Deconstructing Xen

Lei Shi*†, Yuming Wu*†, Yubin Xia*†, Nathan Dautenhahn‡, Haibo Chen*†, Binyu Zang†, Haibing Guan*, Jinming Li§
*Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University
†Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
‡Department of Computer and Information Sciences, University of Pennsylvania
§Huawei Technologies, Inc.
{*sleepytodeath@gmail.com, yumingwu233@gmail.com, xiayubin@sjtu.edu.cn, ndd@cis.upenn.edu
haibochen@sjtu.edu.cn, byzang@sjtu.edu.cn, hbguan@sjtu.edu.cn, lijinming@huawei.com*}

***Abstract*—Hypervisors have quickly become essential but are vulnerable to attack. Unfortunately, efficiently hardening hypervisors is challenging because they lack a privileged security monitor and decomposition strategies. In this work we systematically analyze the 191 Xen hypervisor vulnerabilities from Xen Security Advisories, revealing that the majority (144) are in the core hypervisor not Dom0. We then use the analysis to provide a novel deconstruction of Xen, called *Nexen*, into a security monitor, a shared service domain, and per-VM Xen *slices* that are isolated by a *least-privileged* sandboxing framework. We implement *Nexen* using the Nested Kernel architecture, efficiently nesting itself within the Xen address space, and extend the Nested Kernel design by adding services for arbitrarily many protection domains along with dynamic allocators, data isolation, and cross-domain control-flow integrity. The effect is that *Nexen* confines VM-based hypervisor compromises to single Xen VM instances, thwarts 74% (107/144) of known Xen vulnerabilities, and enforces Xen code integrity (defending against all code injection compromises) while observing negligible overhead (1.2% on average). Overall, we believe that *Nexen* is uniquely positioned to provide a fundamental need for hypervisor hardening at minimal performance and implementation costs.**

I. Introduction

Virtualization is one of the key enabling technologies of today's multi-tenant cloud. Through adding a privileged software layer (i.e., the hypervisor), virtualization can simultaneously support tens, hundreds, or even thousands of guest virtual machines (VMs) on a single server. However, as the number of concurrent VMs increases so too does the impact of a hypervisor compromise, i.e., any single exploit undermines all VM security.

Unfortunately, one of the most widely-used hypervisors, Xen [7], is highly susceptible to attack because it employs a monolithic design (a single point of failure) and comprises a complex set of growing functionality including VM management, scheduling, instruction emulation, IPC (event channels), and memory management. As Xen's functionality has increased so too has its code base, rising from 45K lines-of-code (LoC) in v2.0 to 270K LoC in v4.0. Such a large code base inevitably leads to a large number of bugs that become security vulnerabilities [31]. Attackers can easily exploit a known hypervisor vulnerability to 'jail break" fr om a guest VM to the hypervisor to gain full control of the system. For example, a privilege escalation caused by non-canonical address handling (in a hypercall) can lead to an attacker gaining control of Xen [13], undermining all security in multi-tenant cloud environments.

To understand the security threat to Xen, we systematically studied all 191 security vulnerabilities published on the Xen Security Advisories (XSA) list[1] [35], of which 144 (75.39%) are directly related to the core hypervisor. Among the 144 vulnerabilities, 61.81% lead to host denial-of-service (DoS) attacks, 15.28% lead to privilege escalation, 13.89% lead to information leak, and 13.20% use the hypervisor to attack guest VMs. Furthermore, we found that more than half of the core vulnerabilities are located in per-VM logic (e.g., guest memory management, CPU virtualization, instruction emulation).

While there has been much work aiming at improving the security of the virtualization layer [37], [12], [23], none of it has provided an efficient way to harden the Xen core. For example, CloudVisor [37] uses an 'but-of-the-box" approac h by introducing a tiny nested hypervisor to protect VMs from the potentially malicious Xen. Colp et al. [12] propose an approach to decomposing the management VM of Xen (i.e., Dom0) into multiple unprivileged domains while Nguyen et al. [23] propose Min-V, a hypervisor based on Microsoft's Hyper-V to disable non-critical virtual devices for a VM, reducing the attack surface. However, none of them aim at hardening the hypervisor itself. While DeHype [34] aims at removing KVM out of the globally shared trusted computing base (TCB), the hosted hypervisor, which includes a complete Linux, remains in each VMs TCB while being large and vulnerable (including all Linux vulnerabilities).

As our security analysis demonstrates, the Xen core is fundamentally at risk. However, it is unclear how to effectively mitigate these threats. Therefore, we present *Nexen*, a novel deconstruction and reorganization of Xen that separates and confines hypervisor operations. The design of *Nexen* is inspired

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23455

[1]The actual number is 201, but 7 of them are not used, XSA-166 is too vague to be counted in our study, XSA-161 was withdrawn, and XSA-99 is irrelevant

by the principle of least privilege [24] and informed directly by our vulnerability analysis. We decompose Xen into multiple internal domains (iDoms): a privileged security monitor, one shared service domain, and multiple per-VM Xen *slices*. A VM-slice contains a subset of duplicated Xen code and per-instance private data. In this way, a malicious slice cannot directly access data within guest VM address spaces, and a malicious guest VM cannot affect other VMs or the host system, even if it has compromised the per-VM slices.

In addition to deconstructing Xen we also address the core challenge of efficiently enforcing separation, a unique issue for Xen because the hypervisor operates at the highest hardware privilege level. *Nexen* achieves this through same-privilege memory isolation [14], [4] to enforce cross-VM data and control-flow integrity. Specifically, *Nexen* extends the privileged security monitor from the Nested Kernel architecture, to isolate and control the memory management unit (MMU), which mediates all memory mapping updates to provide high level security policies. *Nexen* extends the Nested Kernel by adding secure and private memory allocators, multi-slice support, secure slice control transfers, and private and shared slice data control: in this sense a slice is analogous to a lightweight process.

We have implemented a prototype of our design which mitigates 107 out of 144 vulnerability (74%). Evaluation results also indicate that the performance overhead is negligible.

Our contributions: To summarize, this paper makes the following contributions:

- A systematic analysis on 191 Xen vulnerabilities (Sections II and V).
- *Nexen*, a novel deconstruction of Xen into a security monitor, shared service domain, and sandboxed per-VM slices (Section III) implemented in Xen (Section IV) that efficiently uses paged based isolation mechanisms for fine-grained data isolation.
- As informed by the analysis, a novel least-privilege decomposition strategy that places highly vulnerable code into per-VM slices while maintaining high performance and either eliminating vulnerabilities entirely or confining exploits (evaluated in Section V).
- Efficient code, memory, and control-flow integrity enforcement between Xen and VMs (evaluated in Section VI).

II. Motivation and Background

A. Attack Surface of Xen

The Xen virtualization layer comprises the Xen hypervisor, a privileged VM (i.e., Dom0) and a number of unprivileged VMs. Each of these can be compromised in one of the following ways: 1) an unprivileged VM may attack another VM through inter-domain communication (mostly shared memory); 2) a malicious platform user may compromise Dom0 through the management interface, resulting in control of all management operations and I/O stacks of other VMs; and worst of all, 3) an unprivileged VM may attack the hypervisor through vulnerable hypercalls or buggy code emulation, fully compromising all security on the system.

TABLE I. Xen modules that the attacks target

Target	Ratio	Target	Ratio
Memory management	25.69%	Domain control	4.17%
CPU virtualization	21.53%	Domain building	3.47%
Code emulation	13.19%	Event channel	2.08%
I/O	9.03%	XSM	1.39%
Exception handling	5.56%	Scheduler	0.69%
Grant table	4.86%	Others	3.47%
Global	4.17%		

TABLE II. Key steps of vulnerability

Key Step	Ratio
Memory corruption	45.14%
Misuse of h/w feature	22.22%
Live lock	8.33%
Infinite loop	6.25%
False BUG_ON	6.25%
General fault	4.86%
Run out of resource	4.17%
Dead lock	3.47%

TABLE III. The results of different attacks

Result	Ratio
Host DoS	61.81%
Privilege escalation (to host)	15.28%
Info leak	13.89%
Guest DoS (self)	6.25%
Guest DoS (other)	2.78%
Privilege escalation (to guest kernel)	3.82%

In this section we summarize our investigation of Xen attacks as they relate to the target code module, vulnerability steps, and high level compromise result. Our results are derived from analyzing the Xen Security Advisories (XSA) vulnerability database, which lists 191 discovered vulnerabilities between early 2011 to the middle 2016. A comprehensive evaluation and analysis of these results as well as how *Nexen* defends against them is presented in an online appendix located at http://ipads.se.sjtu.edu.cn/xsa/ [1].

A large portion of such vulnerabilities (75.39%) are related to the hypervisor. They either directly target the hypervisor or aims at VMs but take advantage of bugs in the hypervisor. Other ones (24.61%) are mostly flaws in QEMU and tool stack, which reside in Dom0. Since the latter ones can be effectively mitigated by disaggregating drivers and domain management tools, e.g., using different driver domains and management domains for different guest VMs, we focus on vulnerabilities related to the hypervisor in this paper.

We classified these vulnerabilities in three different ways. The first way is based on target, i.e., the functionality module where the exploit happens. Table I presents the distribution of vulnerabilities. We can see that I/O, memory management and CPU virtualization(including code emulation) are the most dangerous modules, while modules like scheduler and event channel has nearly no known vulnerabilities.

The second way is based on the result that a vulnerability may cause. We can observe from the Table III that most of these vulnerabilities cause host DoS, information leakage or privilege escalation to the hypervisor.

