INSTITUTO POLITÉCNICO
DE VIANA DO CASTELO

ESTG

INSTITUTO POLITÉCNICO
DE VIANA DO CASTELO

RESIDUAL MEMORY OF VIRTUAL MACHINES VISIBLE TO THE HOST: AN EMPIRICAL STUDY ON ISOLATION FAULTS

Marco Pinto Moreira

Escola Superior de Tecnologia e Gestão

RESIDUAL MEMORY OF VIRTUAL MACHINES VISIBLE TO THE HOST: AN EMPIRICAL STUDY ON ISOLATION FAULTS

2026

Instituto Politécnico
de Viana do Castelo

Residual Memory of Virtual Machines Visible to the Host: An Empirical Study on Isolation Faults

Autor

Marco Pinto Moreira

Trabalho orientado por

Professor Pedro Filipe Cruz Pinto

Professor António Alberto dos Santos Pinto

Mestrado em Cibersegurança

Fevereiro de 2026

Mestrado em Cibersegurança
Master in Cybersecurity

Residual Memory of Virtual Machines Visible to the Host: An Empirical Study on Isolation Faults

a master's thesis authored by

Marco Pinto Moreira

and supervised by

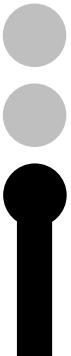
Pedro Filipe Cruz Pinto
Professor Adjunto, Instituto Superior de Engenharia do Porto

António Alberto dos Santos Pinto
Professor Coordenador, Instituto Politécnico do Porto

This thesis was submitted in partial fulfilment of the requirements for the Master's degree in Cybersecurity at the Instituto Politécnico de Viana do Castelo

ipvc

27 of February, 2026



Abstract

Virtual Machines (VMs) are widely used in virtualized environments to provide isolation between guests sharing the same physical hardware. By abstracting Central Processing Unit (CPU), memory, storage and networking resources, hypervisors enable efficient resource consolidation while maintaining separation between machines.

Memory plays a central role in this model, as it temporarily stores active programs and execution data. In virtualized systems, physical memory is dynamically allocated, reclaimed and reassigned across VM lifecycle events, including execution, shutdown and reboot. These operations rely on the assumption that isolation mechanisms prevent residual data from persisting beyond the intended execution context.

However, the effectiveness of memory isolation and memory clearing mechanisms remains an open concern. If volatile memory pages are not properly sanitized before being reassigned, residual data generated by one VM can remain present in host memory beyond its lifecycle. This challenges common assumptions regarding automatic memory clearing and can increase the exposure surface of sensitive information in shared infrastructures. The risk is further amplified by the fact that application data is typically not encrypted while residing in volatile memory.

This research examines how memory content created within a VM persists across various stages of the VM lifecycle and examines whether such data remains observable at the host level. The study evaluates memory behavior during running, shutdown, and reboot transitions to analyze patterns of retention, relocation, and duplication.

To study this behavior, controlled experiments are conducted by injecting 25 million unique, identifiable tags into the memory of VMs. Host memory dumps are collected at multiple VM execution stages and analyzed using automated scanning techniques. The evaluation is performed across three virtualization platforms: Microsoft Hyper-V,

VMware, and Oracle VirtualBox.

The results indicate that approximately 99% of injected tags are detectable in host memory while the VM is running. After shutdown and reboot transitions, a substantial proportion of tags remain present, with persistence rates varying across platforms. Although page-level retention decreases over lifecycle stages, most tags are not fully eliminated but instead relocated to different host physical pages. Additionally, memory duplication is observed, with identical tags appearing in multiple host memory regions despite being inserted only once.

The analysis shows that memory content is not fully removed after a VM shutdown. Differences are also observed between virtualization platforms in how memory is managed and retained. The results obtained highlight privacy risks related to volatile memory in virtualized environments and emphasize the importance of improving memory isolation and sanitization mechanisms, particularly in shared infrastructures.

Keywords: Data Persistency. Privacy. Memory. Virtual Machine. Host. Residual Data.

Resumo

Máquinas virtuais são amplamente utilizadas em ambientes virtualizados para fornecer isolamento entre sistemas que partilham o mesmo hardware físico. Ao abstrair o processador, a memória, o armazenamento e a rede, os hipervisores permitem uma consolidação eficiente de recursos, mantendo simultaneamente a separação entre máquinas.

A memória desempenha um papel central neste modelo, uma vez que armazena temporariamente programas ativos e dados em execução. Em sistemas virtualizados, a memória física é dinamicamente alocada, libertada e reatribuída ao longo dos diferentes eventos do ciclo de vida das máquinas virtuais, incluindo execução, encerramento e reinício. Estas operações baseiam-se no pressuposto de que os mecanismos de isolamento impedem que dados residuais persistam para além do contexto de execução previsto.

No entanto, a eficácia dos mecanismos de isolamento e de limpeza de memória continua a ser uma preocupação em aberto. Se as páginas de memória volátil não forem devidamente apagadas antes de serem reatribuídas, dados residuais gerados por uma máquina virtual podem permanecer na memória do anfitrião para além do seu ciclo de vida. Isto desafia suposições comuns relativas à limpeza automática da memória e pode aumentar a superfície de exposição de informação sensível em infraestruturas partilhadas. O risco é ainda amplificado pelo facto de os dados das aplicações, em regra geral, não se encontrarem encriptados enquanto residem em memória volátil.

Este trabalho investiga de que forma o conteúdo de memória criado no interior de uma máquina virtual persiste ao longo das várias fases do seu ciclo de vida e analisa se esses dados permanecem observáveis ao nível do anfitrião. O estudo avalia o comportamento da memória durante as transições de execução, encerramento e reinício, de modo a identificar padrões de retenção, relocalização e duplicação.

Para estudar este comportamento, foram conduzidas experiências controladas através

da injeção de 25 milhões de etiquetas únicas identificáveis na memória das máquinas virtuais. Foram recolhidos *dumps* de memória do anfitrião em múltiplas fases de execução da máquina virtual e analisados, recorrendo a técnicas automatizadas. A avaliação foi realizada em três plataformas de virtualização: Microsoft Hyper-V, VMware e Oracle VirtualBox.

Os resultados indicam que aproximadamente 99% das etiquetas injetadas são detetáveis na memória do anfitrião enquanto a máquina virtual se encontra em execução. Após as transições de encerramento e reinício, uma proporção substancial das etiquetas permanece presente, variando as taxas de persistência entre plataformas. Embora a retenção ao nível da página diminua ao longo das fases do ciclo de vida, a maioria das etiquetas não é totalmente eliminada, sendo antes relocalizada para diferentes páginas físicas do anfitrião. Adicionalmente, foi observada duplicação de memória, com etiquetas idênticas a surgirem em múltiplas regiões da memória do anfitrião, apesar de terem sido inseridas apenas uma vez.

A análise demonstra que o conteúdo da memória não é completamente removido após o encerramento de uma máquina virtual. Foram igualmente observadas diferenças entre plataformas de virtualização no que respeita à forma como a memória é gerida e retida. Os resultados obtidos evidenciam riscos de privacidade associados à memória volátil em ambientes virtualizados e reforçam a importância de melhorar os mecanismos de isolamento e de sanitização de memória, particularmente em infraestruturas partilhadas.

Palavras-chave: Persistência de Dados. Privacidade. Memória. Máquina Virtual. Dados Residuais.

Aknowledgements

I would like to thank my supervisors, Prof. Pedro Pinto and Prof. António Pinto, for their continuous guidance, support, and mentorship throughout this journey. Their expertise, feedback, and encouragement were fundamental to the development of this work.

I would like to thank IPVC–ESTG for providing the academic environment, institutional support, and resources that made this research possible. The conditions offered by the institution were essential to my academic growth and to the successful completion of this journey.

I would like to thank my company, Immera, for the flexibility and support while I embraced this challenge. I am especially thankful to the entire team for their support and encouragement.

I would also like to thank my dear friend Raquel Rodrigues, who, despite the distance, remained a constant source of support and motivation throughout this process.

Finally, I would like to thank my family for their unconditional love, patience, and constant support. Their encouragement enabled me to reach this milestone. I am particularly grateful to my parents, my sister, my brother-in-law, and my grandparents for their motivation throughout this journey.

To my girlfriend, Inês Lopes, thank you for your continuous support, understanding, and encouragement. Your presence and motivation meant more than words can express.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

AWS Amazon Web Services

CPU Central Processing Unit

CSV Comma-Separated Values

DRAM Dynamic random access memory

FaaS Function as a Service

HTML Hypertext Markup Language

IaaS Infrastructure as a Service

IP Internet Protocol

KB Kilobyte

KVM Kernel-based Virtual Machine

OS Operating System

PaaS Platform as a Service

RAM Random Access Memory

SaaS Software as a Service

VM Virtual Machine

XaaS Anything/Everything as a Service

Chapter 1

Introduction

Virtualization has become a critical infrastructure for businesses and individuals due to its scalability, flexibility and cost-effectiveness [27, 3]. By virtualizing hardware resources such as Central Processing Unit (CPU), memory, storage devices, network interfaces and other input/output components, infrastructure providers enable multiple users to share the same physical system while operating in logically isolated environments known as Virtual Machines (VMs). Although this model improves hardware utilization, it also introduces security and privacy challenges related to data confidentiality and isolation [30, 40].

1.1 Problem Statement and Motivation

Virtualized environments rely on the sharing of fundamental hardware resources, including CPU, memory, storage subsystems and network interfaces. While the hypervisor manages each of these resources, memory plays a particularly critical role due to its direct relation to program execution and data handling. In both VMs and host systems, volatile memory is used by the Operating System (OS) to temporarily store active programs, data required for their execution and other data. Volatile memory provides fast access to data, enabling efficient multitasking and high system performance. However, its volatile and shared nature also makes it a critical component from a security and privacy perspective, as sensitive data residing in memory can be exposed through memory dumps or microarchitectural attacks [38]. If the data stored in the memory used by one VM leaks into the memory of another or is directly accessed by the host machine, this could result in

serious security issues, such as unauthorized access to sensitive data or information leaks. This issue raises distinct concerns across different VM lifecycle states. While the VM is running, volatile memory exposure can underestimate isolation guarantees between host and guest. During shutdown or reboot states, memory can be released and reassigned, potentially allowing previously allocated data to persist beyond the intended execution context.

Thus, memory isolation should be enforced to ensure that data stored in the volatile memory of one VM cannot be accessed or leaked into another VM or into the host machine. The effectiveness of the memory-isolation mechanisms used by infrastructure providers is not always clear and there is a need to examine whether and how they are actually applied. This is essential to maintain the privacy and security of sensitive information, as virtualized environments often host multiple clients, each with their data and applications [7].

The following set of research questions guides this work:

RQ1: How is the information inserted into memory in a VM, handled, exposed and persisted in host environments?

RQ2: When is memory cleared during the virtualization processes and how does this influence what remains accessible afterwards?

RQ3: What security and privacy implications arise from retention or leakage of memory content in virtualized environments?

1.2 Objectives

This research aims to evaluate the persistence of volatile memory associated with VMs within host systems across different virtualization platforms. The study focuses on understanding how memory content generated inside a VM evolves throughout its lifecycle, including execution, shutdown and reboot states. By examining these transitions, the research seeks to determine whether memory-isolation mechanisms effectively prevent residual data from remaining accessible after VM state changes.

Additionally, this work seeks to examine how memory content is redistributed within host memory when persistence occurs. This includes analyzing whether data remains

confined to specific memory regions or if it is relocated over time.

Finally, the research also intends to identify differences in memory management practices and to evaluate their implications for isolation and data confidentiality in virtualized environments.

1.3 Scientific Contributions

The work presented provides the following scientific contributions:

1. Marco Moreira. "Assess Memory Isolation Mechanisms Used on Cloud Providers". In: *CyberSec 25*. 7 February, Viana do Castelo, Portugal, 2025 - In this work, a collection and analysis of different memory dumps was made to check if any data from other VM appeared. The objective was to assess the effectiveness of memory isolation mechanisms used by cloud providers to ensure data security in virtualized environments. In a Google Cloud VM, differences between memory dumps were observed, indicating the potential for data residuals.

2. Marco Moreira, Pedro Pinto and António Pinto "Host-Visible Residual Memory in Virtual Machines: An Empirical Study on Isolation Faults". In: *CSF 2026*. (Submitted) 26th-29th July, Lisbon, Portugal, 2026 - This study presents a systematic empirical analysis of residual volatile memory in virtualized environments. It evaluates memory persistence across VM lifecycle events on widely used hypervisors and provides a page-level characterization of memory relocation and duplication on the host system.

1.4 Organization

This document is organized as follows. Chapter 2 provides the background on virtualization, including hypervisors, virtualization platforms, resource management, memory virtualization and cloud deployment. Chapter 3 reviews related work and compares other studies relevant to virtualized memory and virtualized environments. Chapter 4 discusses memory isolation in virtualized systems and presents the threat model, along with the scope and limitations of this work. Chapter 5 describes the research methodology, includ-

ing the host and VM setup, the tag injection program, memory acquisition procedures and the analysis process. Chapter 6 presents the experimental results, covering memory persistence, allocation behavior, memory mapping and duplication. Chapter 7 presents the discussion structured around the research questions and privacy implications. Finally, Chapter 8 concludes the work and outlines potential directions for future research.

Chapter 2

Background on Virtualization

Virtualization has revolutionized the way computing resources are delivered and consumed. This technique allows multiple OSs to run on the same physical machine at the same time. This is achieved through a software layer called a hypervisor, which controls access to the hardware resources. Each VM runs in an isolated environment and behaves as if it were executing on dedicated hardware. Virtualization improves hardware utilization and flexibility by allowing resources to be dynamically allocated and reclaimed. VMs can be started, stopped and rebooted with little effort.

The fundamental principles of virtualization were formally defined by Popek and Goldberg [33]. Their work describes the conditions under which a computer system can efficiently support VMs while preserving correctness and isolation. These principles remain the foundation of modern virtualization systems.

Figure 2.1 presents a simplified view of a virtualized system architecture. Multiple VMs execute on the same physical host, each running its own guest OS and applications. The hypervisor layer is positioned directly above the host's hardware and is responsible for mediating access to physical resources, including CPU, memory and storage.

Although the hardware is shared, the hypervisor enforces logical isolation between VMs, ensuring that each guest operates as an independent execution environment while coexisting on the same host system.

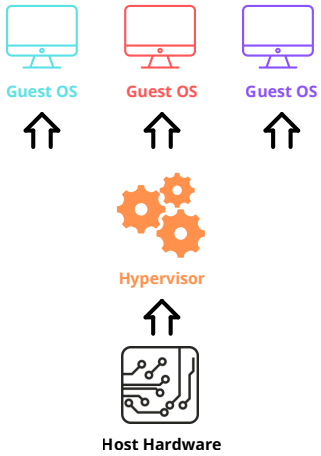


Figure 2.1: Simplified architecture of a virtualized system.

2.1 Hypervisors

A hypervisor is a software layer that enables virtualization by allowing multiple VMs to execute concurrently on a single physical system. It abstracts the hardware and presents each VM with a virtualized view of system resources. The hypervisor is responsible for allocating and scheduling hardware resources such as the CPU, memory, storage and input/output devices, while maintaining separation between guest environments.

Hypervisors are commonly classified into two main categories. In Figure 2.2, Type-1 hypervisors, also referred to as bare-metal hypervisors, execute directly on the physical hardware and are typically deployed in server and cloud infrastructures. Type-2 hypervisors operate on top of a host OS and are generally used in desktop or development environments. Although their architectural placement differs, both types perform similar core functions, including resource mediation, isolation enforcement and control of VM lifecycle operations [4]. In this model, OSs depend on both the hypervisor and the host OS for resource access. While both architectures enable virtualization and logical isolation, their structural placement influences resource mediation, performance characteristics and system complexity.

