Demystifying the Access Control Mechanism of ESXi VMKernel

Yue Liu*◇, Zexiang Zhang†◇, Jiaxun Zhu‡, Hao Zheng, Jiaqing Huang, Wenbo Shen‡
Gaoning Pan§¶, Yuliang Lu†, Min Zhang†, Zulie Pan†, Guang Cheng*✉

*Southeast University, China †National University of Defense Technology, China
‡Zhejiang University, China §Hangzhou Dianzi University, China
¶Zhejiang Provincial Key Laboratory of Sensitive Data Security and Confidentiality Governance, China

***Abstract*—VMware ESXi is a widely deployed enterprise-grade Type-1 hypervisor that serves as the foundation for modern cloud infrastructure. To reinforce privilege isolation, ESXi introduced a mandatory access control mechanism in VMKernel. However, due to VMKernel's proprietary and closed-source nature, its internal access control architecture remains largely opaque and under-explored. Prior research has focused primarily on virtual device vulnerabilities and virtual machine escape, leaving the internal access control mechanisms and privilege model of VMKernel largely unexamined.**

To address this gap, we conduct the first comprehensive security analysis of VMKernel's access control mechanism. We develop a domain-control structure oriented analysis method to reconstruct key internal permission logic, and design a structure-aware debugging framework to support fine-grained runtime validation. Using this framework, we uncover several critical design flaws, including writable and unprotected in-memory control structures and exploitable developer-reserved syscall interfaces. We demonstrate three practical attack scenarios that abuse these flaws to bypass sandbox restrictions, escalate privileges, and gain persistent access. In total, we discovered and reported 14 vulnerabilities to VMware, all of which have been confirmed and fixed, with a total of $42,000 in bug bounties awarded.

I. Introduction

VMware ESXi is a leading enterprise-grade Type-1 virtualization platform, widely deployed in private clouds, enterprise data centers, and other mission-critical environments. It is built on a bare-metal architecture that delivers high performance, strong isolation, and fine-grained resource scheduling, enabling large-scale virtual machine(VM) deployments and high-availability cluster management. Within the bare-metal hypervisor segment, ESXi holds over 45% of the global market share [1], establishing itself as a cornerstone of modern cloud infrastructure and playing a critical role in business continuity, security, and elastic resource management.

◇Both authors contributed equally.
✉ Corresponding author: Guang Cheng (chengguang@seu.edu.cn)

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240700
www.ndss-symposium.org

However, the widespread adoption of ESXi also makes it a high-value target for attackers. In real-world incidents, the APT group "Dark Angels" reportedly compromised ESXi systems and conducted large-scale ransomware campaigns, extorting over $135 million [2]. Additionally, at the Pwn2Own hacking competition, a successful ESXi exploit was valued at $150,000, topping the event's bounty leader board [3]. These cases demonstrate the critical importance and urgency of reinforcing ESXi's security mechanisms.

To enhance its security, ESXi has incorporated a dedicated access-control mechanism since version 6.5, serving as a foundational layer for regulating system behavior and enforcing privilege separation across kernel components. This mechanism is implemented within VMKernel, the core kernel of ESXi responsible for system-wide resource management. It adopts a sandbox-like isolation model, partitioning system processes into distinct security domains and enforcing whitelist-based restrictions on critical operations such as syscalls, file access, and network communication. This architecture establishes a strong internal security boundary within ESXi, while structurally limiting the attack surface, particularly for threats involving virtual machine escape.

ESXi's access-control mechanism is pivotal in defending against VM escape threats, yet its design and security guarantees remain largely unexamined. One reason is that VMKernel—the proprietary, closed-source core of ESXi—lacks public documentation and symbol information, and analyzing its permission-management strategy is exceptionally challenging. Prior research has concentrated on vulnerabilities in the VMX module or virtual devices to achieve VM escape [4]–[6], while little attention has been paid to how VMKernel enforces system-wide privilege constraints that ensure access isolation and resource security among internal kernel components. This knowledge gap limits the understanding of ESXi's overall security boundary and impedes further enhancement of its protective mechanisms.

To bridge this gap, we conduct a in-depth analysis of VMKernel's access-control mechanism, aiming to reveal its internal privilege-management architecture, validation logic, and vulnerabilities. Our study pursues two core objectives: (1) to understand, through static analysis, the data structures and algorithms VMKernel employs to enforce permission

checks on critical resources; and (2) to examine these enforcement paths at runtime, evaluating whether adversaries can circumvent the intended policies—such as via structural tampering—and thereby expose hidden security threats.

Achieving these objectives, however, is non-trivial due to two key challenges. First, VMKernel's access-control mechanism spans a wide range of heterogeneous resources, including system calls, file access, and network communication. Our preliminary analysis reveals that permission checks are scattered across diverse execution paths without a centralized entry point, resulting in fragmented analysis and a lack of structural guidance. Second, debugging support is severely limited. Although VMware DebugStub [7] provides basic system-level tracing, it lacks semantic and structural awareness of VMKernel's internal components. Given the complexity and high frequency of access-control logic execution, existing tools fail to deliver the necessary breakpoint precision or path control, thereby hindering effective dynamic analysis and exploit validation.

To address these challenges, we propose two techniques. First, we introduce a *domain-control structure oriented VMKernel access-control mechanism analysis* method that comprehensively reconstructs VMKernel's access-control data structures and validation paths. Second, we extend VMware's DebugStub interface to build a *structure-aware VMKernel access-control mechanism debugging* framework, enabling precise runtime observation of permission checks and validation of potential attack paths.

Our study uncovers multiple security weaknesses in VMKernel's access-control mechanism that undermine its ability to enforce isolation. We find that its access-control mechanism relies on a centralized in-memory structure that lacks write protection and integrity checks, rendering it vulnerable to tampering. We further identify several developer-reserved debug syscalls that expose high-privilege debug interfaces and can be abused after sandbox escape. Additionally, more than a dozen syscall handlers suffer from memory safety flaws such as stack overflows and information leaks, creating a broad and exploitable attack surface.

Based on these findings, we conducted controlled proof-of-concept attacks in a secure environment, demonstrating successful privilege escalation and other practical exploit effects on ESXi. We responsibly disclosed all 14 identified vulnerabilities to VMware, all of which have been confirmed and patched, with a total of $42,000 awarded through its official bug bounty program.

In sum, this paper makes the following contributions.

- We develop the first analysis framework specifically targeting the VMKernel access-control mechanism.
- We propose two novel analysis techniques for analyzing and deconstructing VMKernel's access-control logic.
- We identify three critical security flaws in VMKernel's access-control design.
- We design three practical attacks that exploit the flaws to compromise system integrity.

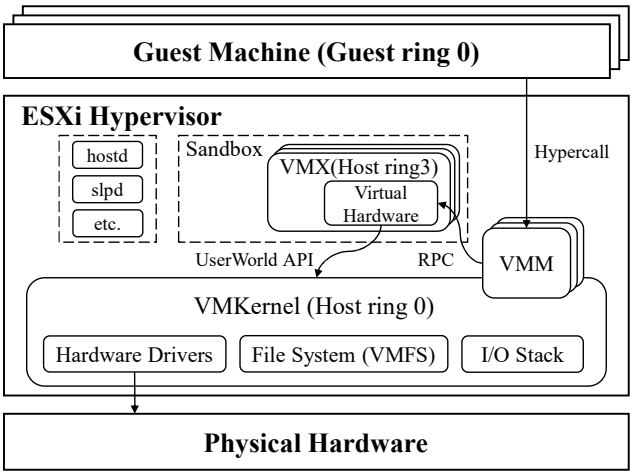


Fig. 1: Architecture of VMware ESXi

- We responsibly disclosed 14 security vulnerabilities to VMware, all of which have been confirmed and fixed.

II. Background

In this section, we first introduce the architecture of ESXi, outlining the roles and interactions of its core components. We then analyze the major security threats it currently faces. Finally, we briefly present the access-control mechanisms implemented by ESXi to mitigate such threats, laying the groundwork for subsequent security analysis.

A. Architecture of ESXi

VMware ESXi is a proprietary, closed-source Type-1 hypervisor known for its strong security guarantees and widespread use in private cloud deployments [8]. As shown in Fig. 1, ESXi adopts a modular architecture comprising four core components: the Guest Machine, the Virtual Machine Monitor (VMM), the Virtual Machine Extension (VMX), and the VMKernel. The VMM provides the execution environment for guest OSes and manages their I/O, while the VMX, running in the host's ring-3 layer, integrates virtual hardware functions. Each Guest Machine is paired with a dedicated VMM and VMX process. At the core, the VMKernel contains physical device drivers, a lightweight file system (VMFS), and a syscall interface, enforcing unified access control and resource scheduling. Additionally, ESXi includes auxiliary service components such as `hostd`, which processes vCenter management requests, and `slpd`, which provides Service Location Protocol (SLP) services. User World [9] is a minimal user-space environment within VMKernel for running essential system processes like VMX, it exposes limited APIs and file access, providing a controlled runtime that extends VMKernel functionality while maintaining process-level isolation.

In ESXi, guest access to host hardware is accomplished through the coordinated operation of multiple components. Specifically, a guest initiates a request via memory-mapped I/O (MMIO), port-mapped I/O (PIO), or a hypercall. The Virtual Machine Monitor (VMM) captures and interprets the request,

then invokes the corresponding virtual device in the VMX process using a Remote Procedure Call (RPC). The virtual device, in turn, forwards the request to the VMKernel via the User World API interface. After performing access-control checks and resource scheduling, the VMKernel carries out the low-level hardware interaction. The result is then propagated back to the guest system along the original path.