Table II shows the result of our third way of classification based on the key step of exploiting a vulnerability. Here live lock means long non-preemptible operation can be performed

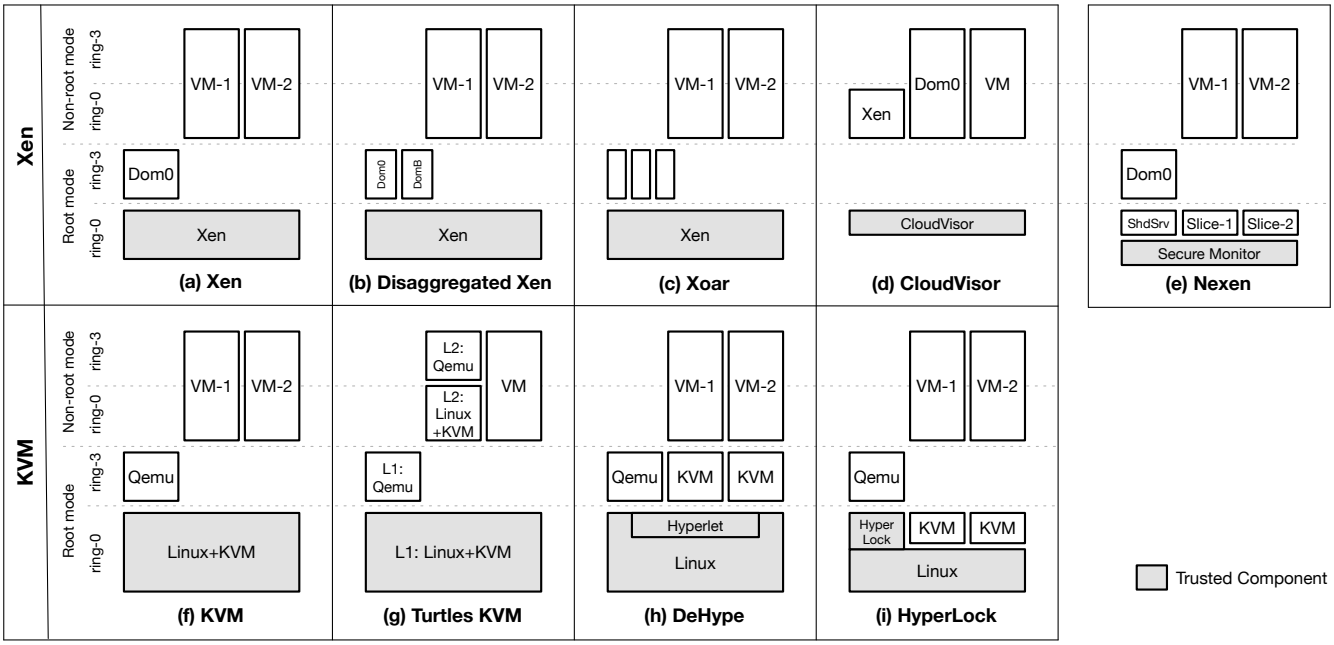

Fig. 1. Comparison between hypervisor reorganization approaches for both Xen and KVM.

TABLE IV. COMPARISON ON ATTACKS DEFENDING.

	Hypervisor illegally accesses VM's data	Guest causes host DoS	Guest application hacks its own VM by hypervisor
Disaggregated Xen [22]	No	No	No
Xoar [12]	No	No	No
Turtles KVM [9]	No	Yes	No
DeHype [34]	No	Yes	No
HyperLock [33]	No	Yes	No
CloudVisor [37]	Yes	No	Yes
Nexen	Yes	Yes	Yes

without rate limiting. An observation is that most vulnerabilities simply cause a CPU hanging or a fault that will kill the host. Although many vulnerabilities can cause memory corruption, their affecting ranges are usually very limited. As shown in the above table, only a few of them have the potential to eventually achieve privilege escalation.

Another key observation is that, although most of the catastrophic vulnerabilties and CPU hangings can be caught, in most cases, the handler still has to kill the entire host instead of recovering. The main reason is that the hypervisor lacks the precision to identify individually corrupted components.

Overall, we believe that a reliable isolation mechanism with the ability to limit the privilege of each part of the hypervisor can effectively prevent most attacks, which we demonstrate in the rest of the paper.

B. Previous Solutions

Prior research has explored various hypervisor hardening techniques. Figure 1 classifies core related efforts according to their platform (Xen or KVM), trusted components, and the hardware privilege layer each component resides: *ring-0* or *ring-3* and *root mode* or *non-root mode*.

The top half of the figure shows the architectures securing Xen. Xen has a Dom0, which is a privileged para-virtualized VM that is responsible for I/O operations[2]. VMs run in *non-root mode* without modification, a.k.a., hardware-assisted virtual machine (HVM). Disaggregated Xen [22] decomposes Dom0, moving all the code for building a guest VM to a separate VM named DomB ('B" for 'Builder'). Thus any vulnerabilities of the domain builder can be isolated within the VM boundary without affecting other VMs or the host system. Xoar [12] takes a further step by decomposing Dom0 into 7 different kinds of VMs, each focusing on just one functionality, to achieve better fault isolation and smaller attack surface. CloudVisor [37] targets a different goal: to protect the guest VMs from a malicious hypervisor. It leverages nested virtualization that puts Xen and Dom0 in *non-root mode* so that all privileged operations will trap to CloudVisor for security checking. CloudVisor can effectively defend against attacks leveraging the hypervisor's vulnerabilities to attack the guest VM, e.g., in-guest privilege escalation. Most of these systems focus on isolating Dom0's vulnerabilities, but none of them can defend against host DoS attacks through Xen exploits.

The bottom half of the figure shows hardening of KVM. Unlike Xen, KVM is a kernel module in Linux. It only handles hardware events generated by the CPU, leaving most of the resource management (like the virtual CPU scheduling and memory management) to the Linux kernel. Qemu emulates devices at user-level. The Turtles project [9] has implemented nested virtualization for KVM that can run guest VMs inside a guest VM as a sandboxing mechanism. Xen has since added support for nested virtualization as well [16]. Although Intel keeps updating its processor to have better support for nested virtualization [11], performance overheads are still non-negligible. DeHype [34] and HyperLock [33] decompose

[2]Dom0 typically runs in *ring-3* in x86-64 and *ring-1* in x86-32. Here we only consider x86-64.

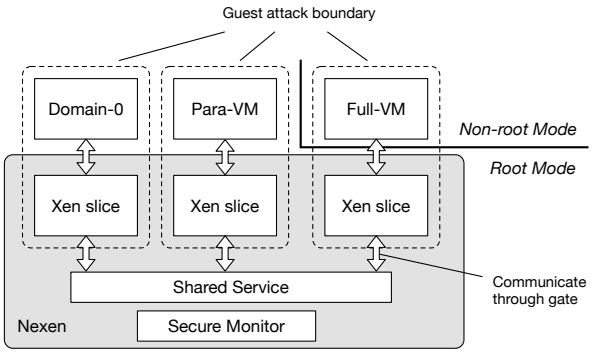

Fig. 2. The architecture overview.

KVM by creating a KVM instance for each guest VM. As a result guest VMs can only impact (e.g., crash) its own instance. DeHype puts the KVM instances in *ring-3*, resulting in high performance overhead. HyperLock implements an in-kernel isolation mechanism that enables different KVM instances running within *ring-0* to reduce the performance overhead while still retaining isolation.

III. Design

The primary goal of *Nexen* is to harden Xen against various security threats. This is challenging because Xen operates with ultimate system authority: there is no privilege layer to enforce the hardening. Our key idea is to deconstruct Xen into separate protection domains that apply the principle of least privilege and to do so at a single privilege level.

In this section we overview *Nexen*, present our technique to obtain single privilege layer isolation, describe the isolation services that enable least-privilege, and present our decomposition strategy for informed separation.

A. Nexen *Overview*

The *Nexen* architecture (Figure 2) decomposes the monolithic hypervisor into a minimal, fully privileged security monitor, *monitor*, a less privileged *shared service* domain, and fully sandboxed *Xen slices*. All these domains run in the highest privilege of the system, i.e., ring 0 of the root mode. The core challenge of doing this at a single privilege layer is obtaining a tamper-proof protection mechanism with which to enforce isolation within Xen. To do so we utilize and extend the Nested Kernel Architecture design [14] to isolate the security monitor while operating in root-mode.

Nexen uses the isolated security monitor to control all updates to the MMU. By controlling the MMU *Nexen* can guarantee isolation between internal domains and manage privileges. With carefully designed policies, *Nexen* can ensure each internal domain has only the necessary privileges, which significantly reduces the attack surface of the whole system.

The next challenge *Nexen* considers is to devise a valuable deconstruction of Xen. In our security analysis we observed that many vulnerabilities are localized to specific units of functionality in Xen. If we sandbox this functionality then we would be providing valuable security enhancement. This is similar to the *device driver* isolation literature, where highly susceptible code is sandboxed [38], [26].

TABLE V. Attacks considered & not considered by *Nexen*.

Malicious Component	Steal or Tamper with VM's Data	Guest DoS	Host DoS
VM (User)	N.A.	Considered	Considered
VM (Kernel)	Not Considered	N.A.	Considered
Other VM	Considered	Considered	Considered
Xen Slice	Considered	Not considered	Considered
Shared Service	Considered	Not considered	Not considered

Therefore, *Nexen* decomposes Xen into per-VM slices that are naturally sandboxed from all other components in the system. Each Xen slice is bound to one VM and serves only this VM. VMs will only interact with their own Xen slice during runtime. Xen slices share code but each has its own data. They are the least privileged internal domains, and errors in one Xen slice are not considered dangerous to the whole system or other VMs.

