Byunghoon Kang. ATRA: Address translation redirection attack against hardware-based external monitors. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 167–178, New York, NY, USA, 2014. Association for Computing Machinery. doi: 10.1145/2660267.2660303.

[27] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

[28] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 756–771, New York, NY, USA, 2021. Association for Computing Machinery. doi: 10.1145/3477132.3483565.

[29] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. BareBox: Efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, page 403–412, New York, NY, USA, 2011. Association for Computing Machinery. doi: 10.1145/2076732.2076790.

[30] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[31] Yohei Kuga, Ryo Nakamura, Takeshi Matsuya, and Yuji Sekiya. NetTLP: A development platform for PCIe devices in software interacting with hardware. In *17th USENIX Symposium on Networked Systems Design*

and Implementation, pages 141–155. USENIX Association, February 2020. URL https://www.usenix.org/conference/nsdi20/presentation/kuga.

[32] Tobias Latzo, Julian Brost, and Felix Freiling. BMCLeech: Introducing stealthy memory forensics to BMC. *Forensic Science International: Digital Investigation*, 32:300919, 2020. ISSN 2666-2817. doi: https://doi.org/10.1016/j.fsidi.2020.300919. URL https://www.sciencedirect.com/science/article/pii/S2666281720300147.

[33] Hojoon Lee, HyunGon Moon, DaeHee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Brent ByungHoon Kang. KI-Mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 511–526, Washington, D.C., August 2013. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/lee.

[34] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, page 386–395, New York, NY, USA, 2014. Association for Computing Machinery. doi: 10.1145/2664243.2664252.

[35] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery. doi: 10.1145/2934872.2934897.

[36] Letitia W. Li, Guillaume Duc, and Renaud Pacalet. Hardware-assisted memory tracing on new socs embedding fpga fabrics. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC '15, page 461–470, New York, NY, USA, 2015. Association for Computing Machinery. doi: 10.1145/2818000.2818030. URL https://doi.org/10.1145/2818000.2818030.

[37] *Network Function Representors*. The Linux Kernel, 2024. https://docs.kernel.org/next/networking/representors.html.

[38] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[39] Hongyi Liu, Jiarong Xing, Yibo Huang, Danyang Zhuo, Srinivas Devadas, and Ang Chen. Remote direct memory introspection. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6043–6060, Anaheim, CA, August 2023. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity23/presentation/liu-hongyi.

[40] Yutao Liu, Yubin Xia, Haibing Guan, Binyu Zang, and Haibo Chen. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 416–427, 2014. doi: 10.1109/HPCA.2014.6835951.

[41] Ziyi Liu, JongHyuk Lee, Junyuan Zeng, Yuanfeng Wen, Zhiqiang Lin, and Weidong Shi. CPU transparent protection of OS kernel and hypervisor integrity with programmable DRAM. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 392–403, New York, NY, USA, 2013. Association for Computing Machinery. doi: 10.1145/2485922.2485956.

[42] *Data Processing Units (DPU) Empowering 5G carrier, enterprise and AI cloud data infrastructure*. Marvell, 2024. https://www.marvell.com/products/data-processing-units.html.

[43] *VM Inspector for Azure virtual machines (Preview)*. Microsoft, 2022. https://learn.microsoft.com/en-us/troubleshoot/azure/virtual-machines/vm-inspector-azure-virtual-machines.

[44] *Virtual machines in Azure*. Microsoft, 2022. https://learn.microsoft.com/en-us/azure/virtual-machines/overview.

[45] *Project Catapult*. Microsoft, 2022. https://www.microsoft.com/en-us/research/project/project-catapult/.

[46] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. Vigilare: Toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 28–37. Association for Computing Machinery, 2012. doi: 10.1145/2382196.2382202.

[47] *Agilio SmartNICs - Netronome*. Netronome, 2024. https://netronome.com/agilio-smartnics/.