Table 2.1 shows the main differences between Type-1 and Type-2 hypervisors. Regarding architectural placement, the Type-1 hypervisors execute directly on the hardware, while Type-2 hypervisors operate on top of a host OS. This structural difference influences performance characteristics, resource management and the system's attack surface. In par-

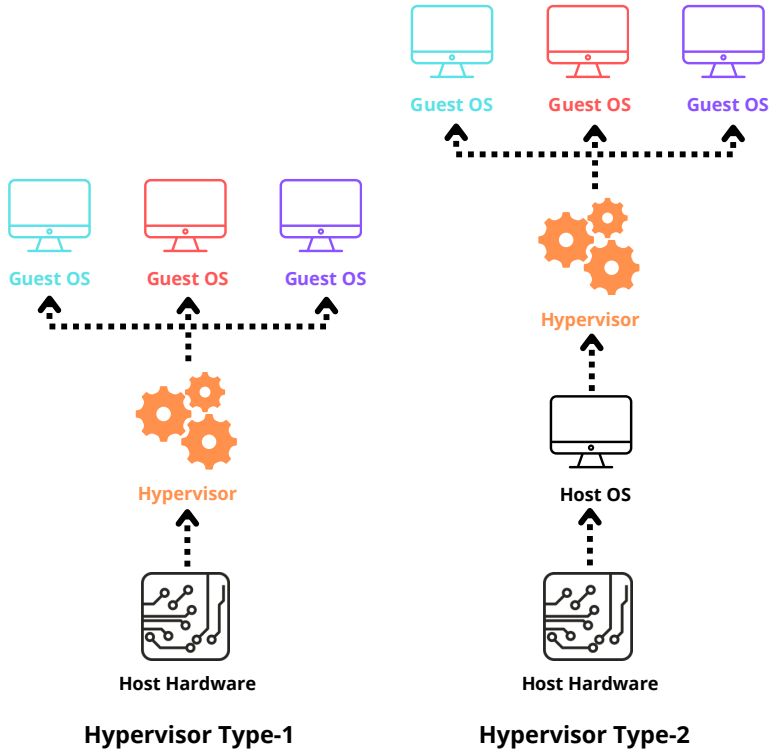


Figure 2.2: Comparison between Type-1 and Type-2 hypervisors.

ticular, Type-1 hypervisors typically provide more direct control over hardware resources and imply lower performance cost, while Type-2 hypervisors can introduce additional execution cost due to the presence of the host OS layer. Despite these differences, both types are responsible for enforcing logical isolation between VMs and managing access to shared physical resources. At the hardware level, modern hypervisors rely on supported virtualization features to intercept and manage privileged instructions executed by VM OSs.

2.2 Virtualization Platforms

Virtualization platforms represent complete software ecosystems that implement hypervisor technology alongside management, storage and networking components. While the hypervisor provides the core mechanism for hardware abstraction and isolation, virtualization platforms integrate additional tools for deployment, monitoring, resource allocation and lifecycle management of VMs. These platforms differ in architectural design, licensing model and deployment context, ranging from enterprise data center solutions to

Table 2.1: Comparison between Type-1 and Type-2 hypervisors.

	Type-1 (Bare-Metal)	**Type-2 (Hosted)**
Execution Layer	Runs directly on physical hardware	Runs on top of a host OS
Typical Deployment	Data centers, cloud infrastructures, enterprise servers	Desktop environments, development and testing systems
Performance	Generally higher due to direct hardware access	Lower due to the additional host OS layer
Isolation Model	Isolation enforced directly by the hypervisor	Isolation enforced by the hypervisor but dependent on the host OS
Resource Management	Direct control over CPU, memory and I/O resources	Resource management through the host OS
Security Surface	Smaller attack surface	Larger attack surface due to the host OS
Examples	Microsoft Hyper-V, Xen	Oracle VirtualBox, VMware Workstation

lightweight desktop-oriented systems.

Among widely adopted virtualization platforms, several implementations are commonly used in both enterprise and academic contexts. Microsoft Hyper-V[1] and VMware[2] solutions provide enterprise-oriented virtualization environments, typically deployed in server infrastructures and cloud systems. Open-source alternatives such as Kernel-based Virtual Machine (KVM)[3] and Proxmox VE[4] are widely used in Linux-based infrastructures. In desktop and development environments, solutions such as Oracle VirtualBox[5], VMware Workstation and QEMU[6] provide hosted virtualization capabilities. These virtualization platforms can be described as follows:

- Microsoft Hyper-V - is a Type-1 hypervisor integrated into Windows Server and certain Windows client editions. It is widely deployed in enterprise environments and forms the foundation of Microsoft Azure infrastructure.

[1] `https://learn.microsoft.com/pt-pt/windows-server/virtualization/hyper-v/`
[2] `https://www.vmware.com/`
[3] `https://linux-kvm.org/page/Main_Page`
[4] `https://www.proxmox.com/en/`
[5] `https://www.virtualbox.org/`
[6] `https://www.qemu.org/`

- VMware - offers both Type-1 and Type-2 solutions, including ESXi for enterprise deployments and Workstation for desktop virtualization. VMware platforms are widely adopted in commercial data centers.
- Oracle VirtualBox - is a Type-2 hypervisor commonly used in academic, testing and development environments. It provides cross-platform support and ease of deployment.
- KVM - is an open-source virtualization module integrated into the Linux kernel, enabling Linux to function as a Type-1 hypervisor.
- Proxmox VE - builds upon KVM and container technologies, offering a web-based management platform for enterprise virtualization environments.
- QEMU - provides machine emulation and hardware virtualization capabilities and is often combined with KVM for performance acceleration.

2.3 Resource Management in Virtual Environments

Resource management is a central responsibility of virtualization platforms. In a virtualized environment, multiple VMs share the same physical hardware, requiring the hypervisor to coordinate access to computational resources in a controlled and efficient manner. The objective of resource management is to maximize hardware utilization while preserving performance stability and logical separation between VMs [34].

The hypervisor regulates CPU access by scheduling virtual CPUs into physical cores. This scheduling process determines how computing time is distributed among VMs and must balance the output of the system [16]. Modern virtualization platforms rely on hardware-assisted mechanisms to allow guest OSs to execute most instructions directly on the CPU, while privileged operations are mediated by the hypervisor.

In addition to CPU allocation, the hypervisor manages memory, storage and input/output resources. Storage and input/output virtualization abstracts physical devices and presents controlled virtual interfaces to guest systems. Disk images, virtual network interfaces and shared buffers are handled in a manner that allows multiple VMs to coexist without direct access to one another's resources [44].

Resource management also includes dynamic allocation and reclamation. VMs can be created, resized, suspended, rebooted, or terminated during operation. Throughout these lifecycle events, the hypervisor must allocate and later reclaim physical resources so they can be reassigned to other VMs.

While resource sharing improves hardware efficiency and scalability, it introduces additional complexity in how physical resources are reused over time. In particular, memory resources require careful handling when they are reclaimed and reassigned.

2.4 Memory Virtualization

Computer systems rely on a hierarchical memory architecture composed of caches, main memory (Random Access Memory (RAM)) and secondary storage devices such as drives and hard disks. Caches provide extremely low-latency access to frequently used data but are limited in size. Secondary storage offers large capacity but with significantly higher latency. Positioned between these layers, RAM serves as the primary working memory of the system [17], storing instructions and data that are actively used by the CPU.

Unlike persistent storage devices, RAM is volatile, meaning that its content is expected to be lost when power is removed. OSs rely on this property to manage memory allocation and process isolation. When a process terminates, the memory it previously occupied is released and made available for reassignment. Proper memory management requires that previously used memory not unintentionally expose residual data when reused by other processes.

In addition to physical memory, modern OSs implement virtual memory mechanisms that extend available memory capacity through paging and swapping techniques. Virtual memory abstracts physical memory by mapping virtual addresses used by applications to physical memory pages and when necessary, inactive pages can be temporarily stored on disk. While virtual memory improves flexibility and resource utilization, it further increases the complexity of memory allocation and reclamation processes.

Ideally, when a VM is shutdown or rebooted, the memory previously assigned to it should be cleared before being reused. This prevents data belonging to one VM from

remaining accessible to the host or to other VMs.

Previous works explored the concept of using volatile memory as more than passive storage. In [13], the authors propose architectures where computation is performed directly within memory. The goal of this approach is to reduce the overhead caused by moving data between the processor and memory. The work shows that RAM plays a central role in system design and directly affects performance and security.

Memory virtualization is a technique used by OSs and hypervisors to give each process or VM the illusion of having its own private and continuous memory space. Instead of accessing physical RAM directly, software components operate on virtual memory addresses, which are translated to physical memory addresses by the system.

This technique is a responsibility of the hypervisor, as it controls the translation and isolation of guest memory accesses. Each VM maintains its own virtual address space, where memory references issued by applications are first translated by the guest OS into guest physical addresses. These guest physical addresses are then translated by the hypervisor into host physical memory addresses. This two-level address translation is commonly implemented using mechanisms like hardware-assisted approaches, including extended or nested page tables [25]. Through this layered translation process, the hypervisor enforces memory isolation by controlling which host physical pages are accessible to each VM.

Beyond address translation, the hypervisor is also responsible for memory allocation, reclamation and reuse. When a VM is created, the hypervisor reserves and assigns host physical pages to it. When the VM is stopped, rebooted, or its memory demand changes, those pages can be reclaimed and reassigned. To preserve isolation guarantees, the hypervisor is expected to ensure that previously allocated memory pages do not expose residual data to other VMs once they are released. An overview of hypervisor design principles and early virtualization architectures is provided by Barham et al. [5].

Several memory management techniques are used to support memory virtualization, including address translation, paging and dynamic memory allocation. An overview of these techniques and their implications in virtualized systems is presented by Mishra and Kulkarni [29]. Their survey highlights how memory virtualization mechanisms aim to balance isolation, performance and efficient memory utilization, while also introducing challenges related to memory reuse and management.

Memory virtualization improves flexibility and efficiency. Physical memory can be allocated dynamically based on the needs of each VM and memory can be reclaimed when a VM is shutdown. Techniques such as paging and address translation support this behavior. An overview of these concepts is presented in the *BrainKart* article on memory virtualization [9].

From a security perspective, virtualization mechanisms are expected to enforce strict memory isolation, such that memory released by a VM should not remain accessible or retain sensitive information. However, prior work has shown that physical memory pages can be reclaimed and reused without immediate sanitization [2, 41, 36].

2.5 Virtualized Environments on Cloud deployment

Cloud computing provides on-demand access to shared resources, such as computing power, storage and networking, over the Internet. This allows organizations to scale their operations flexibly while reducing the costs associated with maintaining physical infrastructure [8].

A key innovation enabling cloud computing is virtualization, which allows multiple VMs to run on a single physical server. This improves resource utilization but introduces unique security challenges due to the infrastructure's shared nature. In particular, the memory isolation between VMs becomes critical to ensure that sensitive data from one machine is not accessible to others.

Figure 2.3 provides an overview of cloud computing. Cloud services such as applications, storage and virtualized computing resources are hosted on remote infrastructure and accessed by users over the Internet.

Cloud providers are organizations that offer computing resources and services over the Internet, enabling users to access scalable infrastructure without the need to invest in physical hardware [19]. These providers typically operate large data centers that host shared physical resources, which are abstracted and delivered to customers through virtualization. Some of the leading cloud providers include [18]:

- Amazon Web Services (AWS)
- Microsoft Azure

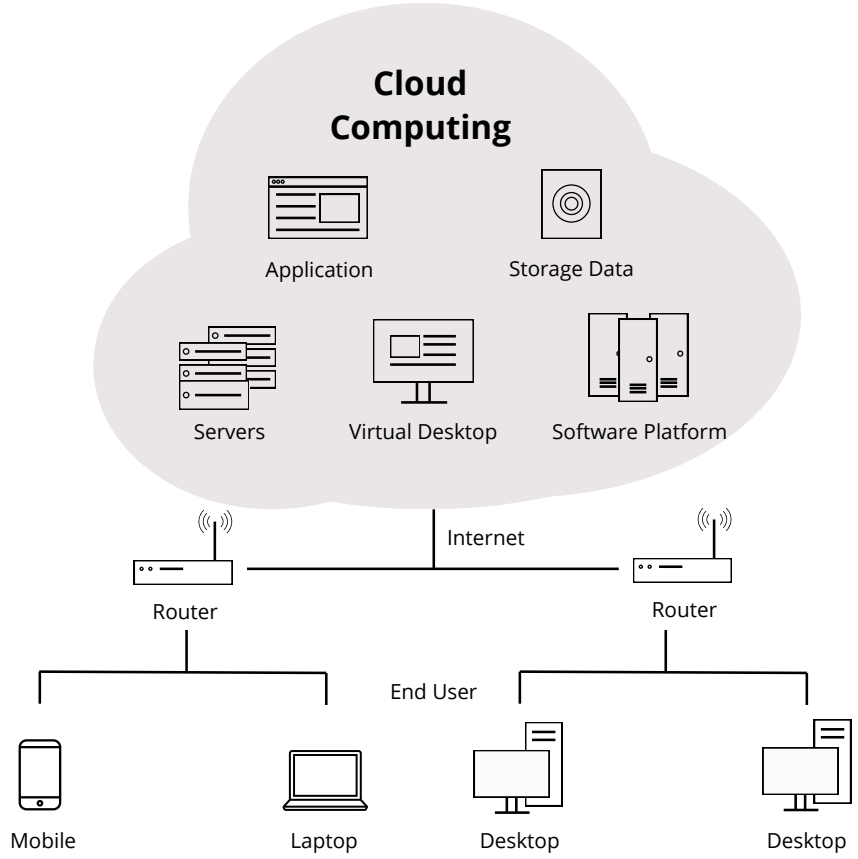


Figure 2.3: Overview of cloud computing

- Oracle Cloud
- IBM Cloud
- OVHcloud
- Google Cloud Platform

Cloud providers offer their services on a pay-as-you-go basis, allowing businesses and individuals to use resources such as compute power, storage and networking on demand. This flexibility has driven the adoption of cloud computing across various industries, from healthcare to finance and beyond. However, the shared nature of these resources introduces critical security concerns, particularly in multi-environments where multiple VMs operate on the same physical server.

One of the most significant challenges facing cloud providers is ensuring isolation between machines. This involves separating not only storage and networking, but also transient resources such as RAMs, which are often used simultaneously by multiple VMs.

Most cloud computing services fall into five broad categories [32].

- Software as a Service (SaaS) - Cloud service model in which applications are delivered over the Internet and accessed through a web interface or client application. The service provider manages the infrastructure, platform and application layers, while users interact only with the software itself.

- Platform as a Service (PaaS) - Provides a managed development and deployment environment that enables users to build, test and deploy applications without managing the infrastructure. The provider supplies runtime environments, development tools and middleware components hosted in the cloud.

- Infrastructure as a Service (IaaS) - Delivers virtualized computing resources, including VMs, storage and networking components, on a demand-driven basis. Users retain control over OSs and applications, while the provider manages the physical hardware and virtualization layer.

- Anything/Everything as a Service (XaaS) - Refers to a broad cloud computing paradigm in which a wide range of services beyond traditional infrastructure, platform and software offerings—are delivered over the Internet under a service-based model.

- Function as a Service (FaaS) - Is a serverless computing model in which users deploy discrete functions that are executed in response to events. The cloud provider automatically manages infrastructure provisioning, scaling and execution environments, allowing developers to focus solely on application logic.

Chapter 3

Related Work

This chapter reviews prior research related to memory persistence, isolation, and security in virtualized environments. The literature is organized into thematic categories, including memory remanence and persistence, isolation and cross-domain attacks, memory deduplication and page sharing, cloud security and virtualization threats, architectural protection mechanisms, and volatile memory analysis.

3.1 Memory Remanence and Persistence

Memory remanence refers to the persistence of data in main memory after it is expected to be cleared. Several studies have shown that data can remain accessible after deallocation, shutdown, or VM termination. In virtualized and cloud environments, this behavior becomes particularly relevant, as memory can be reassigned across different VMs or reused during lifecycle transitions. This section reviews prior work that examines how memory content persists in OSs, VMs, and cloud infrastructures.