Unfortunately, a simple decomposition of all functionality into slices is untenable because subsets of functionality interact across slice boundaries. High frequency privilege boundary crossing cause high performance degradation. So we create a single, slightly more privileged shared service domain ~~b~~ ut still not as privileged as the security monitor. Deciding what to place in per-VM slices and the shared services domain is non-trivial and one of the key contributions of this work.

In the following sections, we first introduce the design of the core security monitor. Then we describe how to build internal domains based on the security monitor, along with their interfaces and properties. Finally, we show how to deconstruct the hypervisor to minimize its vulnerability.

B. Assumptions and Threat Model

We consider that an attacker can take full control of a user application running in a guest VM, and tries to gain higher privilege or issue DoS attack by exploiting the hypervisor and its own OS. We consider the attack against hypervisor through its flaws. We also consider the attack against guest OS *through the hypervisor's vulnerability*, but not through the guest OS's own bugs.

We also consider that an attacker can deploy a complete malicious guest VM on the virtualized platform, and try to attack the hypervisor to further attack other VMs and the entire platform through illegal data accessing or DoS attacks.

The Xen slices and shared service are not in the TCB of our system. Even if they are compromised, they cannot illegally access guest VM's data. However, they can issue DoS attack. Specifically, a Xen slice can just stop serving its own VM, while the shared service may crash the host by disabling scheduling. However, we do not consider physical attacks as well as side channel attacks between different VMs. A complete threat model matrix is listed in Table V.

C. Isolating the Monitor

The monitor is the most fundamental element to *Nexen* protections. If the monitor is compromised all security in the system is lost ~~th~~is is true of any protection system. The

monitor must therefore be tamper-proof without creating high overheads or forcing large changes to the Xen code base.

Instead of deprivileging Xen by moving it into Ring-3 we use nested MMU virtualization [14], which nests a memory protection domain within the larger system at a single privilege level. The benefit is that *Nexen* creates minimal performance degradation and modification to Xen while gaining a tamper-proof monitor. Nested MMU virtualization works by monitoring all modifications of virtual-to-physical translations (mappings) and explicitly removing MMU modifying instructions from the unprivileged component ~~a~~ technique called code deprivileging.

Nexen virtualizes the MMU by configuring all virtual address translations (mappings) to page-table-pages as read-only. Then any unexpected modifications to the page tables can be detected by having traps go directly to the monitor. Further, accesses to the MMU through privileged instructions must be protected. This includes accesses to CR0 controlling the paging and to CR3 controlling the address spaces. *Nexen* removes all instances of such operations from the deprivileged Xen code base such that there are no instructions that can modify the MMU state: we validate this assumption by performing a binary scan to ensure no aligned or unaligned privileged instructions exist. The last element is to ensure that none of the Xen components inject privileged instructions. Because the monitor has control of the page-tables it can easily enforce code integrity and data execution prevention.

By restricting control of the MMU to the monitor, *Nexen* greatly reduces the TCB for memory management and isolation services. It also enables the monitor to control critical privileges in Xen including properties like code execution and entry gates.

D. Intra-Domain Slicing

The primary goal of *Nexen* is to enhance Xen's security by deconstruction. To do this *Nexen* provides the core abstraction of a *slice* to represent internal domain. *Nexen* extends the monitor to provide a set of basic functionality that is required to securely create, manage, and permit interactions between internal domains. As shown by our vulnerability analysis, isolation and minimizing privileges are effective ways to limit the attack surface and control the damage.

Nexen enables two types of internal domains. The global shared service and per-VM Xen slices. Components like the scheduler and domain management are placed in the shared service while functions related to only one VM, e.g., code emulation and nested page table management, are *replicated* to each Xen slice.

1) Internal Domain API: Nexen provides an internal domain to the shared service for the management of slices. Shared service and the monitor are built as the system boots. All Xen slices must be built by explicitly calling the following interfaces provided by the monitor.

- `void* nx_create_dom(int dom_id)`
- `void nx_destroy_dom(int dom_id)`
- `void* nx_secure_malloc(size_t size, int owner,int policy)`
- `void nx_free(void* p)`

These interfaces are open only to the shared service which is responsible for building new domains. Since we use memory mapping based isolation mechanism, each domain has its own address space. `nx_create_dom()` will create a new address space for the specified new domain and return the address of its root page table. This is called during guest VM booting. `nx_secure_malloc()` is used to assign a memory region for storing an internal domain's own data. An owner will have full read/write access to the data. Access permissions exposed to other domains are specified by the 'policy' argument. The other two functions are simply the reverse operations for them, used during the shutdown or force a destruction of a domain.

2) Controlling Memory Access Permissions: Memory access permission is the first class of privileges controlled by *Nexen*. Memory regions are mapped differently in different internal domains so that one internal domain can only see/modify what's safe for it to see/modify. Mappings in internal domains are initialized during domain building and updated later using a tracing mechanism in the monitor if necessary. monitor controls all MMU updates to make sure no internal domain could break the isolation and violate memory access policies by modifying page tables.

The memory access permissions are presented in Figure 3. Shared service is allowed to manage memory access permissions through the `nx_secure_malloc()` interface. Various policies are available for different purposes. For example, during booting, shared service will declare its own data as invisible to all slices. When building a new Xen slice, its inner data is "granted" to it, which means they can be modified only in this internal domain. By default, everything inside a guest VM should not be visible to any internal domain.

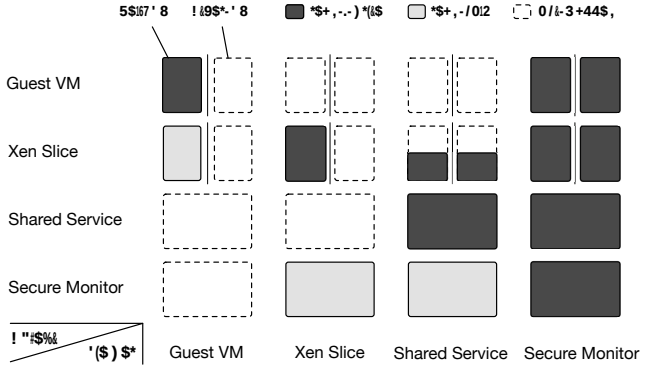

Fig. 3. Memory access permissions of different components of *Nexen*.

The security monitor does not have its own address space. It is shared by all internal domains. Every domain can directly request its services, but no one is able to tamper with the monitor's inner data. Shared service and every Xen slice have their own address space. Shared service has its own piece of data and code. Xen slices share the same code while each has its own data.

3) Controlling Privileged Instructions: Privileged instruction is the second class of privilege controlled by *Nexen*. Many special instructions may potentially violate the memory access policies or even directly harm other internal domains than the

caller. Execution of these instructions must undergo careful examination. We treat them as "privileged" instructions.

Nexen limits the instruction set each internal domain can execute to avoid abuse of privileged instructions. No direct execution of privileged instructions is allowed outside the security monitor. Internal domains request the monitor to execute the instructions for them. The monitor enforces a sound sanity check to prevent unauthorized use and malicious use of these instructions.

4) Control Flow: To support the interaction between internal domains, *Nexen* provides a secure call gate for domain switching:

- `nx_entry(int domain_type, int dom_id)`

The *domain_type* argument specifies whether the switch target is a Xen slice or the shared service. When domain type is Xen slice, 'dom_id' shows the ID of the target Xen slice.

The call gate guarantee the following features:

- **Non-bypassable**: It is impossible to switch to another internal domain without calling the gate.
- **Unforgeable**: Any call gate not called from its expected position is not accepted. Each call gate is bound to both its return address and its `domain_type` argument, which must be hard-coded.
- **Atomic**: Switches of control flow and address space are atomic. Any attempt to switch to an internal domain's address space while redirecting the control flow to another one will fail the return address check and be rejected.

Even if the attacker has successfully exploited a vulnerability leading to privilege escalation, she will only gain full control inside one internal domain, which normally is a Xen slice. To do any meaningful attack, the attacker must try to intrude into another domain through the call gates, which is much harder than the initial exploit.

We carefully restrict control flows to minimize the possibility for internal domains to attack each other.

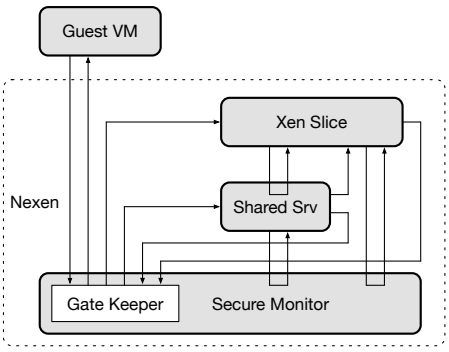

Fig. 4. The control flow between components in *Nexen*.

There is a small piece of code called *gate keeper* that controls all exits from VMs to the hypervisor and entries from the hypervisor to VMs. On entering the hypervisor, the gate keeper records the trap frame and dispatch the control flow to the correct internal domain. On exiting the hypervisor, the gate keeper checks VMCS and other necessary running states to make sure no policy is violated. If the reason of entering the hypervisor is an interrupt, the control flow will be transferred to the shared service. The shared service will deal with the interrupt itself or dispatch it to a VM and exit to current VM's Xen slice. For other reasons, mainly a hypercall or a code emulation, the control flow directly goes to current VM's Xen slice.

A Xen slice usually deal with the request itself. It may use some services provided by shared service or security monitor. The control flow will always return to the Xen slice except for scheduling. Eventually, the control flow goes to the gate keeper to return to the VM.