[48] Jr. Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot—a coprocessor-based kernel runtime integrity monitor. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association.

[49] *GPUDirect RDMA*. NVIDIA, 2022. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[50] *DOCA SDK Documentation*. NVIDIA, 2022. https://docs.nvidia.com/doca/sdk/.

[51] *NVIDIA BlueField Data Processing Units*. NVIDIA, 2022. https://www.nvidia.com/en-us/networking/products/data-processing-unit.

[52] *Oracle Cloud Infrastructure Chooses NVIDIA BlueField Data Center Acceleration Platform*. NVIDIA, 2023. https://web.archive.org/web/20240229015230/https://nvidianews.nvidia.com/news/oracle-cloud-infrastructure-chooses-nvidia-bluefield-data-center-acceleration-platform.

[53] *Detecting Out-of-Band Malware with NVIDIA BlueField DPU*. NVIDIA, 2023. https://developer.nvidia.com/blog/detecting-out-of-band-malware-with-bluefield-dpu/.

[54] *Virtual Switch on BlueField DPU*. NVIDIA, 2023. https://docs.nvidia.com/networking/display/ bluefield-dpuosv385/virtual+switch+on+bluefield+dpu.

[55] *PHYSICAL ADDRESS MEMORY REGION*. NVIDIA, 2023. https://enterprise-support.nvidia.com/s/article/physical-address-memory-region.

[56] *OpenSSL Cryptography and SSL/TLS Toolkit*. OpenSSL project, 2022. https://www.openssl.org/.

[57] Meni Orenbach, Rami Ailabouni, Nael Masalha, Ahmad Saleh, Frank Block, Fritz Alder, and Ahmad Atamli-Reineh. Artifact for "BlueGuard: Accelerated Host and Guest Introspection Using DPUs", January 2025. URL https://doi.org/10.5281/zenodo.14725234.

[58] Ralph Palutke, Simon Ruderich, Matthias Wild, and Felix Freiling. HyperLeech: Stealthy system virtualization with minimal target impact through DMA-Based hypervisor injection. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 165–179, San Sebastian, October 2020. USENIX Association. URL https://www.usenix.org/conference/raid2020/presentation/palutke.

[59] *PCI Express Base Specification*. PCI-SIG, January 2024. Revision 6.2.

[60] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter.

[61] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for NIC-accelerated network applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 663–679, USA, 2018. USENIX Association.

[62] *Redis*. Redis Labs, 2022. https://redis.io.

[63] *LKM Linux rootkit*. Reptile, 2023. https://github.com/f0rb1dd3n/Reptile.

[64] Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[65] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-VM monitoring using hardware virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 477–487, New York, NY, USA, 2009. Association for Computing Machinery. doi: 10.1145/1653662.1653720.

[66] Chad Spensky, Hongyi Hu, and Kevin Leach. LO-PHI: Low-observable physical host instrumentation for malware analysis. In *23th Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[67] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. Process out-grafting: An efficient "out-of-VM" approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, page 363–374, New York, NY, USA, 2011. Association for Computing Machinery. doi: 10.1145/2046707.2046751.

[68] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 733–750, New York, NY, USA, 2020. Association for Computing Machinery. doi: 10.1145/3373376.3378450.

[69] Sahil Suneja, Canturk Isci, Eyal de Lara, and Vasanth Bala. Exploring VM introspection: Techniques and trade-offs. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, page 133–146. Association for Computing Machinery, 2015. doi: 10.1145/2731186.2731196.

[70] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013. doi: 10.1109/SP.2013.13.

[71] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. sRDMA – efficient NIC-based authentication and encryption for remote direct memory access. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 691–704. USENIX Association, July 2020. URL https://www.usenix.org/conference/atc20/presentation/taranov.

[72] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A SmartNIC-driven accelerator-centric architecture for network servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 117–131, New York, NY, USA, 2020. Association for Computing Machinery. doi: 10.1145/3373376.3378528.