The persistence of data in RAM has been empirically demonstrated by Halderman et al. [21], who showed that memory content can remain intact for a short period after power loss and can be recovered under certain conditions. Their work highlights that volatile memory does not lose the content instantaneously and that residual data can persist long enough to be observed or extracted.

Gutmann [20] discusses the persistence of data remnants in magnetic and solid-state storage media, demonstrating that deleted data can remain recoverable depending on how

storage blocks are overwritten or reused. Although the study focuses on persistent storage rather than volatile memory, it establishes a broader principle: data do not necessarily disappear immediately after logical deletion. This observation is conceptually relevant to volatile memory as well, where contents can persist in physical memory until explicitly overwritten or reallocated.

In [11], Chow et al. present one of the earliest systematic analyses of data lifetime and memory remanence in modern OSs. Their work investigates whether sensitive information remains recoverable in RAM after applications release memory or terminate execution. The authors design controlled experiments in which known data patterns are written into memory, followed by standard deallocation procedures provided by the OS. Using kernel-level inspection and memory analysis techniques, they demonstrate that freed memory pages are frequently returned to the system allocator without being overwritten. As a result, subsequent processes can access residual data if they reallocate the same physical pages. Their findings establish that memory remanence is not just a hardware detail but also a consequence of software-level memory management decisions.

Savchenko et al. [36] investigate data remanence in the main memory of VMs, focusing specifically on whether volatile data persists after VM reboot operations. Using KVM as the virtualization platform, the authors design controlled experiments in which identifiable data is placed into VM memory before triggering a reboot. After the reboot, they perform memory acquisition and forensic analysis to determine whether previously written data remains accessible. Their results show that a substantial portion of volatile memory contents can persist across VM reboots, indicating that memory is not systematically sanitized during the reboot process. The study emphasizes the implications of this behavior for digital forensic workflows, particularly in malware analysis scenarios where VMs are repeatedly reused. The authors argue that residual memory can contaminate subsequent analyses if no precautions are taken. However, the scope of their evaluation is limited to a single virtualization technology (KVM) and the analysis primarily concentrates on reboot scenarios. Furthermore, the work is framed from a forensic contamination perspective rather than from a comparative isolation or cross-platform memory management standpoint.

Albelooshi et al. [2] present one of the earliest experimental investigations demon-

strating data remanence in cloud-based VM environments. Their study evaluates whether memory allocated to a terminated VM instance in a public cloud infrastructure is properly sanitized before being reassigned to a new tenant. The authors design controlled experiments in which identifiable data patterns are written into VM memory prior to instance termination. After provisioning a new VM instance, potentially on the same physical host, they analyze the memory space to determine whether residual data from the previous tenant remains accessible. Their experimental results indicate that, under certain conditions, portions of memory previously used by one VM can be recovered in subsequently allocated instances. This provides empirical evidence that memory sanitization mechanisms in cloud environments cannot always guarantee immediate or complete clearing of physical memory before reuse.

Snyder and Jones [41] investigate the effectiveness of data remanence prevention mechanisms in AWS, focusing on whether memory and storage resources are properly sanitized when VM instances are terminated and reallocated. Their methodology consists of deploying controlled VM instances within the AWS infrastructure, writing identifiable data patterns to both volatile memory and persistent storage, and subsequently terminating the instances. New instances are then provisioned, and forensic techniques are employed to analyze whether residual data from prior allocations can be recovered. The study aims to empirically evaluate the practical effectiveness of AWS's sanitization policies rather than relying solely on provider documentation. The authors report that, although AWS implements mechanisms intended to mitigate data remanence, such as zeroing storage blocks before reassignment, complete sanitization cannot always be guaranteed under all tested conditions. In particular, remnants of previously stored data can persist depending on resource type, allocation timing, and reuse patterns.

3.2 Memory Isolation and Cross-Domain Attacks

Virtualization relies on strong isolation between VMs and the host system. However, some studies have shown that isolation guarantees can be weakened by architectural design choices, shared hardware resources, or implementation flaws. In particular, cross-domain and cross-VM attacks demonstrate that information can leak between isolated environ-

ments, even when direct memory access is prevented. This section reviews work examining how isolation boundaries in virtualized systems can be bypassed or indirectly influenced.

Pan et al. [31] examine hypervisor exploitation from the perspective of cross-domain attacks, analyzing how isolation guarantees between host and guest environments can be bypassed through subtle architectural and implementation weaknesses. The authors propose a systematic framework for identifying cross-domain attack surfaces in virtualization systems, focusing on memory handling, shared resources, and state transitions between privilege domains. Their methodology combines architectural analysis with proof-of-concept exploitation techniques to demonstrate how improper validation, shared memory mappings, or insufficient boundary enforcement can allow adversaries to access or influence memory across isolation layers. Rather than targeting a single vulnerability, the work highlights structural patterns in hypervisor design that can enable unintended information leakage or privilege escalation. The study demonstrates that virtualization isolation mechanisms can be impacted by design flaws or insufficiently constrained cross-domain interactions. The authors conclude that isolation at the architectural level must be complemented by careful implementation and verification to prevent leakage of memory contents or execution state across VM boundaries.

Ristenpart et al. [35] demonstrate cross VM information leakage in public cloud environments. Their work investigates whether an attacker can intentionally place a malicious VM on the same physical host as a target VM and subsequently exploit shared hardware resources to extract sensitive information. The authors develop techniques to identify co-residency in Amazon EC2 by analyzing network topology, internal Internet Protocol (IP) addressing patterns and side-channel indicators. Once co-location is achieved, they demonstrate several cross-VM side-channel attacks exploiting shared CPU caches and other microarchitectural resources to infer information about the victim's activity. Their experimental evaluation confirms that multi-tenant cloud infrastructures can allow information leakage through shared physical components, even when strict logical isolation is enforced at the virtualization layer. The study concludes that co-residency combined with shared-resource usage fundamentally challenges the assumption that VMs provide strong isolation in cloud platforms. While virtualization prevents direct memory access between tenants, indirect leakage through shared hardware channels remains possible.

Dolan-Gavitt et al. [12] study VM introspection techniques that allow a host system to observe and analyze the memory state of a guest VM. Their work demonstrates that VM memory can be accessed and interpreted externally, thereby challenging assumptions about the strict isolation of guests. This supports the idea that memory content created inside a VM can remain visible at the host level.

3.3 Memory Deduplication and Page Sharing

Memory deduplication is a common optimization technique in virtualized systems that reduces memory usage by merging identical physical pages across VMs. While this approach improves memory efficiency and enables higher consolidation ratios, it also affects how memory isolation is enforced. Because multiple VMs can share the same physical page, deduplication can introduce new security considerations. This section reviews prior work examining both the performance benefits and the security implications of page sharing mechanisms in virtualized environments.

Shaikh et al. [37] propose a hypervisor-level memory deduplication mechanism designed to identify and merge identical physical pages in order to improve memory efficiency. Their work acknowledges that redundant copies of identical pages can coexist in physical memory before deduplication. However, this redundancy is treated as an optimization opportunity rather than as a persistence or isolation concern.

Suzaki et al. [43] provide one of the earliest security analyses of memory deduplication in virtualized systems. Their work demonstrates that page sharing mechanisms can be exploited as a side channel to infer information about co-resident VMs. The authors show that by crafting memory pages and observing copy-on-write behavior or timing differences, an attacker VM can determine whether another VM holds identical memory content. This creates a covert channel that can leak sensitive information, such as the presence of specific applications or data patterns. The study combines experimental evaluation with proof-of-concept attacks, concluding that transparent page sharing, although designed for performance optimization, can unintentionally weaken isolation guarantees between guests. More recently, Huang et al. [24] revisit memory deduplication from a performance and scalability perspective, focusing on deduplication techniques. Their work proposes

enhancements that extend deduplication mechanisms to larger memory page sizes, aiming to improve efficiency in modern cloud workloads. Through experimental evaluation, they demonstrate that hugepage-aware strategies can significantly increase memory savings and improve system performance while reducing overhead associated with traditional small-page deduplication. The study primarily evaluates performance metrics such as memory utilization, throughput, and deduplication efficiency. Together, these works illustrate the trade-off in memory deduplication mechanisms: while they offer clear benefits in terms of resource optimization and scalability, they can introduce new security risks by altering isolation properties at the physical memory level.

In [6], Barker et al. conduct an empirical study of memory sharing behavior in virtualized environments, examining the effectiveness of content-based page sharing mechanisms in practice. Their methodology involves collecting memory traces from multiple VMs running on shared hosts and analyzing page-level redundancy to determine how much memory can be consolidated through deduplication. By comparing identical 4 Kilobyte (KB) pages across and within VMs, the authors quantify both intra-machine (self-sharing) and inter-machine sharing opportunities. The study also evaluates how workload characteristics, OS similarity, and hypervisor policies influence deduplication efficiency. The authors find that a substantial portion of memory sharing originates from redundancy within a single VM rather than across different VMs. Inter-machine sharing is shown to depend heavily on system homogeneity, particularly the use of identical OSs and application stacks.

3.4 Cloud Security and Virtualization Threats

Cloud computing environments rely heavily on virtualization to enable resource sharing and multi-tenancy. While virtualization improves scalability and efficiency, it also introduces new security challenges. In particular, shared infrastructure and co-resident VMs can increase the risk of data leakage and isolation failures. This section reviews prior work that discusses security threats in cloud-based virtualized systems.

Subashini and Kavitha [42] provide an early and comprehensive survey of security challenges in cloud computing service models, with particular emphasis on virtualization as a foundational enabling technology. Their work focuses on known risks related to

multi-tenancy, data confidentiality, resource sharing, and isolation failures in IaaS environments. The authors discuss virtualization-specific threats such as hypervisor compromise and inter VM data leakage, highlighting how shared physical resources can impact isolation guarantees if not properly managed. As a survey paper, their methodology consists of categorizing and analyzing previously reported vulnerabilities and architectural weaknesses rather than conducting empirical experiments.

3.5 Memory Encryption and Architectural Protection

Beyond software-based isolation, some works propose hardware-supported mechanisms to protect memory content from unauthorized access. Because data stored in RAM is typically maintained in plaintext during execution, isolation often depends on the correctness of the hypervisor and OS. Architectural enhancements and memory encryption techniques aim to strengthen these guarantees by preventing direct observation or manipulation of physical memory. This section reviews prior work addressing memory encryption and hardware-level isolation mechanisms in virtualized environments.

Unterguggenberger et al. [45] introduce a system that leverages Intel's Total Memory Encryption with Multi-Key support to provide scalable in-process isolation of sensitive data. The authors begin from the premise that conventional software isolation mechanisms, such as process separation and access control, do not protect against adversaries capable of reading raw physical memory, whether through privileged compromise, cold-boot techniques, or hardware attacks. Their methodology involves designing a runtime system that assigns distinct hardware encryption keys to protected memory regions within the same process address space. By using CPU supported memory encryption, the system ensures that data stored in Dynamic random access memory (DRAM) remains encrypted outside the processor package, with decryption occurring transparently during CPU access. The authors evaluate the system in terms of scalability, performance overhead and isolation guarantees, demonstrating that hardware-assisted memory encryption can provide strong confidentiality with modest runtime costs. Their findings emphasize that, in typical systems, application data resides in RAM in plaintext and relies on logical isolation mechanisms for protection. Without hardware-backed encryption, any entity capable of

accessing raw physical memory can potentially observe sensitive data.

Jin and Huh [25] propose an architectural enhancement aimed at strengthening memory isolation among VMs at the hardware level. Their work addresses limitations in conventional virtualization architectures, where memory isolation is largely enforced by the hypervisor through software-managed page tables and extended paging mechanisms. The authors argue that this design places significant trust in the hypervisor and can expose isolation guarantees to vulnerabilities arising from software bugs or misconfigurations. The proposed architecture ensures that memory accesses are validated against hardware-enforced ownership rules before translation, thereby preventing unauthorized cross VM memory access even if the hypervisor is compromised. The authors evaluate their design through architectural simulation, analyzing performance and isolation effectiveness. Their results suggest that hardware-supported isolation can significantly reduce the attack surface associated with hypervisor-managed memory translation while maintaining acceptable performance costs.

Henson and Taylor [23] present a comprehensive survey of memory encryption techniques aimed at protecting code and data stored in main memory. Their work systematically categorizes existing approaches into three main groups: hardware-centric designs that perform encryption and decryption within a trusted processor boundary, hybrid solutions that combine hardware support with OS level key management mechanisms and specialized security-oriented devices or coprocessors designed for particular application domains. The authors analyze the architectural principles, security guarantees and performance implications of each category, highlighting that most traditional system designs leave RAM content in plaintext during normal operation. As a result, memory encryption has historically not been integrated into mainstream OS architectures, primarily due to concerns regarding performance, hardware complexity, and limited availability of suitable CPU support. The survey emphasizes that, in the absence of memory encryption, sensitive data stored in RAM remains vulnerable to a wide range of attacks, including physical memory extraction, cold-boot attacks, direct memory access exploitation, and privileged software compromise.

3.6 Memory Forensics and Volatile Memory Analysis

Volatile memory contains valuable information about the current state of a system. Unlike persistent storage, RAM stores active processes, temporary buffers, decrypted content and other runtime content that can not be written to disk. For this reason, memory forensics has become an important field for incident response and security analysis. This section reviews prior work examining techniques for acquiring, analyzing, and interpreting volatile memory in both traditional and virtualized environments.

Sianipar et al. [38] examine the exposure of sensitive data stored in RAM in scenarios involving live memory acquisition and speculative execution–based attacks. The authors begin from the observation that application data, including credentials and cryptographic material, is typically maintained in plaintext in main memory during execution. Their study analyzes how such data can be recovered through memory dumping techniques and hardware-induced leakage mechanisms, emphasizing the practical recoverability of in-memory artifacts. In addition to demonstrating exposure risks, the authors propose mitigation strategies aimed at reducing the residency time of sensitive data in memory. These techniques involve clearing or relocating sensitive buffers once they are no longer required, thereby limiting the amount of data available in memory at any given time. Experimental evaluation shows that shortening the lifetime of sensitive data in RAM significantly reduces its recoverability. The study reinforces the broader assumption that volatile memory often contains accessible sensitive information and that its confidentiality depends largely on proper memory management practices.

Hausknecht, Foit, and Burić in [22] analyse the role of volatile memory in digital forensic investigations, emphasising the value of RAM content for incident response and post-mortem analysis. Their work highlights that RAM frequently contains critical content such as active processes, decrypted files, cryptographic keys, network session information, authentication credentials, and temporary buffers that can never be written to persistent storage. The authors discuss methodologies for live memory acquisition and forensic examination, outlining how memory captures can reveal system state at a specific point in time. Through practical case-oriented analysis, they demonstrate that volatile memory often provides insights unavailable from disk-based forensic analysis alone, particularly in

scenarios involving encryption, malware, or in-memory execution techniques.

Case et al. [10] investigate techniques for live memory forensics, focusing on the dynamic reconstruction of kernel data structures from RAM images. Their work addresses a fundamental challenge in memory forensics: accurately interpreting volatile memory without relying exclusively on OS symbols or predefined structure layouts. The authors propose methods to dynamically identify and reconstruct kernel objects directly from raw memory dumps, enabling the extraction of process lists, loaded modules, network connections, and other critical system information. Their methodology involves analysing memory snapshots from live systems and developing techniques to infer kernel data-structure layouts, even in the presence of address randomisation or partial corruption. The study demonstrates that live memory analysis can reveal detailed information about system state that is unavailable through disk-based forensic examination alone, particularly in cases involving rootkits, in-memory malware, or tampered kernel structures.

3.7 Comparative Analysis

This section compares the studies reviewed and relates them to the research questions of the current study. Instead of repeating each work individually, the comparison summarizes prior research using common criteria, such as whether the study analyzes host or guest memory, considers lifecycle transitions (e.g., shutdown or reboot), evaluates cloud environments and addresses the defined research questions (RQ1–RQ3). The purpose is to identify similarities and differences across the literature and to clarify how the current study differs in scope and approach.