B. Security Threats to ESXi

The primary security threats faced by ESXi mainly stem from attacks targeting its internal components. Attackers typically craft malicious inputs to trigger memory corruption vulnerabilities within these components, allowing arbitrary command execution in the context of the compromised process on ESXi. The impact of such attacks varies depending on the targeted component, with the most representative categories being Remote Code Execution (RCE) and virtual machine escape. RCE attacks usually target ESXi components that directly interact with the Internet. For example, CVE-2021-21974 [10] exploits a vulnerability in the SLP protocol, enabling attackers to launch remote attacks and execute commands on ESXi with the privileges of the slpd process. In contrast, virtual machine escape attacks primarily target components related to virtual machine management, such as VMX or VMM. A representative case is CVE-2024-22252 [11], which exploits a vulnerability in the XHCI virtual device implementation within the VMX process, allowing an attacker to initiate the exploit from within a virtual machine and ultimately execute commands on the host with VMX process privileges. The virtual machine escape attack poses a severe threat to virtualization platforms, as it breaks the boundary between the guest and the host, fundamentally undermining the isolation and trust model upon which hypervisors are built.

C. Access-Control Mechanism of ESXi

To establish strong privilege separation and runtime isolation across critical components, as well as to mitigate the practical threat of high-risk attacks such as virtual machine escape, VMware ESXi introduced a mandatory access-control mechanism starting from version 6.5, further enhancing the platform's security posture. [4] This mechanism is enforced by the VMKernel and establishes sandbox-like isolation by dividing the system into security domains and applying whitelist-based access controls.

Specifically, VMKernel first classifies system components into different security domains based on their usage scenarios. For example, the `globalVMDom` domain, which contains the VMX, and the `appDom` domain, which hosts multiple service process components. Then, it defines fine-grained access-control policies for each domain, specifying the allowed socket types, accessible file paths along with their permissions, and the list of permitted syscalls. For instance, Listing 1 shows the policy configuration for the `globalVMDom` domain, which explicitly restricts VMX processes in terms of network usage, file access, and syscall capabilities.

```
-c dgram_vsocket_bind grant
-c dgram_vsocket_create grant
-c dgram_vsocket_send grant
...
-d tpm2emuObj tpm2emuDom file_exec grant
...
-s genericSys grant
-s vmxSys grant
-s ioctlSys grant
...
-p inet_socket_bind all grant
-p inet_socket_connect loopback grant
-p inet_socket_connect nonloopback grant
...
-r /usr/share/certs r
-r /bin/remoteDeviceConnect rx
-r /bin/vmx rx
...
```

Listing 1: Access-control rules for the `globalVMDom` domain: `-c` for socket operations, `-d` for object execution, `-s` for syscall groups, `-p` for connections, and `-r` for file access.

This security model ensures that even if an attacker successfully exploits a vulnerability to execute arbitrary instructions on ESXi, the resulting privileges remain strictly confined within the security domain of the compromised component. Access to other domains or critical system resources is effectively blocked, thereby limiting the scope of potential damage. For example, the `globalVMDom` domain explicitly disallows the execution of high-privilege sensitive syscalls such as `execve` and restricts access to protected files like `/etc/shadow`. Consequently, even if successfully escaped from the guest machine and breached the isolation between the guest and host , the attacker is prevented from establishing a remote shell, escalating privileges, or launching lateral attacks against other virtual machines. This enforcement mechanism significantly reduces the overall attack surface and enhances the depth of protection across the virtualization platform.

D. Motivation

Existing research has primarily focused on concrete exploitation cases such as RCE and virtual machine escape [4]–[6], while targeted studies of the access-control mechanisms implemented in VMKernel remain limited. In fact, the access-control mechanism plays a crucial role in constraining the internal attack surface of ESXi. Unlike the *Seccomp* [12] mechanism in Linux, which is an BPF-based syscall whitelisting mechanism applied per process, ESXi's access-control mechanism is a standalone, integrated subsystem within the VMKernel that governs multiple privileged resources under a unified design. This broader and more centralized model exposes a fundamentally different security surface and therefore merits dedicated analysis. This integrated design means that once a weakness in this system is exploited, it is often more covert and easier to execute than conventional full-compromise approaches, such as installing maliciously signed drivers in controlled environments. In other words, flaws at

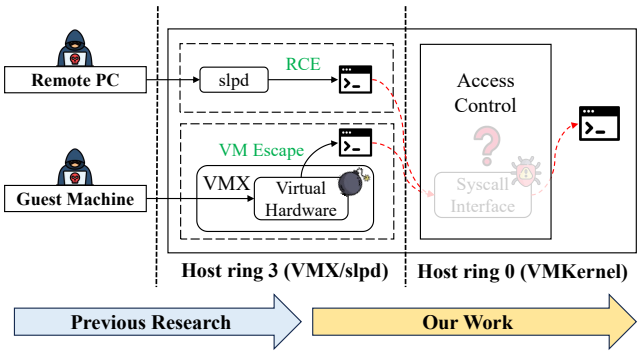


Fig. 2: Our work aims to identify weaknesses in VMKernel's access-control mechanism and break through its sandbox.

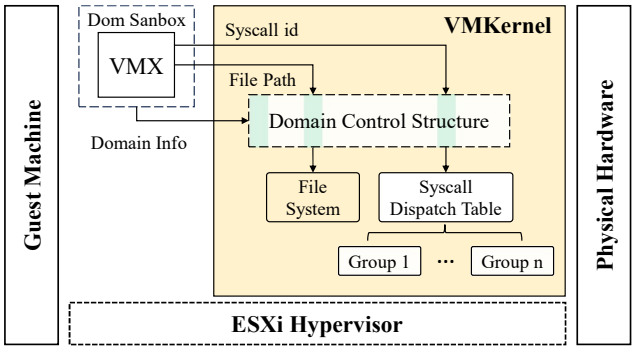


Fig. 3: The Analysis Objective of our work. The components highlighted in white within VMKernel represent the primary focus of this study.

the access-control level not only enable privilege escalation but also reveal a underlying weakness in ESXi's defense-in-depth design, underscoring the need for deeper mechanism-level analysis and reinforcement.

Against this backdrop, as illustrated in Fig. 2, this study aims to bridge this gap through a comprehensive reverse engineering study of VMKernel's access-control system. Our goal is to identify weaknesses that could potentially break through the access-control sandbox, uncover previously undisclosed security issues, and provide insights for VMware to strengthen its protection mechanisms. By analyzing this critical yet understudied component, we seek to deepen the understanding of ESXi's internal privilege enforcement and enhance the overall security of private cloud deployments.

III. Framework Design and Implementation

To enable in-depth analysis of VMKernel's access-control mechanism, we develop a tailored framework designed around its proprietary architecture and enforcement model. The framework aims to reconstruct internal permission logic, monitor runtime validation behavior, and expose potential attack surfaces. This section begins by outlining the analysis objectives and technique challenges, then introduces the framework's overall design and key implementation components.

A. Analysis Objectives

VMKernel employs a sandbox-like access-control mechanism that partitions processes into distinct security groups, each governed by dedicated security policies. This design enables fine-grained permission control over operations such as syscalls, file access, and network communication. Rather than being scattered or ad hoc, its access-control logic is centrally organized through unified data structures and a consistent policy framework, exhibiting a clearly structured implementation style. As illustrated in Fig. 3, this paper aims to thoroughly dissect the implementation of VMKernel's access-control mechanism, with a particular focus on its control structure and the permission allocation strategies for regulated resources such as syscalls, in order to uncover potential security vulnerabilities and exploitable attack surfaces. To this end, we design a analysis framework for the VMKernel, which is intended to fulfill the following two core goals:

Goal 1: Reveal the internal access-control logic of VMKernel. Our first goal is to achieve a comprehensive understanding of the architectural design and enforcement principles underpinning VMKernel's access-control mechanisms. Due to its proprietary and undocumented nature, little is publicly known about how VMKernel defines privilege boundaries or performs permission validation. This lack of transparency hinders rigorous security analysis and defense modeling. By clarifying its internal control structures and identifying how access policies are implemented and enforced, we seek to expose potential design flaws or hidden vulnerabilities that may compromise system integrity.

Goal 2: Enable runtime validation of access-control behavior. The second goal focuses on observing and validating the dynamic behavior of VMKernel's access-control mechanisms during execution. While static analysis provides insights into structural logic, it is insufficient to capture the real-time enforcement of permissions. This goal aims to enable accurate, fine-grained monitoring of access checks at runtime, allowing us to verify security properties, detect inconsistencies or bypasses in the enforcement logic, and evaluate the practical feasibility of potential attacks under real-world conditions.

B. Challenges

To achieve the above goals, our analysis must address two key technical challenges:

Challenge 1: Fragment and Opaque access-control logic. As a commercial closed-source Type-1 hypervisor, VMKernel exhibits a highly complex and fragmented access-control mechanism. As shown in Listing 1, it manages security-domain privileges across multiple dimensions, including syscall execution, file access, and network communication. Because the types of controlled resources and their management approaches differ, the corresponding validation logic is dispersed across various subsystems, lacking a unified verification framework or centralized checkpoint. Our preliminary analysis of the VMKernel binary reveals that permission checks for file-access operations alone are distributed

across as many as 28 different functions. Such a decentralized implementation significantly increases the difficulty of identifying privilege boundaries and modeling access-control behavior, any attempt to manually locate permission-checking instructions across these scattered components would demand extensive human effort and be highly inefficient. Furthermore, as a kernel-level program, VMKernel lacks conventional entry symbols (e.g., `main`, `start_kernel`), which further complicates control-flow analysis.

To date, no prior work has conducted an in-depth study of VMKernel's access-control mechanism. In the absence of symbols, documentation, or prior research, identifying and modeling the internal privilege-management mechanisms of a closed-source Type-1 hypervisor represents a task of considerable technical difficulty and scientific significance.