We explicitly forbid switching between Xen slices since no such a need exists. This eliminates the possibility that a malicious Xen slice can directly intrude into another Xen slice or further attack the bound VM. The only way to switch control flow from one VM and its Xen slice to another is scheduling. Since the scheduler doesn't receive any input generated by a guest, and the context after the switch is decided by the target VM and its Xen slice, an attacker can hardly intrude through this way, which is also evidenced by the scheduler's low vulnerability ratio even in unmodified Xen.

Communication between Xen slices and shared service is strictly limited to prevent a malicious Xen slice from attacking the shared service. The only situations where control flow is allowed to transfer from a Xen slice to shared service are discussed in the next subsection. We'll show that all these transfers are secure enough and can hardly be used as an attack surface.

E. Decomposing into Slices

The monitor and slicing services provide powerful isolation and code integrity properties for Xen. They form the foundation for *Nexen* to apply security relevant partitioning of Xen.

Our deconstruction strategy is to apply the principle of least-privilege: minimize the authority of each domain in the system. The best solution would be to create a complete Xen slice for each VM. Unfortunately, there is functionality that interacts across much of the Xen system. Another challenge is to select partitions that minimize the interface between the shared service and per-VM slices to minimize API abuse. Significant data structures must also be wisely arranged to avoid destructive corruption.

As such *Nexen* must deconstruct Xen in a way that intelligently partitions functionality ~~m~~aximize the val ue of least-privilege while minimizing cross-domain interactions. To manage this we identify functionality that is shared and place it in the *Nexen* shared service domain, which operates at a slightly higher privilege level than per-VM slices. The higher privilege enables special data and cross-domain calling privileges solely for the shared service domain.

Another high level idea of our strategy is to derive our partitioning from the vulnerability analysis. Much like device driver isolation was motivated by the high degree of vulnerability of drivers, we identify the Xen functionality that is most likely to be corrupted and place that into the per-VM slices.

On the other side we identify core components that must be in the shared service domain due to the nature of their operations.

To sum up, we follow three principles in decomposition work to enhance the security of the system. The first one is to avoid inserting dangerous functionalities into the shared service, which is very intuitive. The second one is to avoid runtime communication. Components in shared service can be safe even if it contains relatively more vulnerabilities as long as guests are not able to invoke them actively during runtime. The third one is to separate mechanism from policy. Complex calculation and decision making can be processed by untrusted code. But the final security sensitive operation must be performed by trusted code. This principle can effectively reduce the size of trusted code and alleviate the burden of sanity check.

1) Significant Component Decisions: In decomposing Xen there are several significant decisions to make either because the functionality is pervasive or it is a highly vulnerable component. In this section we provide further analysis of our deconstruction. The Scheduler, Memory Management, and Event Channels are all or partly placed in *Nexen* shared service. Due to its high complexity and vulnerability, Code Emulation and I/O are split across per-VM slices. Note that this is not an exhaustive list.

Scheduler determines which VCPU currently runs on one CPU. Each VCPU has a credit used to calculate its priority. It burns as a VCPU runs. Each CPU has a runqueue recording all VCPUs and their priority. The queue is periodically sorted to maintain the priority order of VCPUs. There are also some global parameters controlling the speed of burning credits, the rate limit of scheduling and other issues. When a scheduling operation happens, the VCPU with the highest priority is picked to run in next round.

We can observe that all significant data is naturally closed for scheduler's inner use. Credit burning, runqueue sorting and scheduling are all triggered by timer interrupts without any interference from guests. Further, CPU is usually shared between VMs and is not suitable to be assigned to any Xen slice. Hence, we put the whole scheduler inside the shared service. No Xen slice is allowed to modify the state of the scheduler.

There are occasions where a guest wants to yield, sleep or wake its VCPU. We open these three interfaces to Xen slice. Their only input is the VCPU pointer used to find its corresponding data structure inside the scheduler, whose validity will be checked on the shared service's side. Nothing changes except for the guest's own VCPU's existence in the runqueue after these operations. Hardly can any malicious data be delivered through this interface, nor can any dangerous behavior be performed.

Event Channel delivers various events between guest VMs, the hypervisor and hardware. Each VM has its own event channel bucket, which contains a number of event channel ports. A port can be bound to an interrupt or another VM's port. VMs maintain their own buckets while the hypervisor helps to deliver the events.

Since event channel buckets are tightly bound to VMs, we put each VM's bucket in its own Xen slice to avoid abuse from other VMs. Interrupts related to one VM will be forwarded to its Xen slice, so delivery of events bound to interrupts are done by Xen slices. When a VM send event to another VM, it will have difficulty writing the pending bit in the target VM's port for the lack of writing permission. We proxy this request through the shared service, which is another interface open to Xen slices. This request is safe enough because the only information provided by the sender is the target VM's port number, which is easy to examine.

Memory Management contains everything related to memory operation, mainly memory allocation and memory mapping update. The core data structures are page tables and a `page_info` region maintaining the usage state of each physical page. Allocator's free page list is built based on the linked list field in `page_info` structure. Other information in `page_info` is referred to and updated during memory mapping update.

Allocator is only used during booting and domain building, so we keep it in the shared service and do not expose any interface to Xen slices. Memory mapping update is mostly related to only one VM. Xen slices are allowed to manage their bound VM's memory mapping update. One challenge here is that `page_info` region is required by both functionalities, each in a different type of internal domain. Fortunately, memory region of Xen and each VM has a clear boundary. We map the region in every internal domain, but only grant to each Xen slice the writing permission of their own page information. Both functions work nicely in this way. Apart from that, no internal domain has the permission to write page tables. After making the page table update decision, they must request the monitor to perform the operation. Such updates are carefully checked to make sure no policy is violated.

Code Emulation and I/O are related to CPU features more than memory. These parts of the hypervisor provide virtual CPU and devices for VMs by catching and emulating VM's privileged operations. The emulation code sometimes runs the privileged instructions itself to get necessary data. This whole process is extremely error-prone. Attackers may directly cause an exception, corrupting memory in another module, or steal sensitive data from other VMs through the misuse of hardware features.

We run the emulation code in each VM's Xen slice and grant VM's VCPU running state to the Xen slice. Although the misused hardware features are largely out of our control, attempt to corrupt other parts of the system must go back to memory. Even if the attacker has successfully raised her privilege to that of the host, *Nexen* can still enforce the memory access policies. The attacker will be isolate in the Xen slice and achieve nothing worthwhile. If the attacker chooses to trigger a deadly exception, the handlers are modified so that only the Xen slice, instead of the whole system, is destroyed. We are able to do this because *Nexen*'s memory isolation gives the guarantee that no crucial data are corrupted during the exception.

IV. *Nexen* Implementation

We implemented a prototype of *Nexen* based on Xen version 4.5 for Intel x86-64 architecture. This section describes how we address three main technical challenges: First, *how*

TABLE VI. CONTENT OF SHARED SERVICE

Shared service content
Scheduler
Allocator part of memory management
Interrupt handlers
Domain building
Event delivery of event channel

to enforce inter-domain isolation and memory access policy? Second, *how to control privileged instructions?* Third, *how to monitor the interposition between the hypervisor and VMs?*

A. Isolation between Internal Domains

We used an isolation mechanism based on memory mapping. Each internal domain resides in its own virtual address space. The permission bits in page table entries are set according to memory access policies. In this way, *Nexen* controls what every internal domain is allowed to read and write.

To achieve this, *Nexen* interposes memory mapping updates and enforces a set of carefully selected invariants to provide flexibility in applying policies on any memory region. Additionally, *Nexen* allows interaction between internal domains while retaining the isolation.

1) Control Memory Mapping Update: Nexen maps all page-table-pages as read-only in all address spaces and enables the Write Protection (WP) bit, forbidding mapping updates. The security monitor's internal data is also mapped read-only to avoid modification from malicious domains. As control flows into the monitor the WP-bit is flipped so that the monitor can update page tables and maintain its internal data structures. On exiting the monitor, WP-bit is enabled. Interrupts are disabled while executing in the monitor to make sure no entity can hijack the control flow when the WP-bit is turned off. In this way, the monitor completely controls all mappings. Memory mapping updates in internal domains are replaced by calls into the monitor, but the logic for their management remains in the domain.

2) Enforcing Memory Invariants: Before each memory mapping update, the security monitor needs to do sanity checking to enforce certain invariants. These invariants are independent of policies and have the highest priority in all rules. They keep the memory layout and the most significant data structures of the system intact in any condition. The invariants of each type of memory are shown in Table VII.

The monitor maintains an internal data structure recording the usage of each physical page. Invariants are mostly based on the memory page's usage type and owner.

3) Enforcing Memory Isolation Policies: To provide a flexible way to protect internal domains' data, memory of Protected Data type can have various policies. The monitor has a special allocator inside itself for allocating protected data, through which policies can be specified for each memory region. The allocator has an inner memory pool recording the address, size and policy related information for every allocated and free memory region. When building a domain, the shared service will ask for memory from the allocator for the new Xen slice's data structures. The policy is given to the monitor along with the request. All the information is recorded in the memory pool. When the new address space is eventually created, the monitor will traverse the memory pool and modify mappings for recorded memory regions according to their policy. As is described in the invariants, mappings for protected data will not be changed once the owner VM has started. So no one can trick the monitor into exposing other domain's data.