Table 3.1 displays the main characteristics of the related work discussed and contrasts them with the current study across evaluation dimensions. The comparison highlights whether each work examines host-level memory, VM memory, lifecycle transitions such as shutdown or reboot, and cloud environments. It also indicates whether the study performs page-level analysis or investigates memory duplication. As shown in the table, prior research has mainly focused on specific aspects of memory security, such as OS level remanence, forensic analysis of volatile memory, cross-VM side channels, deduplication-related risks, or isolation mechanisms. While some studies evaluate residual data in cloud environ-

ments or across VM reallocation scenarios, few systematically analyze memory behavior across explicit VM lifecycle transitions. Moreover, page-level analysis and memory duplication characteristics are generally not addressed in combination. In contrast, the current study integrates host and guest memory evaluation with lifecycle-based experimentation and analysis of page-level behavior and memory duplication.

Table 3.1: Comparative analysis of related work.

Ref.	Focus	Year	Host	VM	Lifecycle Env.	Cloud Analysis	Page-Level	Memory Duplication
[11]	OS-level memory remanence	2005	✓					
[35]	Cross-VM side channels	2009		✓		✓		
[10]	Kernel memory forensics	2010	✓					
[43]	Deduplication security risks	2011		✓		✓		
[25]	Hardware-enforced isolation	2011		✓		✓		
[42]	Cloud security survey	2011				✓		
[6]	Memory sharing efficiency	2012		✓		✓	✓	
[12]	Host-based VM memory introspection	2011	✓	✓			✓	
[23]	Memory encryption survey	2014						
[22]	Forensic value of RAM	2015	✓					
[2]	Cross-tenant remanence	2015	✓	✓	✓	✓		
[41]	Cloud sanitization effectiveness	2019	✓	✓	✓	✓		
[38]	In-memory exposure mitigation	2018		✓		✓		
[24]	Hugepage deduplication	2023		✓		✓	✓	
[36]	VM memory remanence (reboot)	2024	✓	✓	✓			
[31]	Hypervisor cross-domain attacks	2025		✓		✓		
Current Study	Lifecycle memory persistence analysis	2026	✓	✓	✓		✓	✓

Table 3.2 maps the reviewed literature to the research questions defined in the current study. The table shows that the majority of prior work concentrates on security and privacy implications of memory exposure in virtualized or cloud environments (RQ3), particularly through side-channel attacks, deduplication-based leakage, forensic recovery, or isolation mechanisms. However, only a limited number of studies directly examine how memory content generated inside a VM persists in host memory (RQ1) and even fewer explicitly analyze when and how memory is cleared during virtualization processes such as shutdown or reboot (RQ2). Works such as [2], [41], and [36] address persistence and

clearing behavior in the context of cross-tenant reuse or reboot scenarios. In contrast, the current study integrates all three research dimensions by systematically evaluating host-visible persistence and lifecycle-based clearing behavior.

Study	RQ1	RQ2	RQ3
[11]			
[35]			✓
[10]			✓
[43]			✓
[25]			✓
[42]			✓
[6]			
[12]	✓		✓
[23]			✓
[22]			✓
[2]	✓	✓	✓
[41]	✓	✓	✓
[38]			✓
[24]			
[36]	✓	✓	✓
[31]			✓
Current Study	✓	✓	✓

Table 3.2: Mapping of related work to the current study research questions.

3.8 Summary

The reviewed literature indicates that virtualization mechanisms are designed to provide strong logical isolation between VMs. However, several studies demonstrate that memory-related factors such as residual data, page sharing mechanisms, and cross-tenant resource reuse can weaken these guarantees in practice. Other works also highlight that memory content is typically stored in plaintext during execution, meaning that confidentiality depends on isolation mechanisms rather than memory protection. As a result, unintended exposure of memory content can occur if isolation boundaries are not strictly enforced or if memory is not properly cleared.

With these observations, the current study investigates how memory content generated within a VM persists in host memory across different lifecycle states and different hypervisors. Specifically, it evaluates widely used hypervisors and analyzes memory behavior while the VM is running, after shutdown, and following reboot. In addition to measuring persistence, the study examines how VM data is relocated and duplicated at the host memory page level, offering a detailed view of how volatile memory evolves across VM lifecycle events.

Chapter 4

Memory Isolation and Threat Model

This chapter defines the assumptions and security context under which this work is conducted. It describes how memory isolation is expected to operate in virtualized systems, explains the concept of volatile memory and data persistence and formally defines the threat model and scope of the study.

4.1 Memory Isolation in Virtualized Systems

Memory isolation is a fundamental property of virtualization. In a virtualized system, each VM is expected to operate within its own isolated memory space, even though physical memory is shared among multiple VMs and the host system. This isolation is enforced by the hypervisor, which controls how physical memory pages are allocated, mapped, and reclaimed.

From the perspective of a VM, memory appears private and exclusive. When a VM is shutdown or rebooted, the memory previously assigned to it is expected to become inaccessible and should not expose any of its content to other VMs or to the host. This assumption is central to the security guarantees of virtualization and is particularly important in shared environments such as cloud infrastructures.

However, memory isolation is implemented through software and hardware mechanisms that prioritize performance and efficiency. As a result, isolation is not equivalent to

immediate memory erasure, and the behavior of physical memory after a VM's lifecycle events is not directly observable from within the VM itself.

4.2 Volatile Memory and Data Persistence

Although RAM is volatile, the content is not instantly erased when a VM stops using a memory region. Physical memory pages can retain their data until they are overwritten by new allocations.

In virtualized environments, residual memory can occur when memory pages used by a VM are returned to the hypervisor but are not immediately cleared. These pages can later be reassigned to another VM or remain accessible at the host level. While this behavior is often a result of performance optimizations, it raises concerns about data persistence and potential information exposure.

Residual memory does not necessarily indicate a vulnerability or a violation of access controls. However, it challenges the assumption that sensitive data is fully removed from memory once a VM terminates.

4.3 Threat Model

This work considers a well-defined threat model. The goal of the attacker is to observe residual memory content originating from a VM after certain lifecycle events, such as shutdown or reboot.

The attacker is assumed to have the following capabilities:

- Access to memory dumps collected from the host system.
- The ability to analyze memory content.

The attacker is assumed not to have the following capabilities:

- No ability to compromise or modify the hypervisor.
- No ability to bypass VM capabilities.
- No control over physical memory allocation decisions made by the hypervisor.

- No ability to interfere with or influence other VMs.

This threat model focuses on passive observation rather than active exploitation. The attacker does not inject malicious code, manipulate system behavior, or bypass access controls. Instead, the analysis is limited to examining whether memory content persists beyond its expected lifetime and whether such content can be detected under controlled conditions.

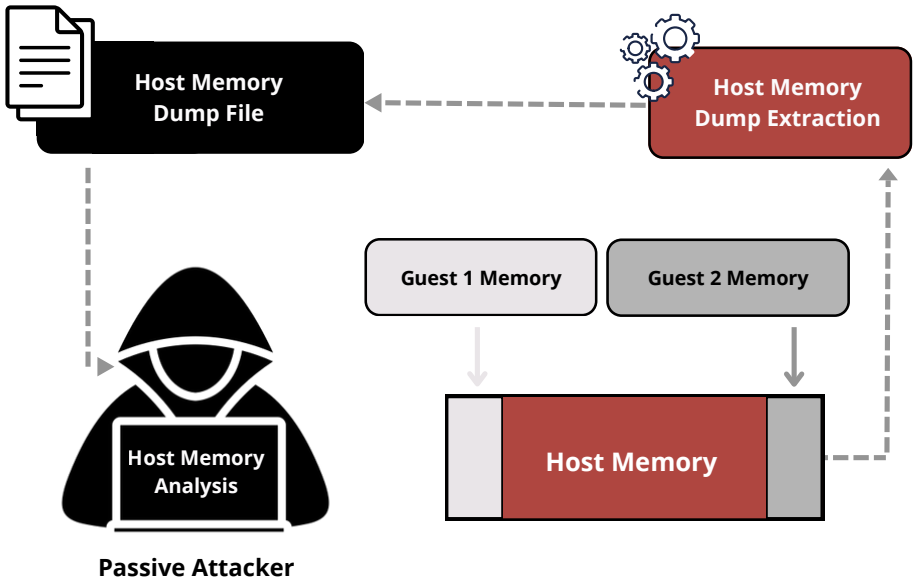


Figure 4.1: Threat Model Illustration

Figure 4.1 illustrates the threat model considered in this work. Two VMs execute on the same host system and share the physical memory managed by the host. The attacker is a passive observer who analyzes memory dumps obtained from the host system. The attacker does not interact with the VMs during execution and does not influence memory allocation or system behavior.

4.4 Scope and Limitations

The scope of this study is limited to the analysis of memory persistence and isolation in virtualized environments. The experiments are designed to observe how memory content generated inside a VM is handled across different lifecycle states and whether traces of the content remain observable.

This work does not claim to demonstrate data leakage in real-world cloud environments, nor does it present an active attack against cloud providers or hypervisors. Additionally, the results do not imply that memory persistence can be exploited in all scenarios or that sensitive data is directly exposed to unprivileged attackers.

Instead, the contribution of this work highlights the practical limitations of VM-level analysis and shows that host-level observation is necessary to accurately assess memory clearing behavior.

Chapter 5

Experimental Methodology

This chapter describes the experimental methodology adopted to evaluate memory persistence and isolation in virtualized environments. It describes the decisions, environment configuration, data generation process, memory acquisition procedures, and analysis techniques used throughout the study.

The chapter begins with preliminary work that motivated the final experimental design. Then, it details the host and VM configuration, followed by the development of the tag injection program used to populate memory with identifiable data. The memory acquisition process is described later, and finally, the analysis procedures applied to the collected memory dumps are presented.

5.1 Preliminary Work

The methodology used in this work is developed through a sequence of defined experimental approaches. Each approach focuses on a specific question and helps to identify limitations that guide the next step of the study. This process makes it possible to refine the methodology and focus on the most suitable perspective for evaluating memory persistence and memory isolation.

5.1.1 Exploratory Analysis in Cloud Environments

The initial approach focuses on cloud infrastructures and aims to evaluate whether cloud providers ensure complete memory isolation between VMs. Memory dumps are

collected from cloud-based VMs and compared across executions. The objective is to identify differences between dumps and to detect the presence of human-readable content, such as Hypertext Markup Language (HTML) fragments. Although differences between memory dumps are observed, the extracted data cannot be conclusively attributed to other VMs, as it can also originate from the OS.

Figure 5.1 illustrates the size difference between two memory dumps collected from the same cloud-based VM at different moments of execution. Both dumps are converted into human-readable text using the same extraction process. The second dump, obtained immediately before a hard reboot, is noticeably smaller than the first one. This difference indicates that the memory content captured at different points in time is not identical. However, the size difference alone does not provide information about the origin or sensitivity of the data and is therefore used only as an observation.

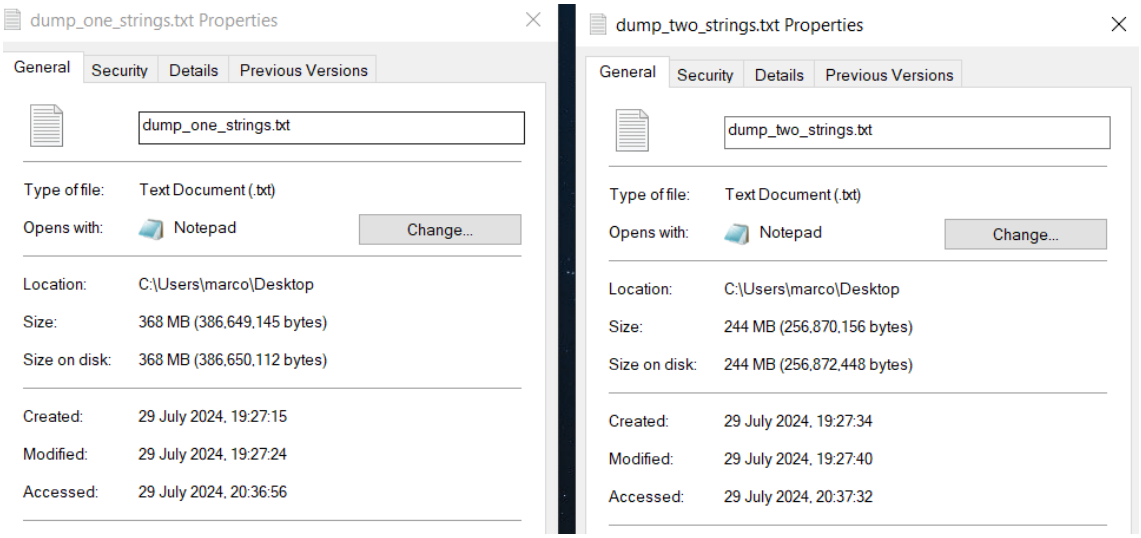


Figure 5.1: Cloud Dump Strings File Sizes

Listing 5.1 shows an example of fragmented HTML content extracted from a cloud memory dump after converting it to human-readable text. The content is incomplete and partially corrupted, indicating that it does not represent a valid or fully reconstructable document. While such fragments suggest the presence of human-readable data in memory, their origin cannot be determined, as they can belong to the OS, running services, or cached application data within the VM.

Listing 5.1: Fragmented HTML content extracted from a cloud memory dump

```
<a␣href="close␣to␣theexamples␣of␣is␣about␣the(see␣below)." id="searchprofessionalis
    ␣availablethe␣official␣</script>
<a␣href="?fam={{$fam}}&b={{$nb}}">[minute]</a>
<a␣href="?fam={{$fam}}&b={{add $nb 1}}">[hour]</a>
<a␣href="?fam={{$fam}}&b={{add $nb 2}}">[total]</a>
<a␣href="?fam={{$.Family}}&b={{$.Bucket}}&exp=1">[Normal/Expanded]</a>
<a␣href="?fam={{$.Family}}&b={{$.Bucket}}&exp=1&rtraced=1">[Traced/Expanded]</a>
<a␣href="?fam={{$.Family}}&b={{$.Bucket}}">[Normal/Summary]</a>
<a␣href="?fam={{$.Family}}&b={{$.Bucket}}&rtraced=1">[Traced/Summary]</a>
<a␣href="goroutine?debug=2">full␣goroutine␣stack␣dump</a>
<a␣href="https://z
<a href="http://www</a><a␣href="http://w
<a href="http://www.css" rel="stylesheet" </div></div><div classlanguage="
    javascript">aria-hidden="true">
<a href="index.phpwas␣established␣inmin.js"></script>
<a href="mailto
Ahrens␣&␣Birner␣Company␣GmbH
```

5.1.2 Memory Persistence Across Cloud Virtual Machine Hard Reboots

The second approach investigates whether memory content persists across hard reboots of cloud-based VMs. A custom program is developed and executed inside the VM to inject (close to 500) identifiable tags into memory. Memory dumps are collected before and after hard reboot operations initiated through the cloud provider. No injected tags are found after reboot, indicating that memory content is not observable after a hard reboot at the VM level.

```
marcopm001@masters:~/ strings dump_before_tags.raw | grep -c "MarcoMoreira"
0
marcopm001@masters:~/ strings dump_with_tags.raw | grep -c "MarcoMoreira"
521
marcopm001@masters:~/ strings dump_after_tags.raw | grep -c "MarcoMoreira"
0
marcopm001@masters:~/
```

Figure 5.2: First Tag Injection on Google Cloud

5.1.3 High Memory Usage in Cloud Environments

To determine whether memory pressure affects memory persistence, the tag injection program is refactored to occupy approximately 90% of the available memory. The same reboot and memory acquisition procedure is applied. The results are consistent with the previous approach, as no tags are detected after reboot. This suggests that increased memory utilization does not change memory visibility from within the VM.

```
marcopm001@masters:~/strings dump_with_overload.raw | grep -c "CyberMastersDegree"
190876446

marcopm001@masters:~/strings dump_after_overload.raw | grep -c "CyberMastersDegree"
0

marcopm001@masters:~/
```

Figure 5.3: Overload Tags Count on Google Cloud

5.1.4 Geographic Distribution of Cloud Resources

The fourth approach evaluates whether geographic location influences memory management mechanisms in cloud infrastructures. The experiments are repeated using VMs deployed in different regions of the same cloud provider. The results remain consistent across regions, indicating that memory handling behavior is independent of geographic location.