Challenge 2: Limited dynamic debugging support for fine-grained tracing. Traditional kernel debugging methods, such as QEMU-based virtualization debugging, can be applied to certain Type-1 hypervisors (e.g., KVM, XEN). However, due to the lack of complete hardware emulation and full support for privileged instructions, they cannot reliably run or debug ESXi [13], the closed-source Type-1 hypervisor. Because VMware Workstation shares a substantial amount of code and system mechanisms with ESXi, using its native DebugStub interface represents the ideal practical option for debugging [14]. Nevertheless, DebugStub operates by emulating breakpoints across the entire virtual machine address space, which imposes significant limitations on both precision and performance—it supports only a small number of breakpoints and lacks fine-grained conditional monitoring capabilities. These constraints make it unsuitable for analyzing the highly distributed permission-validation logic within VMKernel's access-control mechanism. Moreover, VMKernel continuously issues large numbers of system calls during execution; relying solely on conventional breakpoints without structural or semantic awareness results in excessive breakpoint hits, disrupts normal execution, and introduces analytical bias.

Consequently, existing debugging tools cannot provide sufficient observability or semantic depth to trace permission-check operations and structural updates at runtime, severely limiting the ability to validate and evaluate potential attack behaviors. This limitation has become a central technical bottleneck in deeply analyzing the access-control mechanism of VMKernel.

C. Analysis Framework Overview

To achieve our analysis objectives, we design and implement an analysis framework, as illustrated in Fig. 4. Specifically, we construct a three-tier nested virtualization environment: We adopt VMware Workstation as the underlying hypervisor and deploy an ESXi instance as the primary target for analysis. In parallel, a Linux virtual machine is configured as the monitoring host, which simultaneously performs static analysis of the VMKernel and establishes dynamic debugging via VMware Workstation's DebugStub [7] interface. To simulate a realistic cloud computing environment, a nested Linux guest is instantiated within the ESXi hypervisor to trigger the activation of core VMKernel components under typical workload conditions.

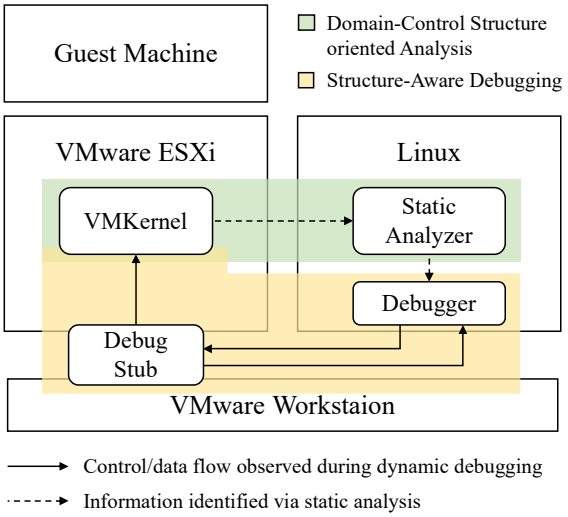


Fig. 4: Overview of Analysis Framework. Based on this framework, we design two techniques in section III-D and III-E

In addition, based on this architecture, we developed two key techniques to address our research challenges: a *domain-control structure oriented VMKernel access-control mechanism analysis* approach and a *structure-aware VMKernel access-control mechanism debugging* method. The following two sections provide a detailed exposition of each technique.

D. Domain-Control Structure oriented VMKernel Access-Control Mechanism Analysis

As discussed in III-B, analyzing VMKernel's access-control mechanism is particularly difficult due to the highly fragmented nature of permission checking logic and the lack of clear analysis entry points. However, as described in II-C, VMKernel applies a unified permission and access-control policy to all processes within the same security domain. This one-to-many design, where multiple processes share a single domain, implies that permission enforcement relies on a centralized data structure and a consistent policy mechanism.

This design provides a natural entry point for our analysis. In this work, we take the reconstruction of the security domain control structure as the foundation, and progressively extract the underlying access-control logic and implementation details throughout the process. Our approach consists of three main steps. First, we locate and reconstruct the internal VMKernel structure responsible for managing security domain permissions, identifying its overall layout and the specific roles of each field. Second, to more effectively trigger permission-checking behavior, we analyze VMKernel's syscall mechanism, including the call entry points, dispatch table structures, and permission-related metadata. Finally, we take the syscall handling process as an entry point and, with the aid of dynamic

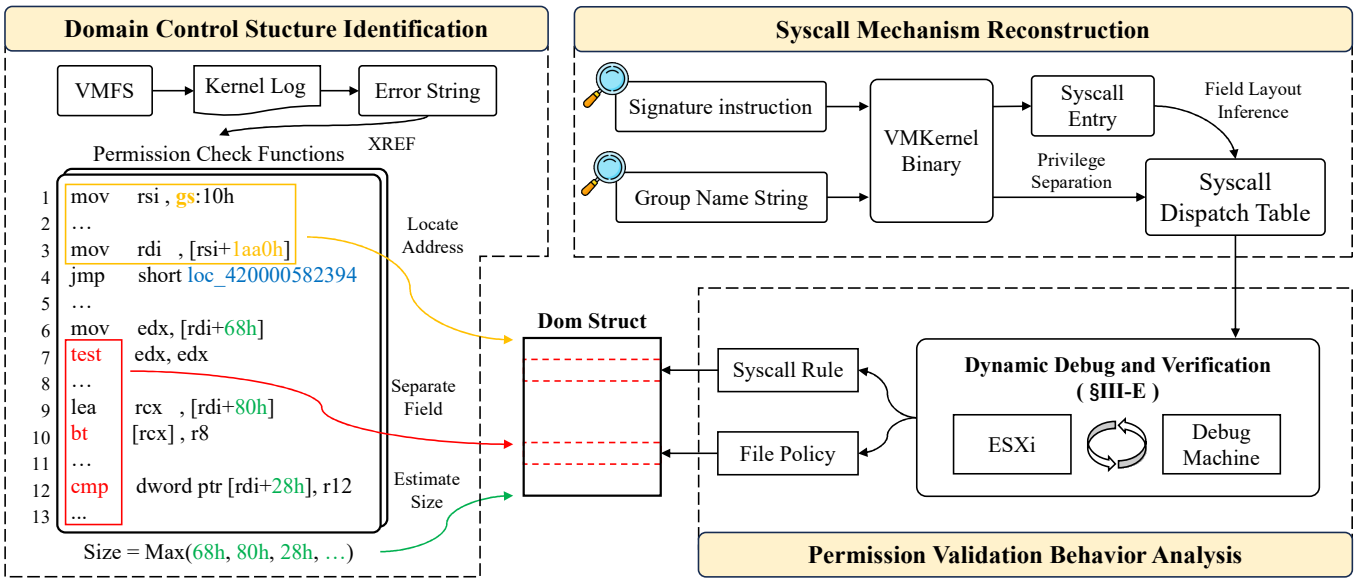


Fig. 5: Domain-Control Structure oriented VMKernel Access Mechanism Analysis

debugging, conduct an in-depth analysis of how the security domain control structure participates in permission checks within the access-control mechanism.

1) Domain Control Structures Identification

We designed a method for automatically identifying the control structure of security domains, which includes four main steps: identifying permission-checking related functions, locating the control structure, determining permission-related fields, and validating the structure layout and boundary. First, we use the dynamic debugging framework to construct diverse security domain contexts and issue random syscall requests, triggering permission check behaviors in VMKernel. We then extract error strings from kernel logs and search the VMKernel binary to locate the corresponding permission-checking functions. Second, in the structure location and field determination stage, based on the characteristic that "multiple processes share a single security domain," we infer that VMKernel must access the security domain control structure from the process context during permission checks. We analyze the instructions accessing contextual information to locate the control structure. Third, by analyzing how this structure is referenced during permission checks, we infer its field layout and overall size. Finally, we perform dynamic testing to validate and refine the inferred distribution of the control structure, eliminating false positives and ensuring the correctness of the reconstructed layout.

As shown in Fig. 5, we first extract error strings from kernel logs to locate the functions of permission validation. Within the function, we identify instructions that access the `gs` or `fs` segment registers (line 1) and trace their reference chains to determine the memory address of the security domain control structure (line 3). We then search for logic-related opcodes such as `test`, `bt`, and `cmp` (line 7,10,12), and identify the key instructions used for permission checks. Since this analysis is performed only within the identified permission-validation functions or corresponding basic blocks, the targeted strategy confines the search to semantically relevant regions, thereby effectively reducing false positives that would otherwise arise from global binary scanning. By performing reaching value analysis [15] on the operands of key instructions, we extract the field offsets within the structure that are involved in permission checks. Based on the register width used in offset access, we further infer the positions and sizes of the relevant fields, and estimate the overall structure size using the maximum offset. Finally, to address potential false negatives, we leverage the fact that the control structure is allocated as a heap object within the kernel. By examining the surrounding heap layout through our dynamic debugging framework, we can accurately determine the structure's full memory boundary and ensure that all managed fields are identified. We further perform manual validation to confirm completeness and eliminate residual false negatives.

2) Syscall Mechanism Reconstruction

Among permission-restricted behaviors, syscalls are the most accessible and thus serve as an ideal entry point for analyzing the VMKernel access-control mechanism. However, ESXi's closed syscall interface and undocumented grouping (Listing 1) complicate permission modeling, necessitating a reconstruction of its syscall architecture. To this end, we propose an automated method for reverse-engineering the VMKernel syscall mechanism (as illustrated in Fig. 5), which consists of three key steps: syscall entry identification, dispatch table structure inference, and group pattern recovery.

First, we observe that syscall dispatching in VMKernel typically exhibits characteristic instruction patterns, such as `call [base_addr + syscall_id * offset]` or `mov edi, [base_addr + syscall_id * offset]`. By statically scanning for such instructions, we locate the

address of the syscall dispatch table. Based on the extracted `offset` value, we derive the mapping between syscall IDs and their corresponding handler functions, thereby identifying multiple syscall handler entry points.