Frequently used policies are:

- **Grant**: Data is granted to a Xen slice. They can be modified by its owner Xen slice. In other internal domains they are mapped read-only.
- **Half Grant**: Data is granted to a Xen slice. They can be modified by the owner Xen slice and the shared service. In other Xen slices they are mapped read-only.
- **Grant and Hide**: Data is granted to a Xen slice. They can be modified by owner Xen slice. In other internal domains they are unmapped.
- **Limit**: Data is granted to the shared service. In all Xen slices they are mapped read-only.
- **Limit and Hide**: Data is granted to the shared service. In all Xen slices they are unmapped.

4) Securing Call Gate: Nexen provides a gate allowing the switch between internal domains. It is a function call with the target internal domain's type and id as arguments. The function itself switches the address space to that of the target domain. The return address determines the target domain's entry point.

To make sure the switches of control flow and address space happen atomically, we turn off interrupt during the execution of the gate and check the validity of the function call's return address. All gates are placed at fixed places in the code. Calling through a function pointer is forbidden. All valid return addresses for an internal domain are collected with the help of the compiler and stored in a table. At each call of the gate, the return address is searched in the table. Only if it is a valid one recorded in the table will the address space switch be executed. The table is sorted and search with binary search to speed up the process.

B. Confining Privileged Instructions

To avoid unexpected violation of isolation, *Nexen* must control the execution of some extremely sensitive instructions. *Nexen* achieves this with two methods: "monopolize" and "hide". Internal domains will not have direct access to such sensitive instructions. Similar to memory mapping update, execution of these instructions must be forwarded to the security monitor, which gives it the chance to perform careful check and manipulation to enforce instruction related invariants that keep the isolation intact.

A "monopolized" instruction only has one presence within the monitor's code. Under this constraint, the instruction is still visible to internal domains. The attacker have to either call the monitor's interface or directly jump to the instruction if she as successfully hijacked the control flow. If the bad consequence of the instruction does not occur immediately after the execution, the monitor could do sanity checking after the instruction and fix the misuse.

TABLE VII. PROTECTION INVARIANTS FOR DIFFERENT TYPES OF MEMORY PAGES

Page Type	Protection Invariants
Guest Memory	These pages belong to a VM. They are invisible to the whole hypervisor but can be accessed by their Xen slice, which can change their mapping and type when the VM wants to exchange pages with the hypervisor.
Sensitive Guest Memory	These pages contain guest VM's secret data. They are invisible to the whole hypervisor and cannot be accessed by even its own Xen slice. They are declared by the guest using a special hypercall and cannot be changed by the hypervisor.
Host Code	These pages contain the code of the hypervisor. They are initialized during booting and will never change. They are the only non-user pages with execution permission and should always be mapped read-only.
Monitor Data	These pages contain the security monitor's inner data. They are always read-only to internal domains.
Protected Data	These pages contain internal domain's own data. Their mappings are initialized according to the policies applied to them. They are initialized during owner's domain building and will never change.
Page Table	These pages contain the hypervisor's page tables. They can be declared and undeclared by the shared service. They are always mapped as read-only. Their type will not be changed unless explicitly undeclared.
Nested Page Table	These pages contain nested page tables describing guest physical to host machine memory mapping. They can be declared and unde- clared by owner's Xen slices. They are always mapped as read-only. The owner's Xen slice can request to update their content. Rules for this update is relatively simple: pages of other VMs and the whole hypervisor except for its Xen slice are always invisible.
Others	Other trivial and unused pages are described by this label, which *Nexen* provides no special protection for.

"Hide" is based on "monopolize". It takes a step further and unmaps the only presence of the instruction. Only when the instruction is used and the sanity checking is passed will the page be mapped. After the execution, the instruction will immediately be unmapped again to avoid abuse. If the execution of the instruction causes unwanted consequence at once, it should be hidden. Under this condition, the attacker have to go through the whole process of entering the monitor and sanity checking for the instruction since she does not have the privilege to update memory mapping necessary for the instruction to appear. A malicious execution of the instruction will not pass the sanity check.

We did a binary code scanning to make sure such instructions, aligned to instruction boundaries or not, do not exist in unwanted code region. To prevent an attacker generating new privileged instructions, we must guarantee the integrity of hypervisor's code. The invariants in the memory protection part has already guaranteed that code section is always read-only and no new kernel mode code mapping is allowed. Instead of directly modifying the code section, an attacker may want to generate code using data region or guest VM's memory region. To block these two bypasses, we explicitly forbid the execution of user code in privileged context. Table VIII shows privileged instructions and invariants.

C. Interposition between VM and Xen

Apart from enforcing isolation inside the hypervisor, the monitor plays the role of a gate keeper between guest VMs and the hypervisor. All interpositions between VMs and the hypervisor are monitored to enhance bidirectional security.

The monitor dispatches the event to a proper internal domain when VMs trap to the hypervisor. Once a guest VM is running and a VMExit occurs, the CPU will trap to the monitor first, which will check the VMExit reason. If the VMExit is VM related, e.g., instruction emulation or hypercall, the monitor will transfer control to the corresponding Xen slice to handle it. For other reasons like timer interrupt, the monitor will transfer control to a shared service component like the scheduler. Once the VMExit has been handled, the handler will transfer control to the monitor, which will eventually resume the execution of guest VM.

If a VMExit handler needs to access some data of the guest VM as auxiliary information, e.g., the instruction to be emulated, it will also call the security monitor module which will then access VM's data and check if it is OK to be retrieved by the handler. A Xen slice can update VMCS arbitrarily. When returning to a guest VM, the VMCS will be checked against a list of fields allowed to be modified for the certain VMExit reason. Unnecessary updates are rolled back before resuming the guest VM.

D. *Export* Nexen *to Other Platforms*

Nexen can be exported to any other platform with memory mapping mechanisms similar to x86's. Memory related policies, invariants and design decisions are independent of platforms. They can mostly be reused. Control instructions and control registers are specific to x86 platform. The system in the new platform must find alternatives to following features: controling memory access permissions of the highest privilege level, forbidding arbitrary code generation and execution, capturing all interrupts, exceptions and interpositions between the hypervisor and VMs. Instructions related to these features, along with any MMU updating instructions, should be considered privileged intructions and be protected.

V. SECURITY ANALYSIS

This section presents a security analysis on how Nexen can ensure security isolation and defend against exploits on each category of security vulnerabilities.

A. Security Isolation

An attacker gaining control of a Xen slice may try to undermine the isolation enforced by Nexen in four ways. Yet, none of them will succeed:

Escalating memory access privilege: Either writing protected memory region directly or writing page table to gain access to protected memory will result in a page fault. *Nexen* will kill the attacker's Xen slice and VM in this case. If an attacker tries to intrude through the secure call gate, she will either lose execution control or fail to gain the desired permission due to the sanity checking enforced by the monitor.

Abusing privileged instructions: Since privileged instructions have been removed from the per-VM slices, the attacker has to reuse those in the security monitor. If she normally calls monitor's interface, the malicious behavior will not pass the sanity checking. If she forges a malicious context and directly jumps to the instruction, the attacker will lose execution control for a while before exiting the monitor. This is because instructions that can immediately hijack the control flow are all

TABLE VIII. INSTRUCTION PROTECTION INVARIANTS.

Instruction	Protection Invariants
MOV CR0	These pages belong to a VM. They are invisible to the whole hypervisor but can be accessed by their Xen slice, which can change their mapping and type when the VM wants to exchange pages with the hypervisor.
MOV CR3	By this instruction, an attacker can change the whole address space and will probably redirect the control flow. Considering that, we hide this instruction and forbid any use of this instruction except for secure call gates and context switch. The target page table base address must point to a declared root page table.
MOV CR4	SMEP bit in CR4 forbids the execution of user code, which is crucial for code integrity. Considering this instruction will not directly hijack the control flow, we can protect it with the same method for CR0, by monopolizing and checking loop.
MOV IDTR	The monitor must control the entry points of all traps. Interrupt descriptor table (IDT) is mapped as read-only to avoid arbitrary modification. IDTR records the base address of IDT, which must be set to a verified value. Since interrupt is turned off inside the monitor, any modification to IDTR will not take effect before exiting. There is no need to hide this instruction.
WRMSR	The NX (Non-eXecutable) bit controls non-executable memory execution. Similar to CR0 and CR4, a checking loop is enough for this instruction.
VMOFF	This instruction turns off the VMX (VM eXtension) mode of CPU, which is destructive in a virtualization system. This will further allow an attacker to turn on real mode, which will probably hijack the control flow. Since the consequence is immediate, we hide this instruction.
VMRESUME	This instruction immediately returns control flow to guest VM. An attack targeting the guest will work after its execution. An attack targeting the hypervisor by corrupting the VMCS will take effect on next VMExit. Sanity check is necessary before the instruction. We hide this instruction and only allow Xen slices to request its execution. Since the control registers will be loaded from the VMCS on next VMExit, we check and enforce the same invariants as listed above before the resume.

hidden from the attacker. The monitor will do sanity checking and fix the misuse during this period.

Fooling The Monitor: The attacker may try to trick the monitor into giving her extra privileges. If the attacker directly requests an operation for which she does not have the permission, the monitor will immediately discover the violation of invariants and reject the request. Instead, the attacker may pretend to be another iDom. *Nexen* will not give her any chance since the only identification used by *Nexen* is a unique number mapped into the read-only region of each iDom's address space.

Fooling Xen: The Nexen architecture largely reused Xen hypervisor's code. Since Nexen has more restrictions on each component's permissions than the original Xen, such reused code may assume themselves having more permissions than those allowed by Nexen. However, since memory and instruction invariants and policies are enforced by the security monitor, which has the highest privilege in the system, these operations performed by the less privileged Xen slice code will not succeed, nor will they give the attacker any extra permission.