[73] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 306–324, New York, NY, USA, 2017. Association for Computing Machinery. doi: 10.1145/3132747.3132762.

[74] *Yara the pattern matching swiss knife for malware researchers (and everyone else)*. VirusTotal, 2022. https://virustotal.github.io/yara.

[75] *The Volatility Framework*. The Volatility Foundation, 2022. https://www.volatilityfoundation.org/.

[76] Jiang Wang, Angelos Stavrou, and Anup Ghosh. Hypercheck: A hardware-assisted integrity monitor. In Somesh Jha, Robin Sommer, and Christian Kreibich, editors, *Recent Advances in Intrusion Detection*, pages 158–177, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[77] Haiquan Xiong, Zhiyong Liu, Weizhi Xu, and Shuai Jiao. LibVMI: a library for bridging the semantic gap between guest OS and VMM. In *12th International Conference on Computer and Information Technology*, pages 549–556. IEEE, 2012.

[78] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath OS architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211. ACM, 2021. doi: 10.1145/3477132.3483569.

[79] Siqi Zhao, Xuhua Ding, Wen Xu, and Dawu Gu. Seeing through the same lens: Introspecting guest address space at native speed. In *26th USENIX Security Symposium*, pages 799–813, August 2017. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/zhao.

[80] Lei Zhou, Jidong Xiao, Kevin Leach, Westley Weimer, Fengwei Zhang, and Guojun Wang. Nighthawk: Transparent system introspection from ring -3. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *Computer Security – ESORICS 2019*, pages 217–238. Springer International Publishing, 2019.

A Data smearing

Internal changes to introspected memory can affect the acquired result, e.g., pslist can return incorrect results if the newly linked process is only added to the list but not yet connected to the rest of the list. This is the *data smearing* problem and is a known limitation of asynchronous OOB systems. To reduce the probability of smearing, analysts can invoke the VMI application multiple times. Unfortunately, we are not aware of prior work that quantifies smearing impact on the detection probability. Therefore, we propose the following simple yet robust model to motivate the need for efficient introspection.

Assumptions. We assume malware infiltrated a VM and can mount single or multiple attacks successfully. Yet, each attack leaves traces in memory we call artifacts that can be used to detect the malware. Also, we assume the proper VMI application that can detect the specific attack is invoked continuously.

Model. Our model constructs a timeline of artifacts' availability in a monitored VM's memory. Each introspected artifact has a set of availability-window scopes. We denote the set of windows as a vector $T_m = < t_0, t_1, ..., t_N >$. Also, we assume the VMI application invocation latency is T_p. While more fine-grained models can be proposed, we opt for a conservative approach and assume the artifact is correctly detected iff the VMI application's execution completely overlaps the time in which the artifact is in memory.

We calculate the maximum number of introspection invocations per entry in T_m, which is equal to $<$

$\lfloor t_0/T_p \rfloor, ..., \lfloor t_N/T_p \rfloor >$. It is clear that if any of the introspection invocations is greater or equal to 2 the probability of detection is 100%, whereas if all are 0 the probability is 0%. However, if some of the introspection invocations are equal to 1 then the artifact is correctly detected if the VMI application does not execute at the window $[(t_i - T_p), T_p]$ as we assume continuous execution of the VMI application. Considering uniform distribution the probability of correct detection is thus $1 - (T_p - (t_i - T_p))/t_i = 1 - (2T_p - t_i)/t_i$ and the overall probability of a single detection is provided next and is equal to the complement probability of incorrectly detecting all artifacts. $P_{detection} = 1 - \prod_{t_i \in T_m} (2T_p - t_i)/t_i$

Case example: pslist. For pslist, an artifact is an element in the process list. A stealthy malware can evade detection by removing the process from the process list upon execution. While testing on our platform, we observed simple programs can have at minimum a total execution time of 1 millisecond or $T_m =< 0.001 >$. This means the detection rate is 100% when $T_p \leq 500$, and for $500usec \leq T_p \leq 1msec$ the detection rate decreases linearly.