Second, based on our observations and prior experience, we find that each syscall entry in the dispatch table is managed by a uniform minimal structure. Accordingly, the address differences between handler functions should be integer multiples of the structure's size. Under this assumption, we calculate the address gaps between the identified handlers and use their greatest common divisor (GCD) to infer the size of the base structure unit. We then enumerate the remaining entries in the dispatch table using this structure size as the step interval to identify additional, previously unrecognized handlers. Building on this, we further analyze the memory layout between handler entries at word-level granularity to reconstruct the structure's internal fields, ultimately recovering the complete syscall dispatch table along with its underlying structure definition. Since the analysis covers the entire address space of the dispatch table, the completeness of the identified syscall entries is guaranteed.

Finally, we identify syscall group semantics by searching for group name strings in Listing 1 (e.g., `genericSys`, `vmxSys`) in the binary and correlating them with group-related fields in the dispatch table, thereby recovering the syscall grouping logic.

3) Permission Validation Behavior Analysis

Based on the previously reconstructed syscall dispatch table, we generate diverse syscall sequences under various privilege contexts within a dynamic debugging framework. Meanwhile, we trace the execution paths of permission-checking functions to observe which fields in the permission structure are accessed and verified under different combinations of contexts and syscalls. Through comparative analysis, we identify the key fields that are associated with syscall behavior and have a tangible impact on access decisions, then we label them as the syscall control fields. We further analyze the meaning of each byte within these control fields to determine how individual bits correspond to specific syscall groups and how their values regulate permission enforcement. We employ a method that combines dynamic execution tracing with validation logic analysis to reconstruct the effect of each byte within the field, enabling us to precisely interpret its semantic role in the access-control process and conduct tampering tests.

In addition, considering that most file operations are indirectly triggered by syscalls, we apply the same approach to analyze the control fields and validation logic involved in file read/write operations. As the procedure is largely similar, the corresponding details are omitted in this paper.

E. Structure-Aware VMKernel Access-Control Mechanism Debugging

As described in III-C, we build a nested virtualization-based dynamic debugging framework using VMware Workstation's DebugStub. However, as discussed in the challenges, analyzing VMKernel's access-control mechanism via DebugStub still faces several obstacles, such as imprecise breakpoint placement and difficulty in interpreting control logic. To address these limitations, we incorporate semantic awareness and structural understanding into the debugging process to improve both precision and observability. As illustrated in Fig. 6, our technical approach is designed to address the challenges mentioned above and consists of two core components: *Symbol Recovery and Structure Injection* and *Privilege Context Emulation and Execution*. In the following, we provide a detailed explanation of these two components.

1) Symbol Recovery and Structure Injection

We observe that although VMKernel is a closed-source component, the ESXi runtime loads several auxiliary modules such as device drivers. The executable files corresponding to these modules can be extracted and unpacked to recover partial symbol information, and parsing ESXi's runtime configuration allows us to determine the modules' memory load addresses. During debugging, we load the recovered symbol tables at the modules' base addresses and inject our reverse-engineered structural definitions into the debugger. These imports improve the readability of call stacks and memory contents, enabling precise breakpoint control at permission-checking instructions to observe changes in permission-related structures and fields. Additionally, we configure the debugger to capture spontaneously issued syscalls during VMKernel execution, collecting syscall parameter ranges that support our violating analysis.

2) Privilege Context Emulation and Execution

To better trigger various permission validation processes and test potential security issues, we designed a privilege context emulation and execution approach. Specifically, we inject an execution agent into the ESXi environment. This agent can simulate different security domain identities by adjusting its parameters and modifying configuration data, and then initiate different syscalls accordingly. In doing so, it emulates diverse privilege contexts and helps activate the full permission validation workflow.

Combining the above two components, our framework achieves both fine-grained observability and controllable execution. The integration of structural awareness with privilege-context emulation enables us to comprehensively analyze VMKernel's runtime behavior and effectively validate our attack hypotheses during dynamic testing.

F. Framework Implementation

As shown in the Fig. 4, we have built and implemented a unified framework for VMKernel analysis and debugging based on the methods described earlier.

The analysis framework comprises two core modules: a static analysis module for *domain-control–structure–oriented access-control analysis* and a dynamic debugging module for *structure-aware access-control debugging*. The static module, implemented on the debugging workstation, includes a custom VMFS parser to extract the VMKernel binary and configuration data, and IDA-based plugins for structure identification, control-structure recovery, and syscall reconstruction. The dynamic module operates across the ESXi target and the monitor

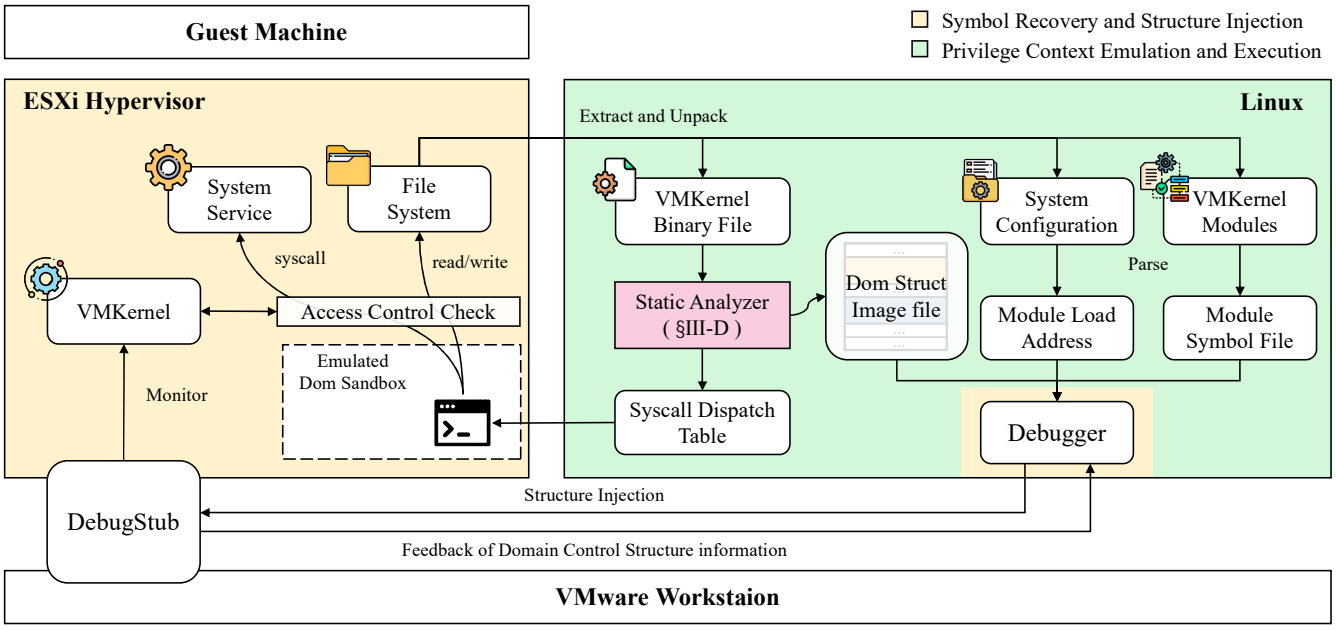


Fig. 6: Structure-Aware VMKernel Access-Control Mechanism Debugging

machine, a GDB plugin on the monitor side locates VMKernel and imports extracted symbols in a compatible format, while a privilege simulation tool on the target side creates custom user contexts and environment variables to emulate different domain identities, enabling detailed observation of privilege-related execution paths.

To distinguish between automated and manual components, we classify the workflow by the extent of human involvement. *Domain Control Structure Identification* and *Syscall Mechanism Reconstruction* are fully automated, allowing the framework to locate permission-checking functions, extract structure layouts, and rebuild syscall dispatch tables without human intervention, reducing the analysis time from several hours of manual effort to just a few minutes. In contrast, *Permission Validation Behavior Analysis* remains semi-automated, as it requires manual inspection and reasoning about runtime behaviors observed during dynamic debugging. This hybrid design combines automated large-scale structural inference with manual semantic verification, ensuring both analytical scalability and precision.

IV. Our Findings

In this section, we first provide a summary of our analysis results, followed by a detailed exposition of each of our findings.

A. Findings Overview

Our analysis was conducted on a workstation equipped with an Intel i9-14900 CPU and 64 GB of RAM, we spent four man-months building the debugging framework and performing reverse engineering. As illustrated in Fig. 7, our findings primarily consist of two key components:

Insecure Proprietary Syscall Mechanism. We found that the VMKernel's syscall mechanism includes three types of syscalls. VMKernel maintains separate dispatch tables for each type, with execution privileges recorded for each syscall. Based on these privileges, syscalls are grouped into different categories. However, despite this design, several security issues still exist in the actual implementation of the system call mechanism, including the presence of developer-intended high-privilege debug interfaces that could be misused.

Unprotected Structured Access Control Design. We found that VMKernel implements access control by maintaining a dedicated domain control structure for each security domain. For syscall control, the structure includes a bitmap field that indicates the set of syscalls accessible to the domain. For file access control, it maintains a linked list, where each node contains a file path and corresponding permission codes to determine whether read or write access is allowed. Notably, the memory region where this control structure resides is not protected by any enforced write restrictions, posing a risk of malicious tampering to bypass access restrictions.