B. Effectiveness in Preventing Exploits

In this subsection we analyze how *Nexen* defends against different types of vulnerabilities.

In total, there are 144 vulnerabilities related to the Xen hypervisor. 127 of them are on the Intel x86 platform. We can directly test the effectiveness over them. Our system can effectively defend against 96 (75.59%) of them. The other 17 vulnerabilities are specific to ARM or AMD processors. Given an equivalent implementation of *Nexen* in these platforms, 11 (64.71%) of them can be prevented. In total, *Nexen* can effectively defend against 107 out of 144 vulnerabilities(74%).

When considering how to prevent attacks, a key observation is that most attacks have a critical step that is non-bypassable. Table I, II, and III categorize vulnerabilities by the position of this key step, attacker's behavior in this step, and the result of the attack. If we can assure (1) this key step happens in the sandbox of Xen slice and (2) any further destructive results will be stopped or limited within the Xen slice, the attack will be successfully prevented. In the design section, we have described how to achieve (1) by moving the most vulnerable parts into Xen slices. They contain most of vulnerabilities that can be exploited as the key step. So the attacker has to be in the context of a Xen slice to do the key step. In this subsection, we will discuss why *Nexen* can achieve (2) by giving concrete data and examples corresponding to each result type.

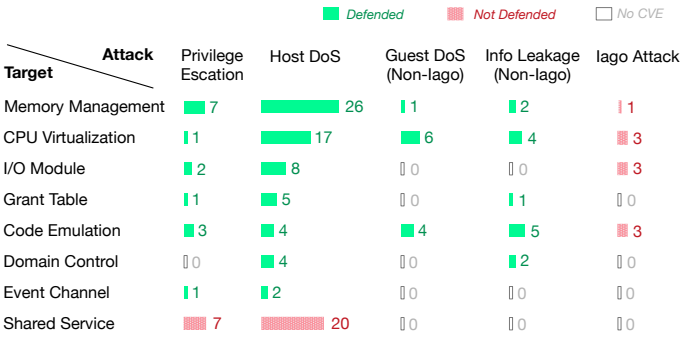

Fig. 5. Effectiveness based on target and result

Figure 5 is a summary of *Nexen*'s effectiveness based on the key step's target and result types. In this figure, we only consider the final results of an attack because *Nexen* mostly stop an attack in its last step. Most attacks can only cause one result. We count them in the normal boxes of the figure with a colored bar representing whether *Nexen* can prevent them (Green for yes, Red for no). Exceptions are those attacks that can potentially achieve privilege escalation, which is an intermediate state that can lead to all kinds of results. They should have been counted once in each of the result types. However, we list their numbers in a separate column to avoid confusion.

There is a clear boundary between those attacks *Nexen* can and can not prevent. Attacks with their key steps happening in Xen slices can mostly be prevented. This is because Xen slice is a sandbox that can be sacrificed. Exceptions are those trying to crash or leak information to a guest VM with Iago attacks. They do not try to harm the hypervisor or other VMs so sandboxing does not work for them. The gate keeper guards interactions between the hypervisor and VMs. Part of attacks that takes effect in a guest VM can be prevented. However, verifying data corrupted by an Iago attack requires recomputing, which the gate keeper is incapable of.

The following includes analysis and experiments about how *Nexen* prevents each type of attacks. Case studies can be found

in Table IX. For those attacks *Nexen* can not prevent, reasons are analyzed in Table X.

Host DoS – False BUG_ON, Page Fault, General Fault The methods used by these attack types to cause a Host DoS are almost the same. They directly trigger an exception (a BUG, e.g., XSA-37/XSA-102/XSA-111/XSA-145/XSA-168, a page fault, e.g., XSA-26/XSA-84/XSA-92/XSA-96/XSA-173, or other kinds of fatal fault, e.g., XSA-12/XSA-44, respectively), the handler of which causes the hypervisor to panic. In an unmodified Xen, this will directly crash the hypervisor and lead to host DoS. In *Nexen*, the handlers for such exceptions are modified: when the attack happens in the context of a Xen slice, the attacker's VM and Xen slice, instead of the whole hypervisor, are killed.

We tested *Nexen*'s effectiveness against this type of attack by calling a customized hypercall that directly triggered a fatal exception. Our system successfully survived the attack with only the attacker's guest VM was killed.

Host DoS – Infinite Loop, Dead Lock, Live Lock The methods used by these attack types to cause a Host DoS are almost the same. They trap a CPU in a task that is non-preemptible for a long time (an extremely long or infinite loop, e.g., XSA-24/XSA-31/XSA-60/XSA-150/XSA-158, a dead lock, e.g., XSA-30/XSA-74/XSA-127, or repetitive long operations, e.g., XSA-11/XSA-45/XSA-89/XSA-118/XSA-146, respectively). One CPU or the entire hypervisor will lose response in this condition. If the watchdog is in use, an NMI will be sent to the CPU after timeout, the handler of which will kill the hypervisor. Either way will cause a DoS in the unmodified Xen. In *Nexen*, the watchdog is in use to detect the trap of the CPU. The NMI will be received normally, but its handler is modified in a similar way as fatal exceptions, that is, only the attacker's Xen slice and VM are killed. If this attack occurs in the context of a Xen slice, no critical data in the hypervisor will be corrupted due to aborting the slice, because a Xen slice does not have the permission to read/write data in other parts of the hypervisor.

We tested *Nexen*'s effectiveness against this type of attack by calling a customized hypercall that directly traps a CPU in a task that is non-preemptible, e.g., an infinite loop in the context of hypervisor. Nexen successfully survived under the attack while only the attacker's guest VM was killed.

Host DoS – Run Out of Resource Attacks of this type try to cause a host DoS by running out a specific type of resources, e.g., memory(XSA-149), disk(XSA-130), or slots of a data structure(XSA-34). Eventually, an unmodified Xen could hang, panic for violating an *ASSERTION*, or crash for a memory corruption. In *Nexen*, each Xen slice and the VM are assigned with their own share of memory and data structure pools. If the attacker try to exhaust any resource, she will only run out her own share, resulting in an error in her Xen slice. As described in previous cases, this error, no matter of which type, will only get the attacker's own VM and Xen slice killed. The system will keep working normally.

We tested *Nexen*'s effectiveness against this type of attack by calling a customized hypercall that keeps allocating memory in the context of the hypervisor. When *Nexen*'s secure allocator was used, nothing happened, because a running Xen slice allocating extra memory is not allowed. When Xen's original allocator was used, the attacker's VM was killed, because Xen slice does not have the permission to touch any data in the allocator and a page fault was triggered.

Info Leak – Memory Out-of-boundary Access Attacks of this type could try to read sensitive data from any part of the system through memory, e.g., XSA-66/XSA-100/XSA-101/XSA-108/XSA-132. They either begin with a memory corruption, e.g., reading out of boundary, or begin with an uninitialized memory mapped or copied to attacker's VM or Xen slice. In the unmodified Xen, a memory corruption could expose the memory of the whole system to the attacker. Also, memory pages can be passed freely inside the hypervisor, leaving a chance for sensitive data to flow to the attacker. In *Nexen*, Xen slices are strictly isolated from each other and the shared service. Sensitive data from other parts of the system are not visible (not mapped) in a Xen slice. In addition to that, pages recycled and passed to a new Xen slice are monitored by the security monitor, who will make sure they are completely cleared during this transition. Thus, all paths from the attacker to victims' sensitive data through memory are blocked.

We tested *Nexen*'s effectiveness against this type of attack by calling a customized hypercall that directly read another VM's state data , to be specific, the domain data structure, in the context of hypervisor. The attacker's guest VM was killed instantly without any return address from the hypercall.

Info Leak – Misuse Hardware Feature Attacks of this type try to get sensitive data directly through registers in the hardware instead of memory, e.g., XSA-52, XSA-62, XSA-172. They mostly start with the hypervisor's failure to completely clear up a register's value. In unmodified Xen, when a register starts to serve another VM, value left in it will be accessible to the new VM, potentially leaking the previous user's information. In *Nexen*, important registers' values are checked and initialized when necessary before entering into the guest. Although some fields are too expensive or semantically too complex to check, information leaked through those containing most valuable information, e.g. stack pointer, PC, EFLAGS, can be avoided.

We tested *Nexen*'s effectiveness against this type of attacks by repeatedly calling a customized hypercall that hang for a while and returns without restoring stack pointer. The hypercall returns normally with the stack pointer restored every time.

Guest DoS (self) Although guest VM is not the primary protection target, *Nexen* does provide some protection against direct attacks aiming at a guest VM. Typically, a bug in the VM's Xen slice, mostly related to CPU virtualization, is exploited by the VM's user program to configure the guest's running state in a malicious way. For example, in XSA-40, an incorrect stack pointer is set for the guest in an operation that can be triggered by a user program. After returning to the guest, the malicious running state will crash the VM's kernel. Other examples of this type include XSA-10, XSA-42, XSA-103, and XSA-106. In *Nexen*, the important running states will be checked by the gate keeper before context switching to guest. Incorrect and malicious configurations are fixed, which will eliminate a considerable number of attacks.

We tested *Nexen*'s effectiveness against this type of attack by calling a customized hypercall that sets the guest VM's

program counter (PC) to 0 before returning. The hypercall returns normally with PC properly restored.

Guest DoS (other) This attack type is very similar to Guest DoS (self). The difference is that the bug in Xen slice is triggered by another VM instead of the victim VM's user program. For example, in XSA-91, Xen fails to context switch the 'CNTKCTL_EL1' register, which allows a malicious guest to change the timer configuration of any other guest VM. Other examples include XSA-33, XSA-36, and XSA-49. In *Nexen*, the approach is also similar, namely, by checking the important running states before context switching to guest and fixing incorrect and malicious configurations.