Status	Nome	Zona	Recomendações	Em uso por	IP interno	IP externo	Conectar
	masters	us-central1-a			10.128.0.4 (nic0)		SSH
	masters-v2	europe-north1-b			10.166.0.2 (nic0)		SSH
	masters-v3	me-central1-c			10.212.0.2 (nic0)		SSH
	masters-v4	asia-northeast1-b			10.146.0.2 (nic0)		SSH
	masters-v5	africa-south1-a			10.218.0.2 (nic0)		SSH

Figure 5.4: Google Cloud Virtual Machines

5.1.5 Local Virtualization Environment

To remove the abstraction introduced by cloud providers, the experiments are replicated in a controlled local virtualization environment using VirtualBox, Hyper-V, and VMware. The same tag injection and reboot methodology is applied. As in the cloud-

based experiments, no tags are found after a hard reboot, reinforcing the limitation of VM–level analysis.

5.1.6 Host-Level Memory Analysis

The limitations identified in all previous approaches motivate the final and most relevant stage of this study. Since the hypervisor controls physical memory allocation, observing memory exclusively from within the VM does not allow distinguishing between memory erasure and memory remapping. Therefore, the final approach analyzes memory from the host system's perspective to determine whether residual VM data remains in physical memory after different VM lifecycle states.

This final approach employs a structured, step-by-step study to evaluate how memory content generated inside a VM is handled and cleared by the host system across different VM lifecycle states. The study is organized into the following stages:

1. Host and VM Setup
2. Tag Injection Program
3. Memory acquisition
4. Analysis

5.2 Stage 1: Host and Virtual Machine Setup

In Stage 1, the host and VM environments are selected and deployed. The host system is a HP Pavilion laptop running Windows 11, equipped with 16 GB of DDR4 RAM in SODIMM form factor. The system features an AMD Ryzen 5 5600H processor with 6 physical cores and 12 logical processors, operating at a base frequency of 3.30 GHz. The host also includes an NVIDIA GeForce GTX 1650 GPU with 8 GB of dedicated video memory. Virtualization support is enabled to allow for the execution of the tested hypervisors. During the experiments, the host machine is dedicated to the test environment to minimize interference from other applications.

Regarding the selection of virtualization platforms, as highlighted by Singh and Chatterjee [40], commonly adopted VM monitors include VMware solutions, Oracle VirtualBox,

and Microsoft Hyper-V, among others. Thus, the following three virtualization platforms are selected for testing: VirtualBox, VMware, and Hyper-V. These platforms are among the most well-known and widely recognized virtualization solutions used in practice.

Table 5.1 describes key characteristics of the selected virtualization platforms. The comparison illustrates differences in architectural design, deployment context, and vendor ecosystems.

Table 5.1: Comparison of the virtualization platforms.

Feature	**VirtualBox**	**VMware**	**Hyper-V**
Vendor	Oracle	VMware, Inc.	Microsoft
Hypervisor Type	Type-2 (hosted)	Type-2 (Workstation)	Type-1 (bare-metal)
Typical Deployment	Desktop virtualization	Enterprise servers, cloud infrastructure, desktop virtualization	Enterprise environments, Windows Server, cloud platforms
Host OS Support	Windows, Linux, macOS	Windows, Linux	Windows
Open/Closed Source	Open-source core with proprietary extensions	Proprietary	Proprietary
Integration with Cloud Platforms	Limited	VMware Cloud ecosystem	Microsoft Azure integration

Each of these virtualization platforms is installed, and a VM is configured with 2048 MB of base memory, 128 MB of video memory, 4 CPUs, and 40 GB of storage. In the VM, the Ubuntu Server 22.04 OS is installed.

5.3 Stage 2: Tag Injection Program

In Stage 2, a program is developed to allocate a large memory region within the VM and populate it with unique, identifiable tags that will be later searched for. The program is developed using the C programming language, as it provides direct control over memory allocation and management. This allows precise manipulation of memory regions and minimizes abstraction layers that could interfere with the placement and persistence of

data in memory. Using C ensures that the injected data is predictably stored in memory. The number of tags, their format and their size are presented in Table 5.2.

Table 5.2: Configuration Variables Used for Tag Injection

Variables	**Value**
NUM_TAGS	25,000,000
TAG_TEMPLATE	XXXX-RX-%08d-DD-MM-YYYY-HH:MM:SS
TAG_SIZE	37 bytes

For each experiment, a total of 25 million tags are inserted into memory. The number of tags is chosen to densely populate a large region of the VM's memory while avoiding memory pressure. Each tag encodes four components: a four-character code representing the virtualization software used in the test, the test round identifier, a unique serial number ranging from 00000000 to 24999999, and a timestamp. An example of a generated tag is

- `HYPE-R1-00001234-26-07-2025-00:00:00`

In this example:

- `HYPE` - identifies the virtualization platform (Hyper-V). Possible values include `HYPE` (Hyper-V), `VMWA` (VMware) and `VBOX` (VirtualBox).
- `R1` - denotes the experimental round. Possible values include `R1`, `R2`, and `R3`, corresponding to different execution rounds of the experiment.
- `00001234` - is the unique serial number assigned to the tag
- `26-07-2025-00:00:00` - represents the date and time at which the tag is generated.

This structured format ensures that each tag is uniquely identifiable and allows the origin and context of each memory entry to be clearly determined during memory dump analysis. Each tag consists of 36 printable characters, which define the searchable identifier used during memory analysis. An additional byte is required to store the null terminator (`\0`) used by the C programming language to mark the end of the string in memory, resulting in a total allocation size of 37 bytes per tag.

After calculating the required memory, the program dynamically allocates the space and populates it with tags in a sequential loop. The code snippet shown in Listing 5.2 highlights the two key operations performed by the program. First, a large contiguous

block of memory is allocated to store all the tags. Then, each tag is generated using *snprintf()* and inserted into the allocated memory buffer using *memcpy()*, ensuring that the tags are sequentially written into memory. The full code is presented in Appendix A.

Listing 5.2: Tag Injection Loop

```
char *memory = malloc(total_size);

for (int i = 0; i < NUM_TAGS; ++i) {
    char tag[TAG_SIZE];
    snprintf(tag, sizeof(tag), TAG_TEMPLATE, i);
    memcpy(memory + i * TAG_SIZE, tag, TAG_SIZE);
}
```

5.4 Stage 3: Memory Acquisition

In Stage 3, memory dumps are obtained using WinPmem[1] on the host machine. In [1], the authors sustain that WinPmem captured Windows volatile memory accurately. Thus, WinPmem is selected as a reliable and commonly used open-source tool for physical memory acquisition on Windows systems.

Since the objective is to observe the persistence of the tags in memory on the host machine at different instants, the memory dumps are to be conducted multiple times and following a structured sequence of steps, as indicated in Figure 5.5.

The sequence of steps comprises eleven steps as follows:

Step 1 - The host machine is powered on

Step 2 - Initial memory dump of the host is acquired to establish a clean baseline.

Step 3 - The VM is then started.

Step 4 - Execution of the tag injection program inside the VM.

Step 5 - To evaluate memory visibility on the host while the VM is active, a host memory dump is acquired while the VM continues running with the populated tags.

Step 6 - The VM is then powered off.

[1] https://github.com/Velocidex/WinPmem

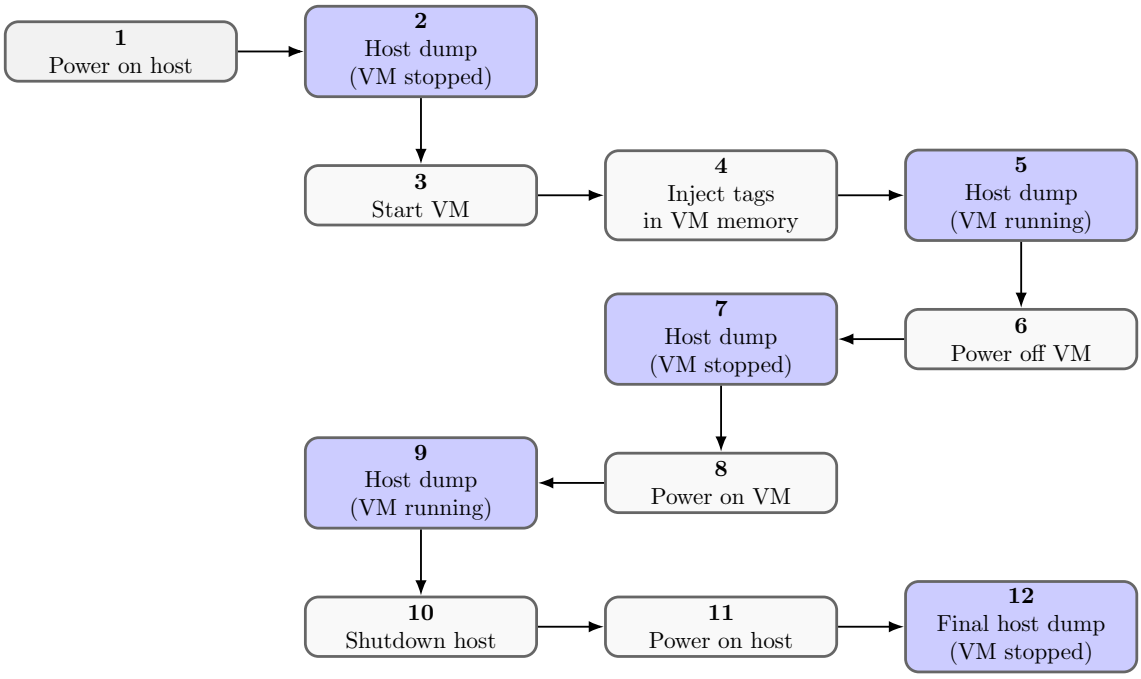


Figure 5.5: Actions and Memory dumps sequence.

Step 7 - Host memory dump is taken to analyze memory persistence after shutdown.

Step 8 - The VM is subsequently powered on again.

Step 9 - An additional host memory dump is then acquired to examine whether the tags persist on the host following the VM reboot.

Step 10 - The host machine is shutdown.

Step 11 - The host machine is powered back on

Step 12 - Host memory dump is performed to complete the experiment. The objective of this final step is to verify whether the host memory is cleared after a full system reboot.

This sequence is executed three times per virtualization platform (Hyper-V, VMware, and VirtualBox), totalling nine test executions. The changes applied between experiments are only due to the use of new tags, changing specifically the test round identifier, the virtualization platform identifier, and the timestamp. The size of the tags remained constant. Furthermore, three different tests (one per platform) are executed to analyse memory allocation and memory mapping.

5.5 Stage 4: Analysis

Stage 4 consisted of the analysis of the memory dumps, searching for the presence of the injected tags, and counting the number of matching tags. This approach enables the quick verification of whether the tags are present in a given dump and how many occurrences are detected. The process is repeated for every dump obtained during the experiments. For dumps where the count is greater than zero, the matching tags are extracted and stored. This ensured that all relevant results are preserved and can be systematically compared across different experimental conditions.

For further analysis, the extracted tag data is processed to enable a detailed comparison across the VM lifecycle states. Each detected tag is associated with its byte offset within the host memory dump and mapped to a host physical memory page based on the system page size. This page-level representation allows memory locations to be compared consistently across different dumps. As programs allocate and release memory, pages can be reassigned, relocated, or overwritten based on system activity [39].

To enable systematic analysis, the relevant information extracted from the memory dumps is converted into a structured Comma-Separated Values (CSV) format.

The analysis script operates directly on raw memory dumps and processes the data in fixed-size chunks to efficiently handle large files. Both ASCII and UTF-16 encoded tags are supported to account for differences in how data can be stored in memory by the OS and applications [26]. This ensures that injected tags can be reliably detected even when they appear in memory using different character encodings. The extracted results are stored in CSV format and used as input for all subsequent analyses, including memory footprint evaluation and tag relocation tracking. The complete implementation of the analysis script is shown in Appendix B.

```
1 Procedure ExtractTags(memory_dump):
2
3     open memory_dump
4     initialize overlap_buffer
5
6     while not end_of_file:
7         chunk = read_next_chunk()
8         data = overlap_buffer + chunk
9
```

```
        search data for ASCII tags
        search data for UTF-16 tags

        for each detected tag:
            determine memory_page
            write (tag_id, page) to CSV

        update overlap_buffer

    close files
```

Listing 5.3: Simplified tag extraction procedure

Listing 5.3 presents a simplified representation of the tag extraction procedure applied to the memory dumps. The process reads the dump file in fixed-size segments to ensure scalability when handling large memory images. For each segment, both ASCII and UTF-16 encoded patterns are searched to account for variations in how data can be represented in memory. Whenever a tag is detected, the corresponding memory page is determined based on the byte offset, and the extracted information is recorded in a structured CSV file.

Python[2] is chosen for this stage of the methodology due to its strong support for binary file handling, regular expression processing, and structured data output.

Another custom Python script automatically scans each memory dump CSV, identifies unique occurrences of the injected tags and records one row per detected tag. Each row contains the tag identifier and the corresponding host physical memory page in which the tag is found. The script is provided in Appendix C.

ChatGPT[3] is used as a supporting tool during the development of the Python scripts employed in this work. The tool is used to assist with code structuring, syntax clarification, and implementation suggestions. All generated code is manually reviewed, tested, and validated.

Listing 5.4 presents an example of a resulting CSV output. Each row corresponds to a detected tag instance based on the unique tag identifier, and the page field indicates the host physical memory page in which the tag is present.

[2] https://www.python.org/
[3] https://openai.com/index/chatgpt/

```
tag_id,page
00001234,27651
00001235,27651
00004567,1043921
```

Listing 5.4: Example CSV output

The script shown in Listing 5.5 processes the set of unique tags extracted from the memory dumps and calculates the number of tags associated with each physical memory page. For each VM state, the script aggregates tag occurrences by page, allowing the identification of pages that contain one or more tags.

```
import pandas as pd
from pathlib import Path

UNI = "../output/unique/"
OUT = "../output/stats/"

Path(OUT).mkdir(exist_ok=True)

for state in ["running", "shutdown", "reboot"]:
    df = pd.read_csv(UNI + f"{state}_unique.csv")
    page_counts = df["page"].value_counts().reset_index()
    page_counts.columns = ["page", "tag_count"]
    page_counts.to_csv(OUT + f"{state}_page_counts.csv", index=False)
```

Listing 5.5: Page count script based on unique tags

Chapter 6

Results

This chapter presents the empirical results obtained from the analysis of host memory dumps across the different virtualization platforms and VM lifecycle states. The findings are organized according to the key aspects evaluated in this study.

Section 6.1 examines memory persistence by quantifying the proportion of injected tags that remain detectable across running, shutdown, and reboot states. Section 6.2 analyzes memory allocation behavior, focusing on how VM-generated data is distributed across host physical memory. Section 6.3 investigates memory mapping patterns, exploring page-level relocation and movement across execution stages. Finally, Section 6.4 evaluates memory duplication, identifying instances in which identical guest-generated data appears across multiple host physical pages.

6.1 Memory Persistence

Memory persistence refers to the fact that data stored in memory is not necessarily removed immediately or completely once it is no longer in use. In virtualized systems, memory persistence is influenced by the OS and the hypervisor's memory management behavior. Memory regions released by a process or VM are typically marked as available rather than actively cleared, meaning that previously stored data can remain present in physical memory until it is overwritten.