B. The Syscall Mechanism of VMKernel

1) Overview of the Syscall Mechanism of VMKernel

Through reverse analysis, we identified that the syscall interface in the VMKernel is divided into three primary categories: `Linux64_Syscall`, `VMKSyscall`, and `VMKPrivateSyscall`. Specifically, we identified 197 entries under `Linux64_Syscall`, 284 entries under `VMKSyscall`, and 388 entries under `VMKPrivateSyscall`. These three categories of syscalls exhibit significant differences in terms of functionality and application scenarios. The `Linux64_Syscall` category is primarily intended to maintain compatibility with standard

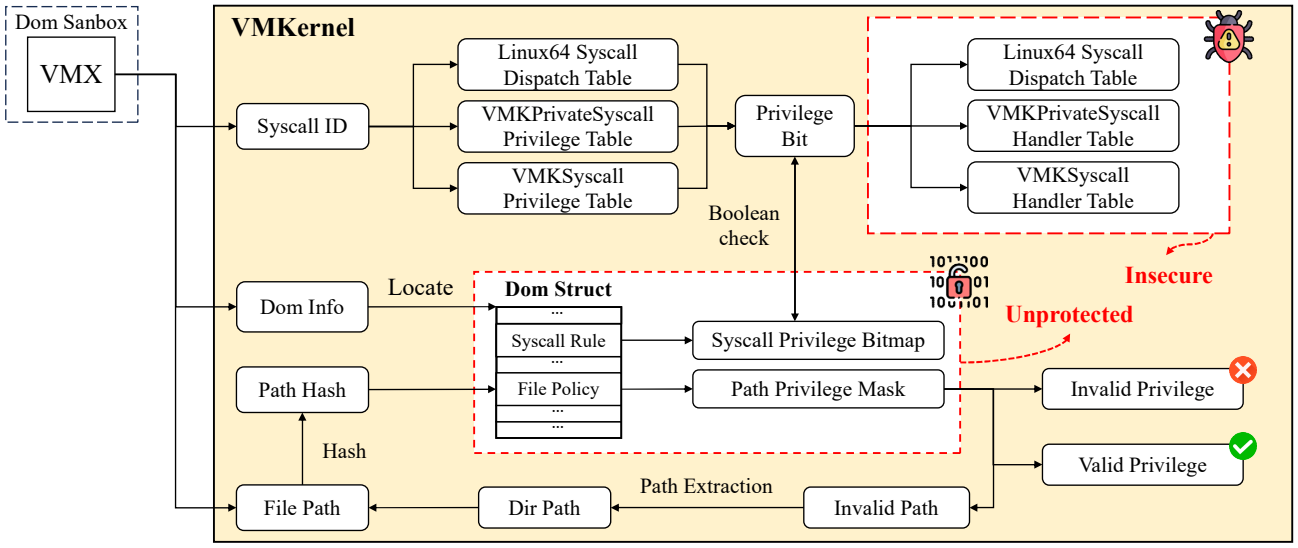


Fig. 7: Findings Overview

Linux syscall, offering basic functionalities such as file I/O, memory management, and signal handling. The `VMKSyscall` category serves as the core interface of the VMKernel, responsible for managing virtual machine life cycles, controlling low-level hardware, handling host resource scheduling, and supporting RPC communication services. These syscalls are widely used across various ESXi modules. For instance, in the open-source project VMtools [16], the names of some syscalls can be observed. In contrast, `VMKPrivateSyscall` refers to a set of private VMKernel interfaces primarily used for internal communication between kernel modules, control of testing procedures, and low-level hardware debugging. Since this category is not documented publicly, it is virtually inaccessible to external parties.

Finding 1: Developer-reserved debug syscalls in `VMKPrivateSyscall` expose high-privilege debug interfaces that can be abused for system compromise.

We discovered that the `VMKPrivateSyscall` retains several developer-specific debug interfaces, including syscalls that allow direct establishment of physical memory mapping regions or the modification of virtual machine states. These interfaces typically operate at high privilege levels. Once an attacker escapes from the VMX sandbox, they can invoke these syscalls to carry out attacks with severe impact on system integrity and isolation.

2) Structure of the Syscall Dispatch Table

At the implementation level, the `Linux64_Syscall` category employs a unified syscall dispatch table to manage the mapping between system calls and their privilege metadata. The table maintains a basic structural object for each syscall, containing three semantically meaningful fields: a handler pointer responsible for executing the corresponding syscall logic, a privilege bit used for runtime validation against the security-domain control structure of the calling process, and a syscall name string that records the symbolic identifier of each syscall to facilitate unified management and selective extension in later versions. Unlike standard Linux kernels that support the full syscall set, ESXi implements only a subset of frequently used syscalls. This table-based design allows centralized management of supported calls and simplifies future expansion. Unlike the structure-based approach used for `Linux64_Syscall`, the `VMKSyscall` and `VMKPrivateSyscall` categories, as illustrated in Fig. 7, maintain syscall information through two separate tables: one for privilege control bits and the other for handler function pointers. This design separates permission management from function implementation, offers greater flexibility in permission verification and access control. It facilitates modular security policy management and enables dynamic configuration of privileges.

3) Syscall Grouping Based on Privilege Control Bits

The syscall access-control policy within the VMKernel differs fundamentally from the Secure Computing Mode mechanism used in Linux. Instead of applying a per-process syscall whitelist filter, ESXi first categorizes processes into distinct security domains. It then classifies all syscalls into functional groups. By restricting each security domain to a specific set of permitted syscall groups, the VMKernel implements a fine-grained syscall access-control policy. As shown in Listing 2, the configuration illustrates the set of syscall groups authorized for the `globalVMDom` security domain.

We conducted an in-depth analysis of the syscall grouping mechanism built into VMKernel and found a one-to-one correspondence between syscall groups and permission bits, syscalls within the same group share the same permission bit. Based on this observation, we identified the permission bits associated with each of the three syscall

```
-s genericSys grant
-s vmxSys grant
-s ioctlSys grant
-s getpgidSys grant
-s getsidSys grant
-s vobSys grant
-s vsiReadSys grant
-s rpcSys grant
-s killSys grant
-s sysctlSys grant
-s syncSys grant
-s forkSys grant
-s forkExecSys grant
-s cloneSys grant
-s openSys grant
-s mprotectSys grant
-s iofilterSys grant
-s crossfdSys grant
-s pmemGenSys grant
-s keyCacheGenSys grant
-s vmfsGenSys grant
```

Listing 2: The Syscall rule of `globalVMDom`

categories and established a mapping to their corresponding syscall groups (see Appendix Table II). By comparing this mapping with the syscall groups accessible by the `globalVMDom` security domain (as shown in Listing 2), we found that this domain is primarily allowed to access syscalls in the `Linux64_Syscall` category, while the majority of `VMKSyscall` and `VMKPrivateSyscall` interfaces are restricted.

Furthermore, we conducted a functional classification analysis of the various syscall categories. We found that the syscalls accessible by the globalVMDom domain are primarily related to relatively safe operations such as VMX process management, input I/O, and system information queries. In contrast, syscalls involving write operations or executable behaviors are largely restricted. This observation indicates that VMKernel employs a fine-grained permission bit mechanism to effectively limit unnecessary syscall access, thereby strengthening the system's overall security posture and reducing the potential risk of virtual machine escape attacks.

Finding 2: We identified memory safety vulnerabilities in over a dozen syscalls, which can lead to memory corruption within the VMKernel.

We found that certain syscalls in VMKernel exhibit inherent security risks. Although VMKernel enforces management through dispatch tables and permission groups, it lacks fine-grained validation when parsing syscall parameters and accessing structure fields. This creates potential for memory corruption, which could compromise kernel stability and isolation. Due to ESXi's closed-source architecture and the absence of public syscall documentation, this area has received little security analysis. As a result, such vulnerabilities constitute a **previously undisclosed attack surface** within the VMKernel and warrant further investigation and defense.

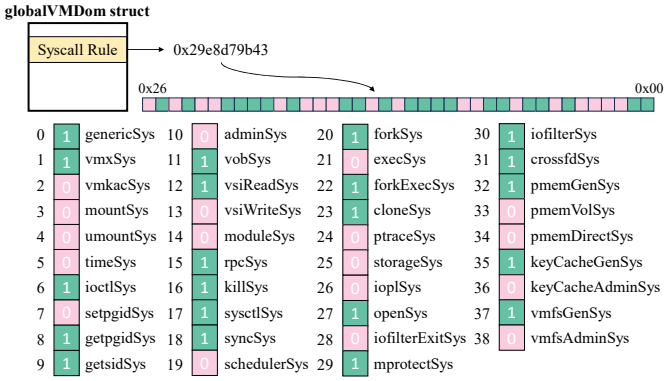


Fig. 8: Syscall Permission Bitmap of `globalVMDom`

C. Permission Checking Behavior Based on Domain Control Structure

1) Overview of the Access-Control Mechanism of VMKernel

The VMKernel maintains a dedicated structure for each security domain in the system, representing the set of operation privileges currently granted to that domain. This structure includes multiple permission fields covering syscall access, file I/O permissions, socket usage rights, and more. At runtime, it remains bound to the current process context and is accessed via the `gs` register. This design ensures that the VMKernel can efficiently retrieve the relevant privilege information when handling user requests.

2) Syscall Permission Validation Behavior

The control structure of each security domain includes a bitmap field that precisely governs access to syscalls. Fig. 8 illustrates the syscall permission field for the `globalVMDom` domain. By decoding this bitmap based on the mapping between privilege bits and syscalls defined in TABLE II, we can reconstruct the set of syscall groups that this domain is authorized to invoke. This result is consistent with Listing 2 and further confirms the correctness of our reverse analysis.

When a syscall is invoked, as illustrated in Fig. 7, the VMKernel first consults the syscall privilege table to determine the corresponding privilege bit index `i`. It then retrieves the permission bitmap field `syscall_rule[]` from the security-domain structure associated with the current execution context. The bit at position `i` indicates whether the syscall is authorized for that domain, if the bit is enabled, the request proceeds; if disabled, the VMKernel rejects the invocation. This bit-wise mapping mechanism provides an efficient and fine-grained means of enforcing syscall-level privilege control.

3) File Operation Permission Validation Behavior

Each security-domain control structure maintains a pointer to a linked list of file permission nodes that define the access rights of specific files within the domain. Each node in this list contains several semantically meaningful fields: two pointers link the node to its predecessor and successor in the list, a pair of identifiers record the node's unique ID and the hash of the corresponding file path, a permission code specifies the access privileges associated with that file, and a variable-length

field stores the actual path string. This linked design enables VMKernel to efficiently organize, traverse, and enforce fine-grained file access policies within each security domain.