We tested *Nexen*'s effectiveness against this type of attack by calling a customized hypercall that hang for a while and set guest VM's PC to 0 before returning. The hypercall returns normally with PC properly restored.

Limitation *Nexen* has mainly three aspects of limitations, which will be fixed in our future work. First, *Nexen* cannot handle vulnerabilities in the shared service. In our design, shared service is a unique component and is shared by all Xen slices. If a logic error residing in this part is exploited, the hypervisor may be compromised. Second, *Nexen* does not prevent abuse of I/O devices well. For example, disks are not managed by *Nexen*, which may be exhausted to cause a DoS. This problem can be addressed by extending *Nexen*'s features to cover these I/O device resources. Third, since *Nexen* is unable to capture all *iret* instructions used to return to a PV guest currently, a PV guest's Xen slice compromised by other VMs can bypass the gate keeper's sanity check and arbitrarily modify the guest's running state. Fortunately, this can only result from a malicious administrator.

VI. Performance Evaluation

We evaluated Nexen's performance overhead by running standard benchmarks in a guest VM. We use SPEC CPU2006 and Linux kernel compilation to evaluate CPU and memory overhead and IOzone (a filesystem performance benchmark) and iperf3 (a network performance benchmark) to evaluate the I/O overhead. The configuration of the testing machine is listed in the Table XI. The configuration of benchmarks are listed in Table XII. 'Round" means the times we ran the benchmark. We show the average results along with standard deviations in bar graphs.

The results of CPU and memory related benchmarks are presented in Figure 6. The Y-axis shows the running time of benchmarks. For purely CPU-intensive applications, e.g., perlbench, h264ref, and astar, there is nearly no overhead. This is reasonable because *Nexen* mostly lies in the critical path of memory operations. CPU execution can rarely be intercepted by *Nexen*. Even for the relatively memory-intensive kernel compilation benchmark, the overhead, less than 1%, is negligible. One reason is that the good control flow pattern of *Nexen* avoids excessive interleaving among different Xen slices and the shared service. Xen hypervisor's proper usage of EPT related hardware feature reduces a lot of VMExits for EPT violation, which further reduces the frequency of calling *Nexen*'s sanity checking function There are benchmarks where *Nexen* slightly out-performs the unmodified Xen, e.g., gcc, mcf and libquantum. Considering they show relatively high

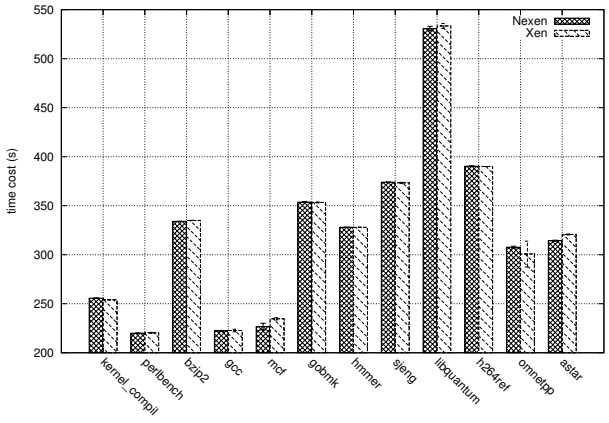

Fig. 6. Performance data of CPU and memory benchmarks.

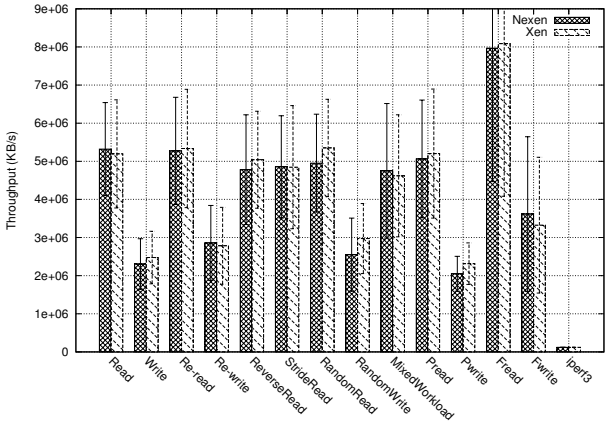

Fig. 7. Performance data of I/O benchmarks.

standard deviations, this could be attributed to measurement variation.

The results for I/O related benchmarks are presented in Figure 7. Iperf3 is a simple tool measuring the network throughput of a system. In our test, the PV drivers, now supported by the native Linux kernel, were used for I/O. Since the data mostly flows through the shared memory between the guest VM and Dom0, the hypervisor is out of the critical path of network I/O. This explains the extremely low overhead in this test (0.02%). IOzone tests various aspects of a filesystem, which indirectly reflect the disk I/O throughput. 4KB block size, 20MB file size and 4 threads were used in this test. The standard deviations for this benchmark set is extremely large. We ran 50 rounds of the test to stabilize the result as much as possible. Benchmarks where *Nexen* out-performs the unmodified Xen are probably result of measurement variation. However, we can not rule out the possibility that *Nexen* changes the pattern of caching and buffering in a way that favors these specific operations. Generally speaking, reading operations are less affected by *Nexen* compared to writing operations. The average overhead in the I/O part is about 2.4%.

Overall, the average overhead of *Nexen* is about 1.2%. Nexen mainly adds to the latency of VMExits and MMU updates. With PV drivers and latest hypervisor version used

TABLE IX. CASE STUDY ON DIFFERENT RESULTS AND TYPES OF ATTACKS.

Result	Attack Type	Nubmer	Case Studies
Host DoS	False BUG_ON	6	XSA-111 (CVE-2014-8866). A piece of hypercall parameter translation code assumes that only the lower 32 bits of a 64-bit register variable are used, violation of which will trigger a BUG_ON that kills the hypervisor. This condition can be deliberately violated by an HVM guest by temporarily changing to 64-bit mode and pass an invalid 64-bit parameter. In Nexen, the vulnerable code runs in the context of a Xen slice because it can only be invoked by a memory management hypercall. The modified BUG_ON logic will only kill current Xen slice VM when it is triggered.
	General Fault	9	XSA-44 (CVE-2014-1917). The logic processing SYSENTER instruction fails to clear NT flag in EFLAGES register, which will lead to a nested GP fault in some situations. This is considered by the original Xen a fatal fault, and will cause the hypervisor to crash. In Nexen, the vulnerable code runs in the context of a Xen slice because it,is part of code emulation subsystem. The modified GP fault handler will kill only the current Xen slice and VM in this situation.
	Page Fault	26	CVE-2014-3967 (one of two CVEs in XSA-96). The implementation of a HVM control operation (HVMOP_inject_msi) fails to do sufficient check for possible conditions of an IRQ. This allows a NULL pointer to be de- referenced, which will lead to a page fault that crashes the hypervisor. In Nexen, this piece of code runs in the context of a Xen slice because it is part of code emulation subsystem. The modified page fault handler will kill only the current Xen slice and VM after the fault.
	Live Lock	9	XSA-5 (CVE-2011-3131). A VM directly controlling a PCI(E) device could issue DMA request to an invalid address. Although this request will be properly rejected, the error handling logic is not preemptable and takes quite some time. Repeating this invalid operation will live lock the CPU. In unmodified Xen, the hypervisor will probably be hung and result in a DoS. In Nexen, this piece of code belongs to the I/O subsystem, which runs in Xen slice context. The CPU under attack will be detected by the watchdog due to losing response for a long time. The NMI sent by the watchdog will interrupt the task and its handler will kill the attacker's Xen slice and VM.
	Dead Lock	4	XSA-74 (CVE-2013-4553). The two locks 'page_alloc_lock' and 'mm_rwlock' are not always taken in the same order. A malicious guest could possibly trigger a deadlock due to this flaw, leading to a host DoS in the unmodified Xen. In Nexen, this piece of code, although deprecated now, should belong to domain control subsystem in Xen slice context. The deadlock will cause one or more CPUs to lose response and trigger watchdog's NMI. Its handler will kill the attacker's Xen slice and VM.
	Infinite Loop	8	XSA-150 (CVE-2015-7970). Under certain circumstance, the hypervisor will search an HVM domain in Populate-on-Demand mode for memory to reclaim. This operation runs without preemption. The guest VM could manipulate its memory in a way that the search becomes a liner scanning, which will hang the hypervisor for a long time. In unmodified Xen, this means a host DoS. In Nexen, this logic, belonging to memory management subsystem, works in the context of Xen slice. Similar to previous examples, the task will be interrupted by watchdog's NMI and the attacker's Xen slice and VM will get killed in the handler.
	Run Out of Resource	4	XSA-149 (CVE-2015-7969). The VCPU pointer array in a domain data structure is not freed on domain teardown. This memory leak, when accumulated over time, could exhaust the host's memory. In an unmodified Xen, this will lead to host DoS. In Nexen, this leak will not accumulate overtime. The VCPU data structure, one of per-domain data structures, is allocated by Nexen's secure allocator and assigned to the domain's Xen slice. Its memory region is recorded in the allocator's memory pool along with the domain's ID. On domain teardown, the memory pool is traversed to search for all memory regions bound to it. Leaked memory will be detected and recycled during this process.
Info Leak	Memory Out-of-boundary Access	11	XSA-108. Xen's code emulation for APIC erroneously emulates read and write permissions for 1024 MSRs where there are actually 256 MSRs. Although writing out of boundary is replaced by no-op which will do nothing, the read operation can go beyond the page set up for APIC emulation and potentially get sensitive data from the hypervisor or other VMs. In Nexen, this piece of code runs in the context of Xen slice because it is part of code emulation subsystem. Since sensitive data of other VMs and the hypervisor are all hidden (unmapped) from the Xen slice, the attacker will either read her own data or read an unmapped page, which leads to a page fault that kills her own VM and Xen slice.
	Misuse Hardware Feature	3	XSA-52. This vulnerability appears on AMD CPU, which is different from Nexen's platform. Given an equivalent implementation on that platform, this attack can be stopped. XSAVE/XSTORE, commonly used to save and restore user running state, is misused so that information other than FOP, FIP and FDP x87 registers are ignored while saving and restoring states with a pending exception. This leaks the running state of previous VM to the attacker. In Nexen, the gate keeper has an internal save for important running states. When returning to guest, registers not restored will be detected and fixed, which wipes the information left by the previous user.
Guest DoS (self)	Various	10	In XSA-40, an incorrect stack pointer is set for the guest in an operation that can be triggeredby a user program. In unmodified Xen, a malicious user could crash the guest VM by triggering this bug.In Nexen, the incorrect value of stack pointer will be detected and fixed by the gate keeper before returning to the guest. The guest VM will keep working normally.
Guest DoS (other)	Various	1	In XSA-91, Xen fails to context switch the 'CNTKCTL_EL1 'register, which allows a malicious guest to changethe timer configuration of any other guest VM. This vulnerability appers on ARM platform.Given a system equivalent to Nexen implemented on ARM, the malicious value of timer register will be detected and fixed by the gate keeper before returning to the victim guest VM. The guest VM will keep working normally.