With this, the first objective of the analysis focuses on determining the percentage of tags that still remain in memory after the tag injection. Although the program injects

25 million tags into memory, not all of them remain in memory due to normal system activity, memory management behavior, and eventually other concurrent processes. The analysis focuses on counting the tag present on the memory dumps created in the steps of the procedure. The steps referenced in the following tables correspond to the steps shown in Figure 5.5.

Table 6.1: Unique tag persistence on host per step for Hyper-V.

Round	**Step**					
	VM On (Step 5)		**VM Off (Step 7)**		**VM Reboot (Step 9)**	
	Tags	**%**	**Tags**	**%**	**Tags**	**%**
R1	24,994,631	99.9	2,894,631	11.6	73,371	0.3
R2	24,982,423	99.9	6,609,423	26.4	664,028	2.7
R3	24,984,936	99.9	10,001,512	40.0	949,421	3.8
Avg	24,987,330	99.9	6,501,855	26.0	562,273	2.3
Std Dev	5,283	<0.1	2,897,000	11.6	362,000	1.5

Table 6.1 shows how many unique VM tags remain in host memory at different stages of the VM when using Hyper-V. While the VM is running (Step 5), almost all injected tags are still present in memory, with an average of a 99.9% of the original data. After the VM is shutdown (Step 7), the number of remaining tags decreases, but a significant amount of data persists on the host. For instance, in Round 3, more than 10 million tags (equivalent to 40%) are detected after shutdown. After the VM is rebooted (Step 9), the number of remaining tags drops even more. Still focusing on Round 3, 949,421 tags (3.8%) are still present in this step. On average, 26.0% of tags remain after shutdown and 2.3% remain after reboot, showing that VM data is gradually cleared but not immediately removed from host memory.

Table 6.2: Unique tag persistence on host per step for VMware.

Round	**Step**					
	VM On (Step 5)		**VM Off (Step 7)**		**VM Reboot (Step 9)**	
	Tags	**%**	**Tags**	**%**	**Tags**	**%**
R1	24,904,961	99.6	7,369,473	29.4	834,629	3.3
R2	24,986,000	99.9	7,996,211	31.9	897,111	3.5
R3	24,986,198	99.9	7,300,812	29.2	914,306	3.6
Avg	24,959,053	99.8	7,555,499	30.2	882,015	3.5
Std Dev	38,200	0.2	316,000	1.3	33,000	0.1

Table 6.2 also shows the persistence of unique VM tags in host memory, but this time

using VMware. When the VM is running (Step 5), almost all tags remain present, similar to what happens in Hyper-V. After the VM is shutdown (Step 7), almost a third of the tags are still found in the host memory. For example, in Round 3, 7,300,812 (%29.2) remain after shutdown. After the reboot (Step 9), almost a million tags can still be visible in memory. On average, 30.2% of tags persist after shutdown and 3.5% remain after reboot.

Table 6.3: Unique tag persistence on host per step for VirtualBox.

Round	**Step**					
	VM On (Step 5)		**VM Off (Step 7)**		**VM Reboot (Step 9)**	
	Tags	**%**	**Tags**	**%**	**Tags**	**%**
R1	24,949,786	99.7	6,940,081	27.7	490,085	1.9
R2	24,971,697	99.8	7,547,296	30.1	417,971	1.6
R3	24,980,514	99.9	8,438,257	33.7	974,322	3.8
Avg	24,967,332	99.8	7,641,878	30.5	627,459	2.5
Std Dev	13,000	0.1	615,000	2.5	247,000	1.0

Table 6.3 shows the persistence of unique VM tags in host memory while using VirtualBox. As observed with the other platforms, nearly all tags remain present while the VM is running (Step 5). After the VM is shutdown (Step 7), this platform has a similar average persistence rate to VMware (30.5%). On Step 9, the average is closer to Hyper-V than VMware (2.5%). Overall, VirtualBox demonstrates a memory persistence behavior similar to VMware.

Figure 6.1 shows the average percentage of tags found in host memory across three key steps of the experiment: while the VM is running (Step 5), after the VM is powered off (Step 7), and after it is rebooted (Step 9). For each virtualization platform, the solid lines represent the average percentage of detected tags, while the shaded areas indicate the corresponding standard deviation across the three test rounds. The results show a clear decrease in the number of retained tags after VM shutdown and an even larger decrease after the reboot. Despite the differences being low, it still denotes some variation in memory retention behavior between the tested hypervisors.

Figure 6.2 presents a per-run comparison (R1–R3) of the number of tags detected in host memory across the same three steps of the experiment (Steps 5,7 and 9). Each bar corresponds to a single experimental run and compares the three virtualization platforms under the same conditions.

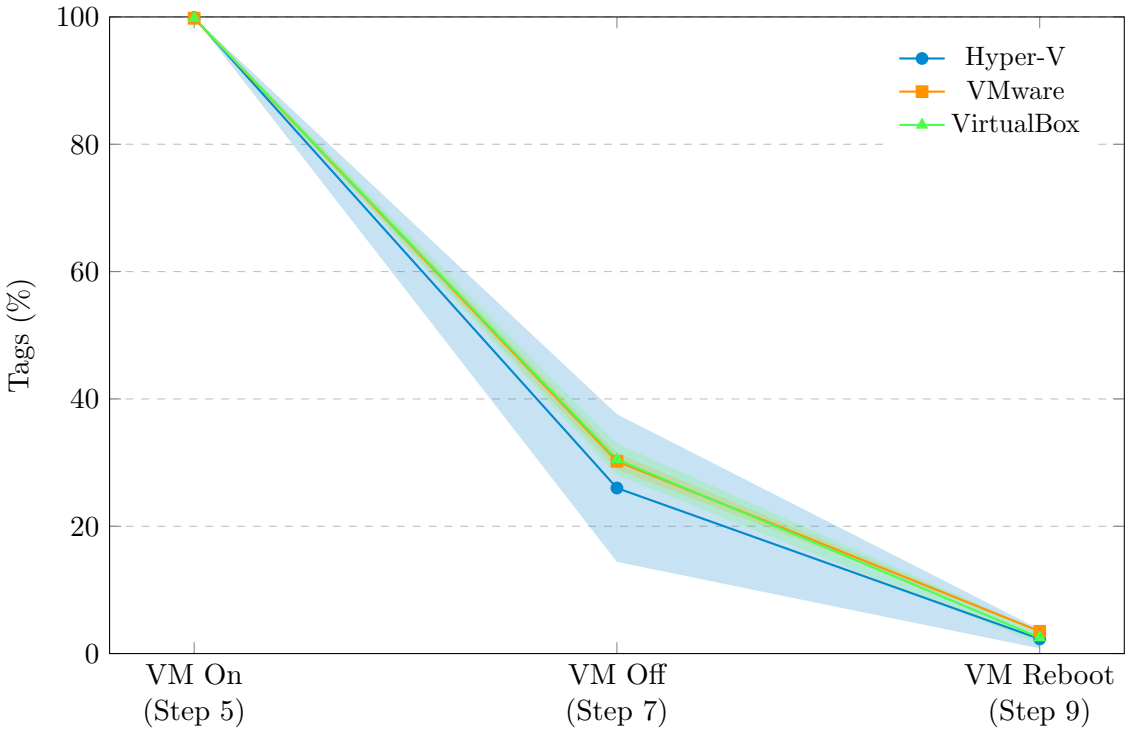


Figure 6.1: Average and standard deviation of unique tag persistence per host dump.

Across all runs, a consistent decrease in the number of detected tags is observed from Step 5 to Step 9, indicating progressive memory reuse or overwriting on the host. However, the magnitude of this reduction varies between platforms and runs. After VM shutdown, a noticeable portion of tags remains present in host memory, and although the reboot stage further reduces the tag count, residual data is still observed.

6.2 Memory Allocation

Memory allocation decisions influence how data is distributed across physical memory pages and how long allocated regions remain unchanged. While the presence and quantity of residual tags provide an initial indication of memory persistence, they do not capture how VM generated data is distributed within the host memory or how it evolves across the VM lifecycle. To address this, the analysis is extended to examine the physical location and relocation of the detected tags, using memory page precision.

The page-level analysis reveals that VM execution leaves a footprint in host physical memory. As shown by the number of memory pages of the host containing at least one

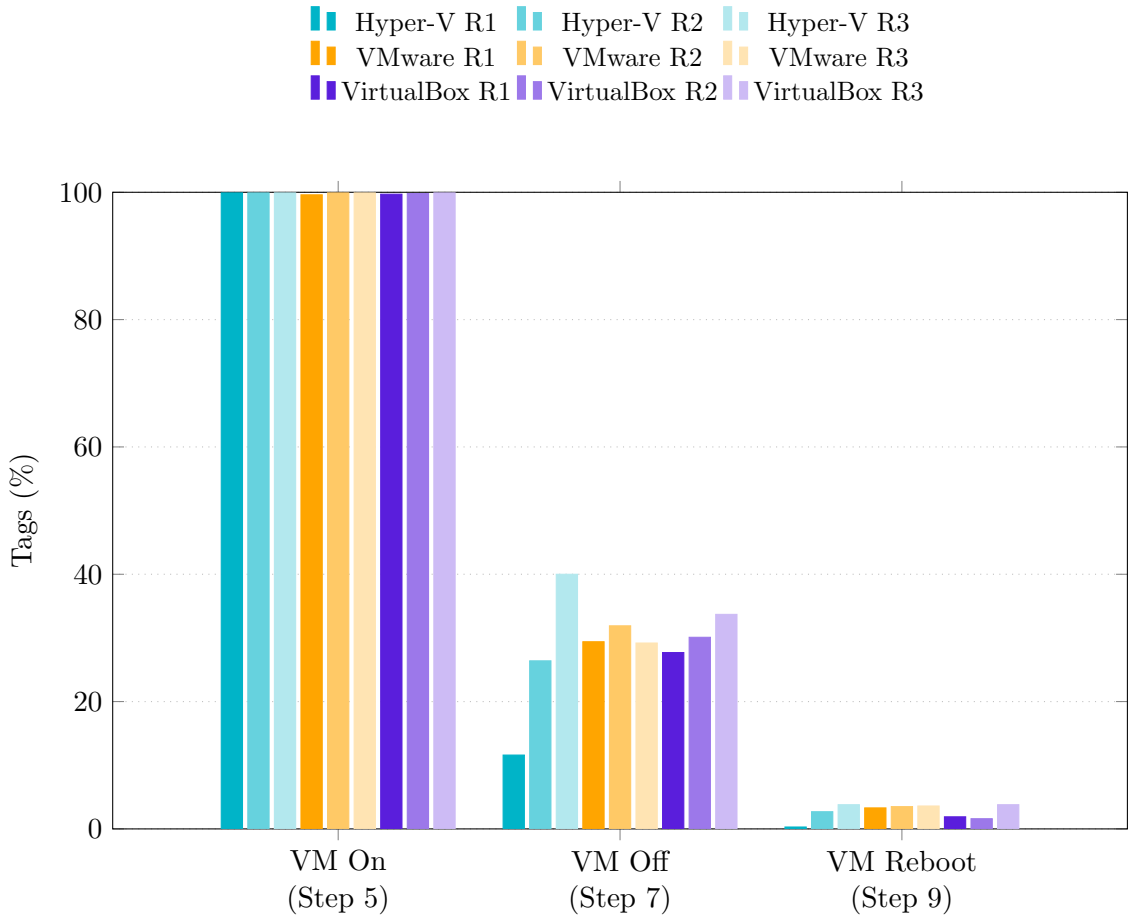


Figure 6.2: Injected tags persisting in host memory per hypervisor and test round (in percentage).

tag. The VM operation affects the host's memory while the VM is running due to memory management. Although this footprint is substantially reduced after the VM shutdown and reboot, it does not immediately disappear, indicating that VM data can persist in host memory even after the VM lifecycle has advanced.

Table 6.4 shows the number of memory pages containing at least one tag, per step. Each memory page has a size of 4 KB, corresponding to the default system page size used by the OS [28] and the hypervisor, which results in a total of 4,317,952 pages per memory dump. The results show very similar numbers across the different hypervisors. All have approximately 5% of memory pages with tags at Step 5, approximately 1,5% at Step 7 and approximately 0.1% at Step 9. This memory allocation analysis is performed using unique tags detected in memory. The extraction of unique tags is carried out using a dedicated script, provided in Appendix C.

Table 6.4: Memory pages containing tags.

VM State	Hyper-V		VMware		VirtualBox	
	Pages	**%**	**Pages**	**%**	**Pages**	**%**
VM On (Step 5)	225,896	5.2	226,039	5.2	225,855	5.2
VM Off (Step 7)	60,318	1.4	67,323	1.6	63,276	1.5
VM Reboot (Step 9)	6,085	0.1	7,651	0.2	4,489	0.1

6.3 Memory Mapping

Memory mapping refers to the dynamic relocation and reuse of memory pages within physical memory as a result of OS and hypervisor memory management decisions.

After identifying that there is a decrease in the number of persisted tags across the VM lifecycle, it is considered that the tags could be moved to a different location between steps.

To analyze the memory movement at the page level, an additional Python script is used to transform the page-count data into a spatial distribution of tags across host physical memory. The script aggregates tag counts into 1 MB memory regions and generates a density plot for each VM state (running, shutdown, and reboot).

Appendix D shows the script used to generate the page-level memory movement illustration. The script reads the per-page tag counts produced from the unique tag CSV files and converts each physical page index into a byte offset using a fixed page size of 4 KB. These offsets are then grouped into 1 MB bins, producing an array in which each element represents the total number of detected tags within a 1 MB region of host memory. Memory pages are grouped into 1 MB regions to reduce noise and better visualize memory movement. The script applies an optional logarithmic scale and normalization to enable comparison between different VM states.

Figures 6.3, 6.4, 6.5 show tag density on host memory during Round R3 for Hyper-V, VMware and VirtualBox, respectively. Each figure depicts the distribution of detected tags within the memory dumps of Steps 5, 7, and 9. The host memory is divided into fixed 1MB regions, shown as vertical bars. The color of each bar represents how many unique tags are found in that memory region and ranges from 0 to 1, where 0 (light yellow) means that no unique tags are found in that region and 1 (dark red) indicates the region with the highest number of unique tags for that VM state. Lighter colors represent fewer

tags, while darker colors represent higher tag density represent higher tag density.

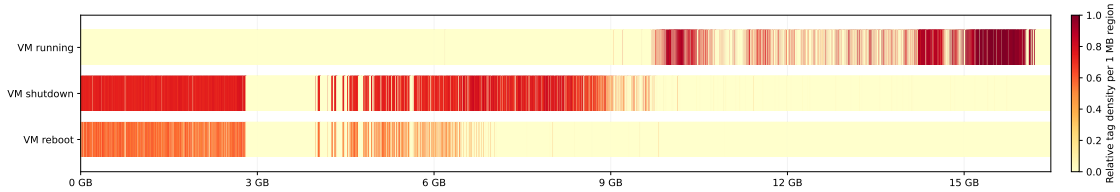


Figure 6.3: Hyper-V unique tag density on host memory

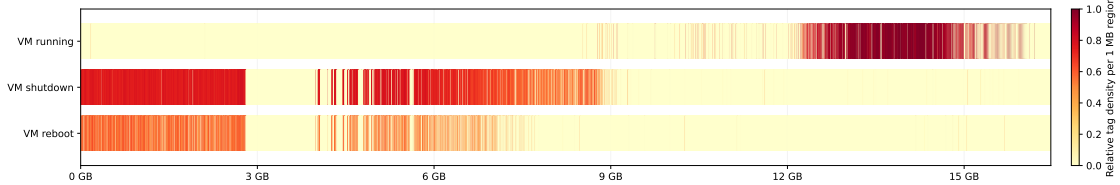


Figure 6.4: VMware unique tag density on host memory

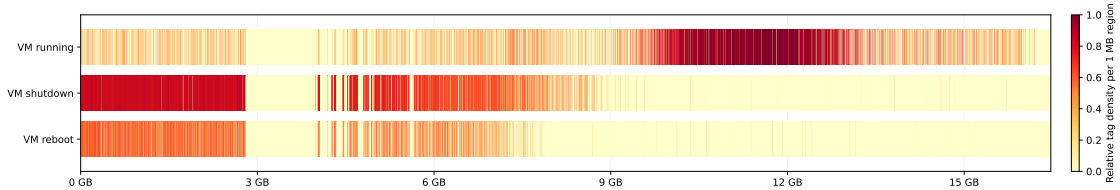


Figure 6.5: VirtualBox unique tag density on host memory

The results show that when the VM is running (Step 5), all three hypervisors show clear regions with high tag density. This means that the memory pages used by the VM are actively allocated and stored in specific areas of the host's physical memory. These regions are stable and indicate how each hypervisor maps guest memory to host memory during execution.