When handling a file access request, the VMKernel retrieves the current security-domain structure and accesses its file permission list. It compares the hash of the target file path—computed using the DJB2 [17] algorithm—with the entries in this list to locate the corresponding permission node. If no exact match is found, the kernel performs hierarchical backtracking along the directory path until a relevant permission entry is identified. The access decision is then made by comparing the required operation type with the permission code stored in the matched node. This hierarchical lookup mechanism enables efficient enforcement of fine-grained file access control within each security domain.

4) Dynamic Adjustment of Permission Fields

We also identified a special control field within the domain control structure that dynamically influences access decisions during permission validation. When this field is set to certain specific values, the VMKernel may temporarily override standard access restrictions. Even if the security domain does not originally possess permission to invoke a particular syscall or access a given file path, the VMKernel can, during the permission check, dynamically set the corresponding syscall permission bit or modify the file permission code, thereby granting temporary access to the request. We speculate that this mechanism is designed to provide greater permission flexibility for high-privilege components such as VMX, enabling them to dynamically activate permissions at runtime based on operational requirements. This dynamic permission-overwriting mechanism reflects VMKernel's unique security strategy in coordinating trusted components.

Finding 3: The domain control structure lacks write protection, as well as integrity mechanisms such as signing or encryption, leaving it vulnerable to unauthorized modification and potential privilege escalation.

We found that VMKernel does not mark the memory region containing domain control structures as read-only; instead, it relies solely on locking mechanisms to prevent concurrent modifications, lacking strict enforcement against write operations. Moreover, the permission-related fields within these structures are not protected by integrity checks or cryptographic safeguards, leaving them vulnerable to tampering. As a result, an attacker with arbitrary kernel write capability could directly modify critical fields in the control structure, bypassing permission checks and enabling restricted syscalls or access to protected resources within the same security domain.

Moreover, our investigation confirmed the presence of certain systemic weaknesses previously noted in prior work [14]. For instance, VMKernel's implementation of KASLR [18] appears incomplete, with some memory segments exhibiting low entropy—creating a feasible window for address brute-forcing. Additionally, we observed that the system does not enable Supervisor Mode Access Prevention (SMAP), thereby allowing kernel-mode code to directly access user-space memory regions. These factors collectively expand the kernel's attack surface and introduce latent risks exploitable by adversaries.

V. Security Analysis and Practical Attacks

In this chapter, we first present the threat model considered in this work, then validate the identified security issues, and finally examine three practical attack scenarios along with their potential impacts.

A. Threat Model

We assume the goal of the attack is to break the sandbox implemented by VMKernel, which means the attacker has exploited a vulnerability, such as a remote code execution or a virtual machine escape flaw to gain the ability to execute arbitrary code within the security domain of the compromised component in the hypervisor. This assumption is practical and supported by real-world cases. For example, in CVE-2018-6981 [19] and CVE-2018-6982 [20], attackers leveraged memory corruption vulnerabilities in the vmxnet3 virtual NIC to achieve code execution in the context of the VMX process [4].

Building on this foothold, the attacker can further exploit memory safety flaws in the implementation of VMKernel syscalls to tamper with the domain control structure. This manipulation allows the attacker to bypass existing access-control mechanisms and escalate privileges or escape the sandbox environment. It is important to note that the attack path discussed in this paper focuses solely on memory vulnerabilities and permission control deficiencies within the VMKernel syscall implementation. We do not examine logical flaws or side-channel attacks targeting the VMKernel sandbox itself.

B. Result Validation

1) Syscall Vulnerability Discovery

We performed a security analysis of syscall handler functions by combining static analysis to identify potential targets with dynamic testing to verify exploitability. In the dynamic phase, we leveraged all identified syscalls for vulnerability exploration, crafting parameters to reach boundary values and issuing randomized syscall sequences to violate permission-checking logic. Through this process, we identified 14 memory safety vulnerabilities across syscall implementations, including stack overflows, information leaks, null pointer dereferences, and other flaw types. All discovered vulnerabilities were triggered by sequences of at most three syscalls. These results reveal systemic weaknesses and widespread fragility in the design and implementation of syscall mechanism within the VMKernel.

2) Manipulation of the Domain Control Structure

To verify whether the permission bitmap in VMKernel's access-control structure is writable and lacks integrity protection at runtime, we used our dynamic debugging framework to directly modify the control structure of the current

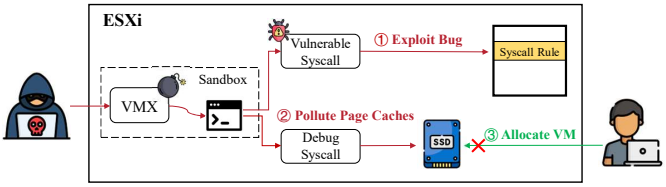
execution domain during a syscall. After booting ESXi, we set a breakpoint at the identified permission check function, issued a syscall under a `globalVMDom` identity, and altered the domain's syscall bitmap via the debugger. Execution then resumed without triggering any integrity violation or runtime error, confirming that the structure lacks proper write protection and is susceptible to unauthorized tampering.

C. Practical Attacks

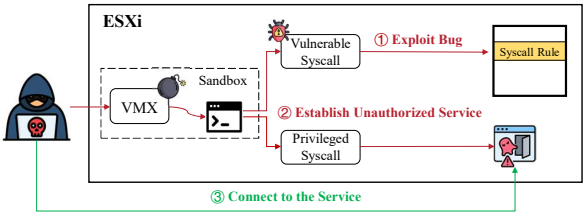
We conducted practical exploit experiments to assess the real-world impact of the identified security issues in VMKernel's access-control mechanism. By exploiting the memory-safety vulnerabilities in Section V-B1, we gained arbitrary read/write access within VMKernel. Building on our finding that control-structure fields are writable at runtime (Section V-B2) and the reverse-engineering results of Sections IV-C2 and IV-C3, We derived two attack primitives corresponding to the two categories of controllable fields, namely syscall privilege bits and file permission codes, which together represent the full exploitable scope of the reconstructed access-control design. *Primitive 1. Tampering with Syscall Privilege Bits:* By modifying the privilege bits associated with specific syscall groups in the access-control structure, an attacker can grant the current security domain the ability to invoke system calls that were originally disallowed. *Primitive 2. Tampering with File Path Permission Codes:* By altering the permission codes assigned to specific file paths, the attacker can enable unauthorized access to restricted files from within the current security domain. Based on the two attack primitives described above, we conducted the following three types of attack attempts and analyzed their potential security impacts. Specifically, *Primitive 1* was used to construct *Attack 1* and *Attack 2*, while *Primitive 2* was used to construct *Attack 3*. These three representative attack scenarios illustrate the practical exploitation potential and real-world impact of access-control vulnerabilities.

Attack 1: Denial-of-Service Attack via Privileged Debug Interface Exploitation. As shown in Fig. 9a, the attacker exploits *Primitive 1* to activate privileged debug syscalls within `VMKPrivateSyscall`, such as those that allow direct mappings to physical memory. With this capability, the attacker can repeatedly manipulate physical memory regions, gradually corrupting kernel data structures or polluting page caches. These actions may cause scheduling inconsistencies or memory corruption, ultimately leading to a denial-of-service (DoS) condition within the VMKernel.

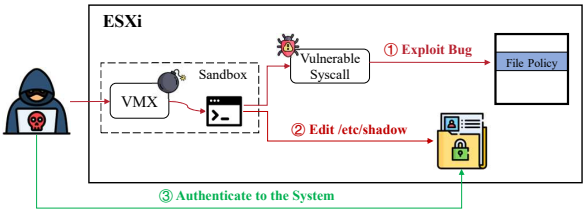
Attack 2: Persistent Access through Unauthorized Service Activation. As illustrated in Fig. 9b, the attacker again uses *Primitive 1* to re-enable the restricted `execve` syscall by altering its corresponding privilege bit. Normally limited to the host domain for legitimate binary execution, this syscall enables the attacker to spawn a lightweight remote service for external control. Once established, the service provides persistent and stealthy access to the host system, allowing the attacker to maintain control without further exploitation.



(a) Denial-of-Service Attack via Privileged Debug Interface Exploitation



(b) Persistent Access through Unauthorized Service Activation



(c) Credential Forgery for Persistent Host Access

Fig. 9: Illustration of three practical attacks under the assumed threat model.

Attack 3: Credential Forgery for Persistent Host Access. As shown in Fig. 9c, the attacker leverages *Primitive 2* to gain unauthorized read–write access to the `/etc/` directory. By modifying the `/etc/shadow` file, either injecting credentials for a custom account or replacing existing ones—the attacker forges valid authentication entries. This grants persistent privilege escalation through legitimate login or remote access, effectively bypassing the system's authentication safeguards.

We exploited a heap overflow in a syscall implementation to execute the three practical attacks within our framework, all achieving their intended effects. These results show that once the sandbox is bypassed, limited manipulation of the access-control mechanism can grant persistent control over host resources, revealing its fragility against such threats.

VI. Discussion

We reported the identified security issues to VMware. This section outlines VMware's official response and remediation status, presents our proposed mitigation strategies for strengthening VMKernel's access-control design, and extends the analysis to another closed-source Type-1 hypervisor to evaluate the generality of our approach.