TABLE X. ANALYSIS OF ATTACKS NEXEN CAN NOT PREVENT

Target	Reason	Number	Analysis
Shared Part	Logic Error	15	This type of vulnerabilities results from the inherent error of codes in the shared part of the system, e.g., domain building(XSA-83). Since the shared part is critical in our system and has relatively higher privilege, exploiting a bug in this part will allow the attacker to do almost anything destructive towards the whole system. Due to the design of *Nexen*, these destructive results can not be prevented.
	Not Supported Feature	7	Xen includes some features that are not essential for virtualization, e.g., PMU(XSA-163). *Nexen* currently does not consider vulnerabilities in these parts. As a result, they are shared by the whole system by default and vulnerabilities in them can lead to the compromising of the whole system. However, this problem can be solved by extending *Nexen* and covering these features.
	Not Supported Resource	2	*Nexen* only limit memory usage of Xen slices and guest VMs. Other hardware resources are left uncontrolled and shared by the whole system, e.g., disk (XSA-130). If an attacker wants to exhaust one of these resources, the host could crash. They can be solved by extending our architecture to cover these non-memory resources and protect and isolate them in a similar way as memory.
	Hardware Bug	3	They are caused by bugs in hardwares. For example, in XSA-9, after executing a certain sequence of safe operations, the CPU could unexpectedly lock itself up. These vulnerabilities can not be avoided unless the manufacturer of the hardware fixes the bug or the system refuses to boot when detecting these problematic hardware.
Guest	Iago Attack	10	The gate keeper monitors every transition between the hypervisor and guest VMs. Typically, if an attacker wants to attack the guest VM kernel or leak some information to the guest, the running state of this VM will be compromized to carry malicious or sensitive data. If the compromized data is simple enough so that a previous state and the operation number are sufficient to check the validty of a new data, *Nexen* can stop this attack. However ,if the attack is well desigened like an Iago attack, which attacks without breaking the isolation, and verifying which requires a recomputing, *Nexen* can not prevent it currently.

TABLE XI. H/W S/W ENVIRONMENT

Host system	Xen 4.5
Host CPU	Intel Core i7-4470 @ 3.4GHz * 8
Host memory	16GB
Guest system	Ubuntu 16.04-1 (HVM)
Guest VCPU number	4
Guest memory	4GB

TABLE XII. BENCHMARK CONFIGURATION

benchmark	round	config
IOzone	50	4KB block size, 20MB file size, 4 threads
SPEC CPU2006	9	real world workload
Kernel Compiling	20	linux 4.7, default config
iperf3	20	TCP package

where the frequency of both events dramatically drops, the overhead of *Nexen* can be further reduced.

VII. RELATED WORK

Hypervisor Re-organization for Security. Besides the systems mentioned in II-B, Nova [25] reorganized the hypervisor to several per-VM hypervisors running in user mode and one small privileged micro-hypervisor running in kernel mode. The attacks from one VM can be limited in one per-VM hypervisor. Min-V [23] uses reduce the TCB of the hypervisor by removing all the unused code base dynamically, which is called *delusional boot*. Min-V first boots a guest VM on a full-fledged hypervisor, then takes a snapshot of the VM and migrates it to the production platform with a different hypervisor that disables all the virtual devices that are not critical to running VMs, and restores the VM on the new platform. SSC [10] proposes a solution to enable multiple Dom0s, which is called "UDom0" that runs as user-level service domains, and enforce the isolation between the UDom0s. These works aimed to protect the hypervisor from guest VMs by reducing the trusted computing base (TCB). However, they only provide limited protection against attacks from a malicious hypervisor.

Hypervisor Fault Tolerance. There are also many researches that target hypervisor's fault isolation and tolerance. ReHype [20] tolerates hardware faults and hypervisor bugs by microrebooting. It can preserve the state of all running VMs so the recovery is transparent to the guest VMs. FTXen [18] focuses on tolerating in-field hardware errors of virtualization software stack on relaxed hardware. It isolates the faults of a relaxed core within the boundary of the guest VM running on that core without affecting other VMs or the hypervisor. Another way to isolate the fault is nested virtualization, e.g., the Turtles project [9] and CloudVisor [37]. Intel keeps improving the hardware support for nested virtualization for better performance, and recently Xen also adds support for nested virtualization in its mainstream [16]. TinyChecker [30] achieves similar goal with nested virtualization by adding a small software layer for hypervisor failure detection and recovery. These systems consider hardware faults and software bugs instead of security vulnerabilities, thus they do not take attacks like privilege escalation or bypassing the mechanism of fault tolerance into consideration.

Hardware-assisted Hypervisor Security. NoHype [19], [29] replaces the software hypervisor by hardware virtualization extensions of processor and I/O devices. However it loses the flexibility of resource management brought by virtualization. HyperSentry [5] leverages System Management Mode (SMM) to protect the hypervisor's control flow. H-SVM [17] and HyperWall [27] decouple memory management and security protection. The hypervisor can manage all the memory resource but cannot access the memory arbitrarily, e.g., once some memory pages are assigned to a guest VM, the hardware ensures that it cannot be accessed by the hypervisor without explicitly sharing. Such design can effectively prevent attacks from the hypervisor to guest VMs, but requires non-trivial hardware modifications.

There are also many work on designing new hardware to protect guest VMs from untrusted hypervisor [17], [28], [36]. Some of the design has already been deployed in commodity hardware, e.g., Intel SGX [3], [15], [21]. Haven [8] successfully runs unmodified application inside enclave protected by hardware from system software including operating system and hypervisor. However, these systems usually consider the attack *from* the malicious hypervisor, but does not consider some types of attack *against* the hypervisor, e.g., the DoS attack that crashes the entire host machine.

MMU Virtualization. HyperSafe [32] proposes a technique named *non-bypassable memory lockdown* that gathers all the MMU operations to a specific module and deprivileges other modules to do similar operations. HyperSafe focuses on protection of hypervisor's control flow integrity (CFI), while our system considers on Xen decomposition and deprivileging besides CFI. Nested Kernel [14] further provides MMU virtualization as a primitive of operating system to enhance the security of all kinds of kernel modules. On ARM platform, TZ-RKP [4] puts the MMU controller into a "secure world" protected by ARM TrustZone [2]. SKEE [6] also depriving the OS kernel from controlling MMU, but not using TrustZone for better performance.

VIII. CONCLUSION

In this paper, we have conducted a systematic research on all the 191 (effective) vulnerabilities published in Xen Security Advisories (XSA), of which 144 (75.39%) are directly related to the hypervisor itself. We then analyzed the distribution of bugs among different components and consequences. Based on the above analysis, we proposed a new architecture for Xen hypervisor, named Nexen, that provides a way to deconstruct Xen so that a malicious hypervisor cannot directly access the data within a guest VM, and a malicious guest VM cannot affect other VM or the host system. Nexen decomposes the Xen hypervisor into different internal domains: multiple per-VM slices and one shared service. Each internal domain has least privilege and are isolated, so that even if one gets compromised, it will not affect other ones. We have implemented a prototype of our design which can correctly handle 107 out of 144 vulnerabilities (74%). The performance evaluation results also indicate that the overhead is negligible.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments. This work is supported in part by National

Key Research and Development Program of China (No. 2016YFB1000104), China National Natural Science Foundation (No. 61303011 and 61572314), a research grant from Huawei Technologies, Inc., National Top-notch Youth Talents Program of China, Zhangjiang Hi-Tech program (No. 201501-YP-B108-012), a foundation for the Author of National Excellent Doctoral Dissertation of PR China (TS0220103006), Singapore NRF (CREATE E2S2), NSF via grant number CNS 1513687, and ONR via grant PHD.

REFERENCES

[1] http://ipads.se.sjtu.edu.cn/xsa/.

[2] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4):18-24, 2004.

[3] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *HASP*, volume 13, 2013.

[4] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90-102. ACM, 2014.

[5] 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*, pages 38-49. ACM, 2010.