Delta introspection modeling. We use the model to showcase that delta introspection can also be used to increase the probability of VMI applications' correct detection of malware. Specifically, assuming delta introspection latency is lower due to doorbell batching: denoted with αT_p with $\alpha < 1$. Thus, the overall probability of detecting an artifact grows to 100% iff executing both delta introspection twice followed by the VMI application and have all complete while the artifact is in memory. Formally, if $t_i/T_p \geq 1 + 2\alpha$ the detection probability is 100% rather than $t_i/T_p \geq 2$. Also, incorrect detection is minimized to the VMI application executing at a smaller window $[(t_i - T_p), 2\alpha T_p]$, and the equivalent detection probability becomes $P_{detection} = 1 - \prod_{t_i \in T_m} ((1 + 2\alpha)T_p - t_i)/t_i$.

B YARA rules

In this section, we describe the YARA rule used to evaluate the regex_scan VMI application in § 6.5, and our experience porting the YARA rule to a regular expression.

The YARA rule describes the Linux/Moose malware [5] and is available online [74]. Effectively, the rule searches all processes for the occurrence of all strings in a provided set. To translate it to regex a direct approach would be to use look-ahead, a construct that does not consume any bytes when matching and enables checking if a specified pattern exists after the assertion.

However, we find that this construct is not supported in our platform [50]. Thus, we opt to use logical or-based representation for the regex. One option is to create a list of all permutations of the strings separated by logic or constructs. Yet, the permutation grows quickly with the number of strings to match, e.g., in the Linux/Moose YARA rule there are 20 strings, which is too large for manual creation, and likely would create performance issues in a regex engine. Instead, we decide to use a more concise representation while using logical or as follows.

```
/Status: OK|--scrypt|stratum\+tcp:\/\/|cmd\.so
    |\/Challenge|processor|cpu model|password is
     wrong|password:|authentication failed|sh|ps
    |chmod|elan2|elan3|chmod: not found|cat \/
    proc\/cpuinfo|\/proc\/\%s\/cmdline"|kill \%s
    /
```

We propose to count the number of matches found for the above regex and validate it is at least as large as the number of strings. While this greatly simplifies porting efforts, it is insufficient for correct matching as a single string of the set may be matched 20 times. Thus, we follow regex matches with a software-based verification for each string in the set.

Note, we consider this a decent tradeoff as we do not expect many false positives in such large string sets. Yet, for smaller string sets the permutation option may be more suitable to overcome false positives and achieve increased performance.

C Malware Evaluation

The following list contains the SHA1 hash sums of all samples used in this work. They can be downloaded from online services such as MalwareBazaar [1].

- a9ef9f0bdd55fd931d909d71264c6587955ea361
- eb30a48246a86f39f0038ebf116a53a45dc3bf1a
- 258d3d3248c644d401262aed9ce62d1a1b620703
- 6d4dc35c0d70a8ca90b87c39e644db36ce29e232
- b4339e8c1b20145e2814724fc24606b5a3014a63
- 9c76a989dc66c52ca4253dbbd538f9db9d4b6d87
- 873c5df9f5854cf412406382eb19ec740d23efca
- 8734f34a073c6f344a864d5fc4d90871aa50d147
- 74b1da190d670fa4c207afb0fbca4d7df701538a
- 7cfbeae5eb73fd529ea219f655bcbb31a639814e
- 7f21f863f00c9bff182c3044ea225574b02107cf
- 4127607668ea5b2fa73365b16f636c45ad93eb99
- 0c203cafd5edde1f1af2686226089d241274b1a9
- f01b9af23aac2cb9eb4b7c82642d15533ccf6db1
- 9db360adc86b08b60ac07d9d2c7e3458ca184312
- 3d716441195624631b5d5e9c468b89afa8dcf4e2