A. Official Fix

We reported 14 vulnerabilities in the VMKernel syscall mechanism to VMware, all officially confirmed and rewarded through the Security Bounty Program ($3,000 each, totaling

TABLE I: List of Confirmed Vulnerabilities

No.	Vulnerability Type	Syscall Category	Status
1	NPD	Linux64_syscall	Fixed
2	Double Free	Linux64_syscall	Fixed
3	Stack Overflow	VMKSyscall	Fixed
4	Stack Overflow	VMKSyscall	Fixed
5	NPD	VMKSyscall	Fixed
6	NPD	VMKSyscall	Fixed
7	Heap Overflow	VMKSyscall	Fixed
8	Heap Overflow	VMKSyscall	Fixed
9	Heap Overflow	VMKSyscall	Fixed
10	Memory Leak	VMKSyscall	Fixed
11	Stack Overflow	VMKPrivateSyscall	Fixed
12	Stack Overflow	VMKPrivateSyscall	Fixed
13	Heap Overflow	VMKPrivateSyscall	Fixed
14	Memory Leak	VMKPrivateSyscall	Fixed

Note: NPD is Null Pointer Dereference

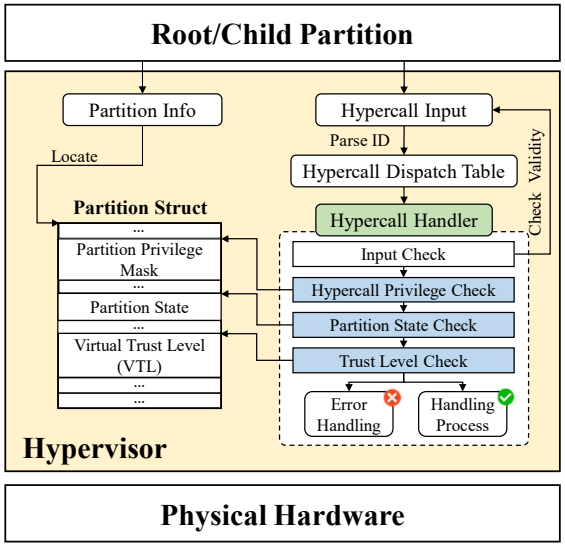


Fig. 10: The access control procedure of Hyper-V's `HvDeletePartition` Hypercall

$42,000; see Table I). These issues were patched in the latest ESXi release [21] without VMSA or CVE assignment. In addition, we also reported mechanism-level weakness in VMKernel's access-control design, which VMware acknowledged as a previously undisclosed and critical attack surface.

B. Mitigation Suggestions

Based on the current patch status, we further propose the following two mitigation strategies. We emphasize that these recommendations are intended to complement existing fixes. Our proposed measures provide additional protection with minimal overhead and can effectively mitigate the security issues and threats identified in this paper.

Minimize or eliminate debug syscall interfaces: As analyzed in Section IV, VMware ESXi retains certain private interfaces originally intended for internal debugging purposes (mainly `VMKPrivateSyscall`), which are attractive targets for exploitation. Since ordinary users do not rely on these syscalls, disabling or removing them in production builds will not affect functionality and can be achieved with minimal overhead. We recommend removing such interfaces from release versions to reduce the attack surface.

Enhance integrity protection for domain control structures: In our practical attack design, the core security risk stems from sensitive fields in VMKernel's domain control structures being writable at runtime without integrity protection. We recommend implementing lightweight integrity checks,such as compact hash verification or page-level write protection. These protection methods incur minimal overhead and would prevent unauthorized modification of control structures, thereby effectively blocking the two attack primitives and three concrete attacks demonstrated in this paper.

C. Method Generalization

To validate our method's generality, we applied it to another closed-source Type-1 hypervisor, Microsoft Hyper-V. Starting from the Hypercall execution flow, we analyzed its access-control mechanism. Although Hyper-V adopts a root/child partition model instead of ESXi's hypervisor/guest design, it similarly manages partition privileges through in-memory structures, allowing our domain-control–structure–oriented analysis(Section III-D) to be readily adapted to a partition-control–structure–oriented one.

In the analysis, we employed a *Partition(Domain) Control Structure Identification* (Section III-D1) method to locate the key privilege control structures in Hyper-V. By tracing the `vmcall` instruction, we identified the entry points of Hypercalls and then examined references to the context associated with the `gs` register to determine the corresponding permission-control structure for the current partition. We further traced how the fields of this structure were referenced and involved in comparison instructions to reconstruct the complete layout of the control structure. Since Hyper-V's Top-Level Functional Specification (TLFS) [22] publicly defines all hypercalls and their invocation IDs, we were able to skip the *Hypercall(Syscall) Mechanism Reconstruction* (Section III-D2) step required in ESXi analysis and proceed directly to *Permission Validation Behavior Analysis* (Section III-D3), reusing our debugging framework for further analysis.

Because Hyper-V performs distinct permission validation logic for each Hypercall, we take `HvDeletePartition` as a representative example and provide a detailed analysis of its permission-checking process during execution. As shown in Fig. 10, when this Hypercall is issued by a child partition, Hyper-V parses the caller's context and parameters, dispatches the request to the corresponding handler, and validates partition permissions. Owing to the multi-level privilege model defined in TLFS (e.g., VTL), permission checks are distributed across handlers. Our structure-oriented method remains effec-

tive under this loosely coupled design, accurately reconstructing Hyper-V's access-control mechanism and demonstrating strong adaptability.

VII. Related Works

Hypervisor Vulnerability Discovery. Hypervisor security has attracted growing attention in recent years. Most prior work applies fuzzing techniques to uncover vulnerabilities in hypervisor components, with a focus on virtual devices [23]–[28] and virtual CPUs [29], [30]. For example, HyperPill [26] adopts snapshot mutation–based fuzzing, while Truman [27] uses Linux driver knowledge to enhance seed generation. These methods have proven effective in discovering vulnerabilities in VMware ESXi. However, existing efforts largely target user-space components, leaving the hypervisor kernel and its internal logic insufficiently explored.

Kernel Security Analysis. Kernel security has been a long-standing focus of academic research. Tools like Syzkaller [31] and its derivatives [32]–[34] target Linux by auto-generating syscall sequences, while systems like AlphaExp [35] and KernelSnitch [36] enhance attack surface analysis through object modeling and side-channel techniques. Although recent efforts have extended to macOS [37], [38] and gVisor [39], VMware ESXi's VMKernel remains underexplored. This paper fills that gap by thoroughly analyzing VMKernel's syscall interface and privilege control mechanisms.

Breaking vSphere Security. In recent years, vSphere (VMware's virtualization suite that includes ESXi) has become a frequent target of successful exploits and an active subject of industrial security research [4]–[6], [40], [41]. Prior works have demonstrated virtual machine escapes via device vulnerabilities [4] and analyzed ESXi's architecture through reverse engineering [41]. However, most efforts focus on exploit chains, while VMKernel's internal privilege model remains understudied. This work fills that gap by comprehensively analyzing the security architecture and attack surface of VMKernel.

VIII. Conclusion

This paper presents a comprehensive analysis of the access control mechanism implemented in VMware ESXi's VMKernel. We develop a domain-control-structure oriented analysis approach and a structure-aware debugging framework to uncover how permissions are enforced and identify security flaws. Our findings reveal that ESXi's access control relies on centralized domain structures lacking write protection, and multiple syscall handlers suffer from memory safety vulnerabilities. We demonstrate three exploit scenarios to bypass access restrictions using these flaws. In total, we report 14 vulnerabilities to VMware, all of which were confirmed and fixed, earning our team $42,000 in bug bounties and official acknowledgment from VMware.

Acknowledgment

We thank the anonymous reviewers for their insightful comments on our work. This work is supported by the National Natural Science Foundation of China (Grant No. U22B2025, 62172093, and 62402147), the "Pioneer" and "Leading Goose" R&D Program of Zhejiang, China (Grant No. 2025C02261, 2025C02263), Zhejiang Provincial Key Laboratory for Sensitive Data Security Protection and Confidentiality Management No. 2024E10048.

Ethics Consideration

This study presents a systematic analysis and vulnerability assessment of the VMKernel access-control mechanisms within the VMware ESXi platform, aiming to uncover potential security risks in real-world deployments and to promote architectural hardening of virtualization systems. Throughout the research, we adhered strictly to responsible disclosure practices: all identified vulnerabilities were promptly reported to VMware through official channels and were only documented and published after confirmation and remediation. The paper does not disclose any unpublished vulnerabilities or attack-ready methods. All validation experiments were conducted in isolated environments, without involving third-party data or unauthorized operations.

References

[1] Liquid Web, "Bare metal hypervisors: What they are & when to use one," https://www.liquidweb.com/blog/bare-metal-hypervisors/, 2023, accessed: 2025-08-07.

[2] Zscaler ThreatLabz, "Shining light on the dark angels ransomware group," Blog post, Zscaler, Oct. 2024. [Online]. Available: https://www.zscaler.com/blogs/security-research/shining-light-dark-angels-ransomware-group

[3] VMware Security Team, "Vmware and pwn2own 2025 berlin," https://blogs.vmware.com/security/2025/05/vmware-and-pwn2own-2025-berlin.html, May 2025, accessed: 2025-08-06.

[4] H. Zhao, Y. Zhang, K. Yang, and T. Kim, "Breaking turtles all the way down: An exploitation chain to break out of VMware ESXi," in 13th USENIX Workshop on Offensive Technologies (WOOT 19), 2019.

[5] Z. Sialveras, "Bugs of yore: A bug hunting journey on vmware's hypervisor," Presented at Black Hat USA 2024, Las Vegas, NV, Aug. 2024, https://i.blackhat.com/BH-US-24/Presentations/US24-Sialveras-Bugs-Of-Yore-Wednesday.pdf.

[6] Y. Jiang and X. Ying, "Urb excalibur: The new vmware all-platform vm escapes," Presented at Black Hat Asia 2024, Singapore, Apr. 2024, https://i.blackhat.com/Asia-24/Presentations/Asia-24-Jiang-URB-Excalibur-The-New-VMware-All-Platform-VM\-Escapes.pdf.