After the VM is shutdown (Step 7), a significant number of tags are still visible in host memory across all hypervisors. Most of these tags now appear in the lower memory regions, showing that memory previously used by the VM is not immediately cleared. This also suggests that shutting down a VM does not fully remove its data from the host memory. Following a VM reboot, the tag density is reduced, but tags are still present across large parts of host memory. This indicates that rebooting the VM also does not

guarantee that previously used memory pages are erased.

6.4 Memory Duplication

Memory duplication refers to the presence of identical data stored in multiple physical memory locations. Although both deduplication and duplication involve repeated memory content, they represent fundamentally different behaviors in virtualized environments. Memory deduplication is an intentional optimization strategy in which identical pages are consolidated into a single shared physical page using copy-on-write mechanisms. This reduces memory consumption and improves consolidation ratios, particularly when similar workloads are co-located.

In contrast, the duplication observed in this study refers to the unintended existence of multiple physical copies of the same guest-generated data. During the analysis, identical tags are detected across distinct regions of host memory, despite only a single instance of each tag being inserted into the VM. This indicates that guest-generated data can reside simultaneously in multiple host memory locations, likely as a result of allocation, relocation, or memory reuse processes.

Tables 6.5, 6.6 and 6.7 report the number of duplicated tags detected in the memory dumps. The *Repeated Tags* column indicates how many distinct tag identifiers appear more than once in memory. The *Total* column represents the total number of occurrences of those repeated tags, including all repetitions.

As shown on Listing 6.1, the number of repeated tags is 2, since two tag identifiers appear more than once and the total duplicated count is 7, corresponding to the sum of their occurrences (5 + 2).

Listing 6.1: Tag Occurance Example

```
1  Tag 00014213 - is found 5 times
2  Tag 02312231 - is found 2 times
3  Tag 00031293 - is found 1 time
```

Table 6.5 shows the persistence of duplicated tags in the host memory when using Hyper-V. For example, in Round 3, when the VM is running, about 1.2 million tags are repeated, accounting for a total of 2.6 million duplicated instances. After the VM is

Table 6.5: Duplicated tag persistence on host per step for Hyper-V.

Round	Step					
	VM On (Step 5)		VM Off (Step 7)		VM Reboot (Step 9)	
	Repeated Tags	Total	Repeated Tags	Total	Repeated Tags	Total
R1	8,801,875	20,944,261	1,134,653	3,124,512	22,475	59,470
R2	4,232,706	11,212,251	4,232,706	11,212,251	275,391	787,644
R3	1,207,673	2,662,681	4,712,195	12,767,889	461,267	1,297,342
Avg	4,747,418	11,606,398	3,359,851	9,034,884	253,044	714,819

shutdown, both the number of repeated tags and the total duplicated instances increase, reaching more than 4.7 million repeated tags and 12.7 million total instances. After the VM reboot, these values drop significantly, with only around 461 thousand repeated tags remaining.

Table 6.6: Duplicated tag persistence on host per step for VMware.

Round	Step					
	VM On (Step 5)		VM Off (Step 7)		VM Reboot (Step 9)	
	Repeated Tags	Total	Repeated Tags	Total	Repeated Tags	Total
R1	5,341,947	15,670,650	4,247,166	11,734,552	356,175	999,851
R2	4,487,599	9,442,253	4,243,339	12,087,556	343,497	967,039
R3	4,165,901	9,315,977	3,446,908	9,959,425	307,979	874,065
Avg	4,665,149	11,476,293	3,979,138	7,235,540	335,884	946,985

Table 6.6 presents the persistence of duplicated tags in host memory for VMware across VM lifecycle steps. Looking again at Round 3 as an example, around 4.2 million repeated tags are present while the VM is running, corresponding to about 9.3 million duplicated instances. After the VM is shutdown, both values decrease slightly, with approximately 3.4 million repeated tags and 10.0 million total instances. After reboot, the number of duplicated tags drops further, leaving about 308 thousand repeated tags and just under one million duplicated instances.

Table 6.7: Duplicated tag persistence on host per step for VirtualBox.

Round	Step					
	VM On (Step 5)		VM Off (Step 7)		VM Reboot (Step 9)	
	Repeated Tags	Total	Repeated Tags	Total	Repeated Tags	Total
R1	3,156,130	6,975,023	2,326,308	6,416,793	169,395	459,279
R2	8,086,840	17,719,252	3,026,313	8,572,609	133,792	372,560
R3	8,073,342	17,606,870	4,437,725	12,707,613	451,170	1,253,813
Avg	6,438,771	14,100,382	3,263,449	9,232,338	251,452	695,217

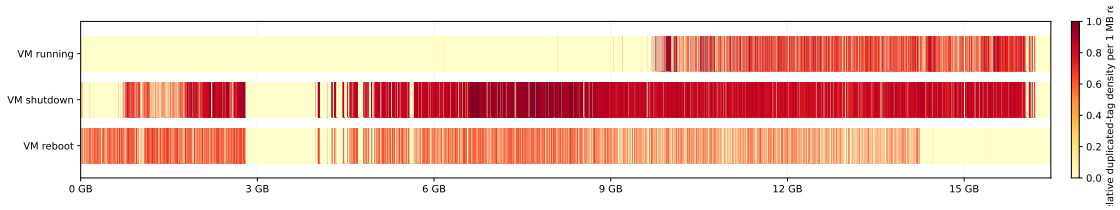


Figure 6.6: Hyper-V duplicated tag density on host memory

Table 6.7 shows the persistence of duplicated tags in host memory for VirtualBox. In Round 3, VirtualBox exhibits a high number of duplicated tags while the VM is running, with just over 8.0 million repeated tags and 17.6 million duplicated instances. After the VM is shutdown, these values decrease but remain substantial, with about 4.4 million repeated tags and 12.7 million total instances. Following the reboot, the number of duplicated tags drops significantly, leaving approximately 451 thousand repeated tags and 1.25 million duplicated instances.

Comparing Tables 6.5, 6.6 and 6.7, all three virtualization platforms demonstrate a similar behaviour. Hyper-V and VMware show comparable average levels of duplicated tag persistence, particularly after shutdown. VirtualBox shows higher variability across rounds, with some rounds showing larger duplicated tag counts during execution.

When comparing the duplicated tag results (Tables 6.5, 6.6, and 6.7) with the unique tag persistence results (Tables 6.1, 6.2 and 6.3), an important difference emerges. Across all platforms, the number of unique tags decreases significantly after VM shutdown and reboot. However, the duplicated tag analysis shows that many of the remaining tags appear multiple times in memory. Even when the total number of unique tags is reduced, the duplicated tag count remains high, indicating that VM generated data is often copied or reused across multiple memory locations. This comparison suggests that memory clearing reduces the number of distinct VM tags, but does not fully eliminate repeated copies of VM-generated data.

Figures 6.6, 6.7, 6.8 show the duplicated tag density on host memory during Round R3 for Hyper-V, VMware and VirtualBox.

Figure 6.6 shows the distribution of duplicated VM tags across host physical memory for Hyper-V. During the VM running state, duplicated tags are present mainly in higher memory regions. After the VM is shutdown, duplicated tags become more widespread

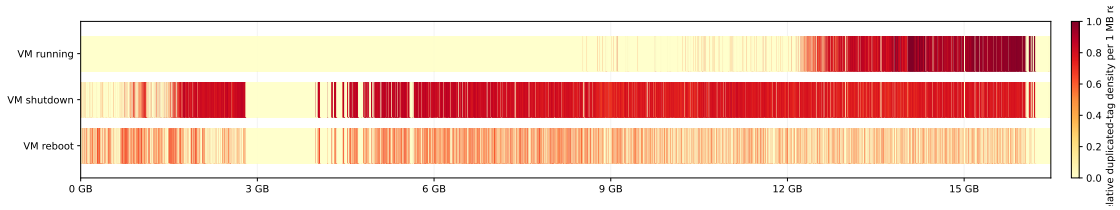


Figure 6.7: VMware duplicated tag density on host memory

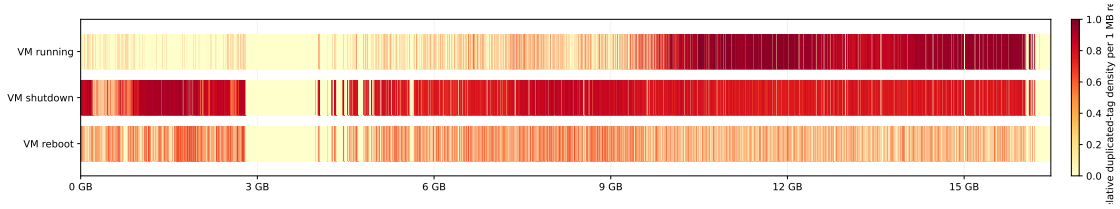


Figure 6.8: VirtualBox duplicated tag density on host memory

and concentrated, covering a larger range of host memory with higher density. Following a reboot, the density decreases, but duplicated tags are still visible across many regions.

Figure 6.7 presents the duplicated tag density for VMware across the three VM states. Similar to Hyper-V, duplicated tags are located in higher memory regions while the VM is running and become significantly more distributed and concentrated after shutdown. In the shutdown state, VMware shows a broad and dense distribution of duplicated tags across host memory. After reboot, the overall density is reduced, but duplicated tags remain spread across the memory space.

Figure 6.8 illustrates the duplicated tag density for VirtualBox. Compared to the other platforms, duplicated tags appear more equally distributed during the VM running state, with increasing density towards higher memory regions. As with Hyper-V and VMware, the shutdown state shows the highest concentration of duplicated tags across host memory. After reboot, the duplicated tag density is reduced but still present across many regions.

In summary, Figures 6.6, 6.7, 6.8 show where duplicated VM tags are located within the host's memory dumps. The results indicate that duplicated tags are not spread equally but are concentrated in specific memory regions. This pattern appears consistently across all platforms and VM states.

Chapter 7

Discussion

The experimental findings are discussed in this chapter by directly addressing each of the defined research questions and examining their implications for memory isolation and data confidentiality in virtualized environments.

7.1 Addressing the Research Questions

This section analyzes the experimental results in relation to the research questions defined in Chapter 1. Each question is examined individually, based on the findings presented in Chapter 6.

- RQ1: How is the information inserted into memory in a VM, handled, exposed and persisted in host environments?
  - The experiments showed that not all injected tags are present at the time of the memory dump. Although the program inserted 25 million tags, on average, approximately 99% are detected in the first host memory dump while the VM is still running (Tables 6.1, 6.2 and 6.3). This indicates that a small portion of the allocated memory is either released or overwritten due to normal system activity.

    Tags are consistently detected in host memory even after the VM is shutdown (Figure 6.1). This shows that parts of the guest memory remain resident in host memory after VM termination.

Memory tags are also found after the VM is rebooted (Figure 6.2). Although the number of detected tags decreases further, their presence suggests that some memory regions previously used by the VM have not yet been overwritten. Since the hypervisor controls the allocation and reuse of physical memory, guest memory pages can persist in volatile memory. Together, these results demonstrate that hard rebooting a VM does not necessarily guarantee the complete removal of memory traces from the host (see Table 6.4).

As reviewed by Faraz [15], memory allocation mechanisms are designed to balance efficiency and performance, dynamically allocating and reclaiming memory based on system demands.

- RQ2: When is memory cleared during virtualization processes, and how does this influence what remains accessible afterwards?
  - The results indicate that the VM does not fully clear its memory during shutdown. Tags injected into the VM remain visible in the host memory dump acquired after the VM is powered off (see Figure 6.2). This suggests that memory is not actively cleared at shutdown and is instead overwritten later as normal memory allocation resumes after reboot. As a result, the host retains portions of the VM's memory because the corresponding memory regions have not yet been overwritten. Moreover, the dumps obtained in the final step (Step 12) contain zero tags; this observation suggests that complete memory clearing occurs only after a host-level hard reboot, when physical memory is fully reinitialized.
- RQ3: What security and privacy implications arise from retention or leakage of memory content in virtualized environments?
  - Volatile memory can be a security risk in virtualized systems where many users share the same machine. If a VM's memory is not properly cleared, part of the data can remain in the host's memory after the VM terminates. This can weaken the isolation guarantees expected by users and potentially expose sensitive information from previous VM instances.

The results show that tag persistence differs across hypervisors and VM states (see Tables 6.1, 6.2, 6.3). While nearly all injected tags are detected on the host while the VM is running (above 99% for all platforms), tag persistence drops significantly after VM shutdown and reboot.

After shutdown, VirtualBox retains the highest percentage of tags on the host (30.5%), followed by VMware (30.2%) and Hyper-V (26.0%). After reboot, persistence is further reduced, but tags remain detectable for all hypervisors, with VMware showing the highest persistence (3.5%), followed by VirtualBox (2.5%) and Hyper-V (2.3%).

7.2 Overview on Privacy Risks

This section examines the privacy implications of the experimental findings. While the current study does not evaluate active exploitation scenarios, the observed memory persistence behavior raises important considerations regarding data confidentiality and isolation guarantees in virtualized environments. The following discussion interprets the results from a privacy perspective.

The persistence of volatile memory content in virtualized environments introduces potential privacy considerations, particularly in multi-tenant infrastructures where multiple users share the same physical host. The experimental results demonstrate that VM-generated data can remain present in host memory beyond VM lifecycle transitions, including shutdown and reboot states. While such persistence does not specifically imply immediate exploitability, it challenges common assumptions regarding automatic memory sanitization and isolation across VM boundaries.

The observed behavior, characterized by dynamic memory reuse, relocation of data across physical regions, and the presence of duplicated content, suggests that residual information can persist as a consequence of normal allocation and reclamation processes. In environments where host-level memory access is possible (e.g., through privileged access, forensic analysis, or misconfiguration), such residual data could increase the exposure surface of previously executed workloads.

Prior research has recognized volatile memory persistence as a security concern and

proposed mitigation strategies, including explicit memory clearing and limiting the lifetime of sensitive data in RAM [14]. These approaches emphasize the importance of proactive memory sanitization rather than reliance on implicit clearing behavior.

The findings of the current study emphasize the need for careful configuration of memory management mechanisms and transparent reclamation policies in virtualized systems, particularly in cloud environments with distinct trust boundaries.

Chapter 8

Conclusions

This work investigated how memory content generated inside VMs persists and evolves within host memory across different VM lifecycle states. Three virtualization platforms (Hyper-V, VMware and VirtualBox) are evaluated by injecting 25 million uniquely identifiable tags into VM memory and analyzing host memory dumps collected at multiple stages of execution.

The results show that not all injected tags remain detectable over time and that the amount of residual data varies across platforms. Importantly, data generated by the VM is consistently found in the host's memory even after the VM is shutdown or rebooted, indicating that memory is not fully cleared throughout the VM lifecycle. Although page-level retention decreases across states, most tags are not removed but instead relocated to different host physical pages, often across large memory distances. This behavior indicates that persistence occurs through dynamic memory reuse and relocation rather than stable retention of specific host memory regions. In addition, the analysis reveals clear evidence of memory duplication, showing that a significant portion of the injected data is replicated across multiple host memory pages.