[7] VMware Community, "Using debugstub to debug a guest linux kernel," VMware Fusion Discussions Forum, 2014, https://communities.vmware.com/t5/VMware-Fusion-Discussions/Using-debugStub-to-debug-a-guest-linux-kernel/td-p/394906.

[8] VMware. (2024) Idc business value study: Vmware cloud foundation delivers 564% 3-year roi with a 10-month payback. VMware Cloud Foundation Blog. [Online]. Available: https://www.vmware.com/docs/vmw-white-paper-business-value-of-cloud-foundation

[9] VMware, Inc., "VMware ESXi Architecture Overview," https://microage.com/wp-content/uploads/2016/02/ESXi_architecture.pdf, 2016, accessed: 2025-08-01.

[10] "CVE-2021-21974: VMware ESXi Service Location Protocol (SLP) Heap Overflow Vulnerability," https://www.cve.org/CVERecord?id=CVE-2021-21974, MITRE Corporation, 2021, accessed: 2025-11-06.

[11] MITRE Corporation, "CVE-2024-22252: VMware ESXi, Workstation, and Fusion XHCI USB Controller Use-After-Free Vulnerability," https://www.cve.org/CVERecord?id=CVE-2024-22252, 2024.

[12] T. L. Foundation, "Seccomp filter," https://www.kernel.org/doc/html/v4.13/userspace-api/seccomp_filter.html, The Linux Foundation, 2025, accessed: 2025-02-10.

[13] Reddit User Community, "Virtualized esxi on kvm?" https://www.reddit.com/r/virtualization/comments/17h4rpg/virtualized_esxi_on_kvm/?tl=zh-hans, Reddit, 2023, discussion thread on r/virtualization, accessed on November 10, 2025.

[14] Zero Day Initiative. (2023) CVE-2022-31696: An Analysis of a VMware ESXi TCP Socket Keepalive Type Confusion LPE. [Online]. Available: https://www.zerodayinitiative.com/blog/2023/6/21/cve-2022-31696-an-analysis-of-a-vmware-esxi-tcp-socket-keepalive\-type-confusion-lpe

[15] J. Zhao, K. Zhu, L. Yu, H. Huang, and Y. Lu, "Yama: Precise opcode-based data flow analysis for detecting php applications vulnerabilities," IEEE Transactions on Information Forensics and Security, 2025.

[16] VMware, Inc., "open-vm-tools: Official repository," https://github.com/vmware/open-vm-tools, 2025.

[17] S. Shah and A. Shaikh, "Hash based optimization for faster access to inverted index," in 2016 International Conference on Inventive Computation Technologies (ICICT), vol. 1. IEEE, 2016, pp. 1–5.

[18] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "Kaslr is dead: long live kaslr," in International Symposium on Engineering Secure Software and Systems. Springer, 2017, pp. 161–176.

[19] National Vulnerability Database, "Cve-2018-6981 detail," https://nvd.nist.gov/vuln/detail/CVE-2018-6981, 2018.

[20] ——, "Cve-2018-6982 detail," https://nvd.nist.gov/vuln/detail/CVE-2018-6982, 2018.

[21] VMware by Broadcom, "VMware Cloud Foundation 9.0 Release Notes," 2025. [Online]. Available: https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vcf-9-0-and-later/9-0/release-notes/vmware-cloud-foundation-90-release-notes.html

[22] Microsoft Corporation, "Hyper-v top-level functional specification (tlfs)," https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/tlfs, 2025, version 7.x. Accessed: 2025-11-06.

[23] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. Wu, "V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing," in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 2197–2213.

[24] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, "Morphuzz: Bending (input) space to fuzz virtual devices," in 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 1221–1238.

[25] Q. Liu, F. Toffalini, Y. Zhou, and M. Payer, "Videzzo: Dependency-aware virtual device fuzzing," in 2023 IEEE Symposium on Security and Privacy (SP). IEEE, 2023, pp. 3228–3245.

[26] A. Bulekov, Q. Liu, M. Egele, and M. Payer, "Hyperpill: Fuzzing for hypervisor-bugs by leveraging the hardware virtualization interface," in 33rd USENIX Security Symposium (USENIX Security 24), 2024.

[27] Z. Ma, Q. Liu, Z. Li, T. Yin, W. Tan, C. Zhang, and M. Payer, "Truman: Constructing device behavior models from os drivers to fuzz virtual devices," in 32nd Annual Network and Distributed System Security Symposium, NDSS, 2025, pp. 24–28.

[28] Z. Zhang, G. Pan, R. Wang, Y. Tao, Z. Pan, C. Tu, M. Zhang, Y. Li, Y. Shen, and C. Wu, "Insvdf: Interface-state-aware virtual device fuzzing," in 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). IEEE Computer Society, 2025, pp. 727–727.

[29] P. Fonseca, X. Wang, and A. Krishnamurthy, "Multinyx: a multi-level abstraction framework for systematic analysis of hypervisors," in Proceedings of the Thirteenth EuroSys Conference, 2018, pp. 1–12.

[30] X. Ge, B. Niu, R. Brotzman, Y. Chen, H. Han, P. Godefroid, and W. Cui, "Hyperfuzzer: An efficient hybrid fuzzer for virtual cpus," in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 366–378.

[31] Google, "syzkaller: Linux kernel fuzzer," https://github.com/google/syzkaller, 2024.

[32] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel." in NDSS, 2020.

[33] S. Pailoor, A. Aday, and S. Jana, "{MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation," in 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 729–743.

[34] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "{StateFuzz}: System {Call-Based}{State-Aware} linux driver fuzzing," in 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 3273–3289.

[35] R. Wang, K. Chen, C. Zhang, Z. Pan, Q. Li, S. Qin, S. Xu, M. Zhang, and Y. Li, "{AlphaEXP}: An expert system for identifying {Security-Sensitive} kernel objects," in 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 4229–4246.

[36] L. Maar, J. Juffinger, T. Steinbauer, D. Gruss, and S. Mangard, "Kernel-snitch: Side-channel attacks on kernel data structures," in Network and Distributed System Security Symposium 2025: NDSS 2025, 2025.

[37] T. Yin, Z. Gao, Z. Xiao, Z. Ma, M. Zheng, and C. Zhang, "{KextFuzz}: Fuzzing {macOS} kernel {EXTensions} on apple silicon via exploiting mitigations," in 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 5039–5054.

[38] Z. Cai, J. Zhu, W. Shen, Y. Yang, R. Chang, Y. Wang, J. Li, and K. Ren, "Demystifying pointer authentication on apple m1," in 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 2833–2848.

[39] Y. Li, Y. Chen, S. Ji, X. Zhang, G. Yan, A. X. Liu, C. Wu, Z. Pan, and P. Lin, "G-fuzz: A directed fuzzing framework for gvisor," IEEE Transactions on Dependable and Secure Computing, vol. 21, no. 1, pp. 168–185, 2023.

[40] H. Zheng, Z. Li, and Y. Liu, "vcenter lost: How the dcerpc vulnerabilities changed the fate of esxi," Presented at Black Hat Asia 2024, Singapore, Apr. 2024, http://i.blackhat.com/Asia-25/Asia-25-Zheng-vCenter-Lost-How-the-DCERPC-Vulnerabilities\-Changed-the-Fate-of-ESXi.pdf.

[41] J. Huang, H. Zheng, and Y. Liu, "Dragon slayingguide: Bug hunting in vmware device virtualization," Presented at DEFCON 32, Las Vegas, NV, Aug. 2024, https://media.defcon.org/DEFCON32/DEFCON32presentations/DEFCON32-\JiaQingHuangHaoZhengYueLiu-DragonSlayingGuideBugHuntingIn\VMwareDeviceVirtualization.pdf.

APPENDIX A
STATISTICS OF SYSCALL PERMISSION GROUPS

TABLE II: Mapping of Syscall Groups and Syscall Counts by Permission Bit

Control Bit	Privilege Group	L64	VMK	VMKP
0	genericSys	0	24	4
1	vmxSys	0	198	329
2	vmkaeSys	0	1	17
3	mountSys	1	1	0
4	umountSys	1	1	0
5	timeSys	0	6	0
6	ioctlSys	1	1	0
7	setpgidSys	1	1	0
8	getpgidSys	1	1	0
9	getsidSys	1	1	0
10	adminSys	0	0	0
11	vobSys	0	10	10
12	vsiReadSys	0	0	10
13	vsiWriteSys	0	0	1
14	moduleSys	0	0	0
15	rpcSys	0	0	0
16	killSys	3	3	0
17	syscallSys	0	1	1
18	sysSys	1	1	0
19	schedulerSys	0	1	0
20	execSys	1	1	0
21	forkSys	1	1	0
22	forkExecSys	0	1	1
23	cloneSys	1	0	0
24	ptraceSys	1	0	0
25	storageSys	0	0	0
26	ioplSys	0	0	0
27	openSys	2	1	0
28	iofilterExitSys	0	0	1
29	mprotectSys	1	0	1
30	iofilterSys	0	0	14
31	crossfdSys	0	0	5
32	pmemGenSys	0	0	12
33	pmemVolSys	0	0	1
34	pmemDirectSys	0	0	0
35	keyCacheGenSys	0	0	0
36	keyCacheAdminSys	0	1	1
37	vmfsGenSys	0	1	0
38	vmfsAdminSys	0	1	0
0xff	Unclassified	170	0	0
Total		**197**	**284**	**388**

Note: L64 is Linux64_Syscall; VMK is VMKSyscall; VMKP is VMKPrivateSyscall.

Syscalls classified as "Unclassified" are defined by a privilege control bit set to `0xff`, which causes them to be bypassed during permission validation. All such syscalls fall under the `linux64_Syscall` category and include standard, commonly used syscalls such as `open`, `close`, and `mmap`. Given their functional analysis, we infer that these syscalls do not involve access to sensitive resources and therefore do not require inclusion in the privilege grouping mechanism.