These findings indicate that memory content generated within a VM can persist beyond the intended execution period and remain observable in host memory across lifecycle transitions. The observed persistence does not result from the static retention of specific physical memory regions. Instead, it is associated with dynamic memory allocation and reuse processes, in which host pages are reassigned while still containing residual data. This suggests that memory persistence in virtualized environments is influenced by how

the hypervisor manages memory over time, rather than by fixed physical mappings. As a result, guest-generated data can continue to appear in host memory due to allocation and reuse mechanisms that operate transparently to the VM.

Future work could extend this analysis to large-scale and cloud-based virtualized infrastructures in order to examine memory persistence under realistic multi-tenant workloads. Evaluating additional hypervisors, configuration parameters, and memory management policies would help determine how different implementation choices influence memory reuse and isolation behavior. Moreover, a more detailed investigation of memory duplication can clarify the mechanisms responsible for repeated occurrences of guest-generated data in host memory, particularly in environments with higher resource contention and dynamic allocation patterns.

References

[1] Waqas Ahmed and Baber Aslam. "A comparison of windows physical memory acquisition tools". In: *MILCOM 2015 - 2015 IEEE Military Communications Conference*. 2015, pp. 1292–1297. DOI: 10.1109/MILCOM.2015.7357623.

[2] Bushra Albelooshi et al. "Experimental proof: Data remanence in cloud vms". In: *2015 IEEE 8th International Conference on Cloud Computing*. IEEE. 2015, pp. 1017–1020.

[3] Michael Armbrust et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[4] AWS. *Type 1 vs Type 2 Hypervisors - Difference Between Hypervisor Types*. https://aws.amazon.com/compare/the-difference-between-type-1-and-type-2-hypervisors//.

[5] Paul Barham et al. "Xen and the art of virtualization". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: Association for Computing Machinery, 2003, pp. 164–177. ISBN: 1581137575. DOI: 10.1145/945445.945462. URL: https://doi.org/10.1145/945445.945462.

[6] Sean Barker et al. "An empirical study of memory sharing in virtual machines". In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 2012, pp. 273–284.

[7] Mohammad-Mahdi Bazm et al. "Isolation in cloud computing infrastructures: new security challenges". In: *Annals of Telecommunications* 74.3 (2019), pp. 197–209.

[8] Sururah A. Bello et al. "Cloud computing in construction industry: Use cases, benefits and challenges". In: *Automation in Construction* 122 (2021), p. 103441. ISSN:

0926-5805. DOI: https://doi.org/10.1016/j.autcon.2020.103441. URL: https://www.sciencedirect.com/science/article/pii/S0926580520310219.

[9] BrainKart. *Memory Virtualization.* https://www.brainkart.com/article/Memory-Virtualization_11340/.

[10] Andrew Case, Lodovico Marziale, and Golden G Richard III. "Dynamic recreation of kernel data structures for live forensics". In: *Digital Investigation* 7 (2010), S32–S40.

[11] Jim Chow et al. "Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation." In: *USENIX Security Symposium*. 2005, pp. 22–22.

[12] Brendan Dolan-Gavitt et al. "Virtuoso: Narrowing the semantic gap in virtual machine introspection". In: *2011 IEEE symposium on security and privacy*. IEEE. 2011, pp. 297–312.

[13] D.G. Elliott et al. "Computational RAM: implementing processors in memory". In: *IEEE Design & Test of Computers* 16.1 (1999), pp. 32–41. DOI: 10.1109/54.748803.

[14] William Enck et al. "Defending Against Attacks on Main Memory Persistence". In: *2008 Annual Computer Security Applications Conference (ACSAC)*. 2008, pp. 65–74. DOI: 10.1109/ACSAC.2008.45.

[15] Ahmed Faraz. "A review of memory allocation and management in computer systems". In: *Computer Science & Engineering: An International Journal (CSEIJ)* 6.4 (2016).

[16] FlackBox. *Virtual Processor Scheduling – VMware & Microsoft.* https://www.flackbox.com/virtual-processor-scheduling-how-vmware-and-microsoft-hypervisors-work-at-the-cpu-level//.

[17] GeeksForGeeks. *Random Access Memory (RAM).* https://www.geeksforgeeks.org/computer-science-fundamentals/random-access-memory-ram////.

[18] GeeksForGeeks. *Top 10 Cloud Platform Service Providers in 2025.* https://www.geeksforgeeks.org/blogs/top-cloud-platform-service-providers///.

[19] Riya Gohil and Hiren Patel. "Comparative Analysis of Cloud Platform: Amazon Web Service, Microsoft Azure, And Google Cloud Provider: A Review". In: *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. 2024, pp. 1–5. DOI: 10.1109/ICCCNT61001.2024.10725914.

[20] Peter Gutmann. "Secure deletion of data from magnetic and solid-state memory". In: *Proceedings of the sixth USENIX security symposium, San Jose, CA*. Vol. 14. 1996, pp. 77–89.

[21] J Alex Halderman et al. "Lest we remember: cold-boot attacks on encryption keys". In: *Communications of the ACM* 52.5 (2009), pp. 91–98.

[22] Kresimir Hausknecht, D Foit, and J Burić. "RAM data significance in digital forensics". In: *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2015, pp. 1372–1375.

[23] Michael Henson and Stephen Taylor. "Memory encryption: A survey of existing techniques". In: *ACM Comput. Surv.* 46.4 (Mar. 2014). ISSN: 0360-0300. DOI: 10.1145/2566673. URL: https://doi.org/10.1145/2566673.

[24] Ruizhe Huang et al. "Towards Efficient Hugepage-aware Memory Deduplication". In: *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless*. 2023, pp. 15–21.

[25] Seongwook Jin and Jaehyuk Huh. "Secure mmu: Architectural support for memory isolation among virtual machines". In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE. 2011, pp. 217–222.

[26] Lenovo. *How are characters stored in computer memory?* https://www.lenovo.com/us/en/glossary/character/////.

[27] Zaigham Mahmood. "Cloud Computing". In: *Computer Communications and Networks, DOI* 10 (2014), pp. 978–3.

[28] Microsoft. *What are the page sizes used by Windows on various processors?* https://devblogs.microsoft.com/oldnewthing/20210510-00/?p=105200/////.

[29] Debadatta Mishra and Purushottam Kulkarni. "A survey of memory management techniques in virtualized systems". In: *Computer Science Review* 29 (2018), pp. 56–73. ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2018.06.002. URL: https://www.sciencedirect.com/science/article/pii/S1574013716301186.

[30] Surya Nepal and Mukaddim Pathan. *Security, privacy and trust in cloud systems*. Springer, 2013.

[31] Gaoning Pan et al. "Breaking Isolation: A New Perspective on Hypervisor Exploitation via Cross-Domain Attacks". In: *arXiv preprint arXiv:2512.04260* (2025).

[32] Harshita Pandey. "Cloud Based Services". In: GeeksForGeeks, Feb. 2023. URL: https://www.geeksforgeeks.org/cloud-based-services/.

[33] Gerald J Popek and Robert P Goldberg. "Formal requirements for virtualizable third generation architectures". In: *Communications of the ACM* 17.7 (1974), pp. 412–421.

[34] Red Hat. *What is virtualization management?* https://www.redhat.com/en/topics/virtualization/what-is-virtualization-management//.

[35] Thomas Ristenpart et al. "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds". In: *Proceedings of the 16th ACM conference on Computer and communications security*. 2009, pp. 199–212.

[36] Ella Savchenko, Jenny Ottmann, and Felix Freiling. "In the time loop: Data remanence in main memory of virtual machines". In: *Forensic Science International: Digital Investigation* 49 (2024). DFRWS USA 2024 - Selected Papers from the 24th Annual Digital Forensics Research Conference USA, p. 301758. ISSN: 2666-2817. DOI: https://doi.org/10.1016/j.fsidi.2024.301758. URL: https://www.sciencedirect.com/science/article/pii/S2666281724000775.

[37] Furquan Shaikh et al. "VMDedup: Memory De-duplication in Hypervisor". In: *2014 IEEE International Conference on Cloud Engineering*. 2014, pp. 379–384. DOI: 10.1109/IC2E.2014.69.

[38] Johannes Sianipar, Muhammad Sukmana, and Christoph Meinel. "Moving Sensitive Data Against Live Memory Dumping, Spectre and Meltdown Attacks". In: *2018 26th International Conference on Systems Engineering (ICSEng)*. 2018, pp. 1–8. DOI: 10.1109/ICSENG.2018.8638178.

[39] Abraham Silberschatz, James L Peterson, and Peter B Galvin. *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc., 1991.

[40] Ashish Singh and Kakali Chatterjee. "Cloud security issues and challenges: A survey". In: *Journal of Network and Computer Applications* 79 (2017), pp. 88–115. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2016.11.027. URL: https://www.sciencedirect.com/science/article/pii/S1084804516302983.

[41] Bradley Lee Snyder and James H Jones. "Determining the effectiveness of data remanence prevention in the AWS cloud". In: *2019 7th International Symposium on Digital Forensics and Security (ISDFS)*. IEEE. 2019, pp. 1–6.

[42] Subashini Subashini and Veeraruna Kavitha. "A survey on security issues in service delivery models of cloud computing". In: *Journal of network and computer applications* 34.1 (2011), pp. 1–11.

[43] Kuniyasu Suzaki et al. "Memory deduplication as a threat to the guest OS". In: *Proceedings of the Fourth European Workshop on System Security*. 2011, pp. 1–6.

[44] TechTarget. *What is storage virtualization?* https://www.techtarget.com/searchstorage/definition/storage-virtualization//.

[45] Martin Unterguggenberger et al. "TME-Box: Scalable In-Process Isolation through Intel TME-MK Memory Encryption". In: *arXiv preprint arXiv:2407.10740* (2024).

Appendices

Appendix A

Code for tag injection program

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/sysinfo.h>

#define NUM_TAGS 25000000
#define TAG_TEMPLATE "VBOX-R1-%08d-05-08-2025-20:00:00"
#define TAG_SIZE 37

int main() {
    struct sysinfo mem_info;
    if (sysinfo(&mem_info) != 0) {
        perror("Failed to get system memory info");
        return 1;
    }

    size_t total_size = NUM_TAGS * TAG_SIZE;
    size_t available_ram = (mem_info.freeram + mem_info.bufferram) * mem_info.
        mem_unit;
    printf("Total size needed: %zu bytes\n", total_size);
    printf("Free RAM available: %zu bytes\n", available_ram);

    if (total_size > available_ram) {
        fprintf(stderr, "Error: Not enough RAM (needed %zu, available %zu)\n",
                total_size, available_ram);
        return 1;
    }
    // Allocate memory using malloc
```

```
29      char *memory = malloc(total_size);
30      if (memory == NULL) {
31          perror("Memory␣allocation␣failed");
32          return 1;
33      }
34
35      // Populate the memory with tags
36      size_t tags_populated = 0;
37      for (int i = 0; i < NUM_TAGS; ++i) {
38          char tag[TAG_SIZE];
39          int written = snprintf(tag, sizeof(tag), TAG_TEMPLATE, i);
40          if (written != TAG_SIZE - 1) {
41              fprintf(stderr,
42                      "Error:␣Tag␣format␣incorrect␣for␣index␣%d.␣Expected␣%d␣chars,␣
                            got␣%d.␣Tag:␣'%s'\n",
43                      i, TAG_SIZE - 1, written, tag);
44              break;
45          }
46          memcpy(memory + i * TAG_SIZE, tag, TAG_SIZE);
47          ++tags_populated;
48      }
49
50      printf("%zu/%d␣tags␣populated.\n", tags_populated, NUM_TAGS);
51      free(memory);
52      return 0;
53  }
```

Appendix B

Memory dumps CSV conversion script

```python
import csv
import re
from pathlib import Path

DUMP_DIR = Path("../dumps")
OUTPUT_DIR = Path("../output/hits")

PAGE_SIZE = 4096
CHUNK_SIZE = 64 * 1024 * 1024 # 64 MB
OVERLAP_BYTES = PAGE_SIZE

ASCII_RE = re.compile(rb"HYPE-R1-(\d{8})-26-07-2025-00:00:00")

UTF16_RE = re.compile(
    rb"(H\x00Y\x00P\x00E\x00-\x00R\x001\x00-\x00"
    rb"(?:\d\x00){8}"
    rb"-\x002\x006\x00-\x000\x007\x00-\x002\x000\x002\x005\x00"
    rb"-\x000\x000\x00:\x000\x000\x00:\x000\x000\x00)"
)

def scan_dump(filename: str, dump_state: str) -> None:
    dump_path = DUMP_DIR / filename
    output_path = OUTPUT_DIR / f"{dump_state}.csv"
    output_path.parent.mkdir(parents=True, exist_ok=True)

    with dump_path.open("rb") as f, output_path.open("w", newline="") as out:
```

```
        writer = csv.writer(out)
        writer.writerow(["tag_id", "page"])

        file_offset = 0
        tail = b""

        while True:
            chunk = f.read(CHUNK_SIZE)
            if not chunk:
                break

            data = tail + chunk
            base = file_offset - len(tail)

            for m in ASCII_RE.finditer(data):
                offset = base + m.start()
                tag_id = m.group(1).decode("ascii")
                writer.writerow([tag_id, offset // PAGE_SIZE])

            for m in UTF16_RE.finditer(data):
                offset = base + m.start()
                raw = m.group(0)
                digits = raw.split(b"-\x00")[2][:16:2]
                tag_id = digits.decode("ascii")
                writer.writerow([tag_id, offset // PAGE_SIZE])

            tail = data[-OVERLAP_BYTES:]
            file_offset += len(chunk)

if __name__ == "__main__":
    scan_dump("hostDump_VMrunning.bin", "running")
    scan_dump("hostDump_VMshutdown.bin", "shutdown")
    scan_dump("hostDump_VMreboot.bin", "reboot")
```

Appendix C

Unique tags extraction script

```python
from pathlib import Path
import pandas as pd

STATES = ["running", "shutdown", "reboot"]

IN_DIR = Path("../output/hits")
OUT_DIR = Path("../output/unique")
OUT_DIR.mkdir(parents=True, exist_ok=True)

def main():
    summary_rows = []

    for state in STATES:
        in_csv = IN_DIR / f"{state}.csv"
        if not in_csv.exists():
            raise FileNotFoundError(f"Missing input file: {in_csv}")

        df = pd.read_csv(
            in_csv,
            usecols=["tag_id", "page"],
            dtype={"tag_id": "string", "page": "int64"},
        )

        unique_df = (
            df.dropna(subset=["tag_id"])
              .drop_duplicates(subset=["tag_id"], keep="first")
              .sort_values("tag_id")
        )

```

```
        out_csv = OUT_DIR / f"{state}_unique.csv"
        unique_df.to_csv(out_csv, index=False)

if __name__ == "__main__":
    main()
```

Appendix D

Memory movement script of unique tags

```python
import os
import math
import argparse
from pathlib import Path

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib as mpl

STATES = ["running", "shutdown", "reboot"]

BASE_STATS = Path("../output/stats")
OUT_DIR = Path("../output/figures")
OUT_DIR.mkdir(parents=True, exist_ok=True)

DUMPS = {
    "running": "../dumps/host_running.bin",
    "shutdown": "../dumps/host_shutdown.bin",
    "reboot": "../dumps/host_reboot.bin",
}

PAGE_SIZE = 4096
MB = 1024 * 1024

def total_pages_from_dump(state: str, fallback_total_pages: int) -> int:
```

```
    try:
        size = os.path.getsize(DUMPS[state])
        return max(1, size // PAGE_SIZE)
    except Exception:
        return fallback_total_pages


def read_page_counts(state: str) -> pd.DataFrame:
    p = BASE_STATS / f"{state}_page_counts.csv"
    if not p.exists():
        raise FileNotFoundError(f"Missing input CSV: {p}")
    return pd.read_csv(p, usecols=["page", "tag_count"], dtype={"page": "int64", "tag_count":
         "int64"})


def pages_to_1mb_bins(df: pd.DataFrame, total_pages: int, page_size: int) -> np.ndarray:
    if df.empty:
        total_bytes = total_pages * page_size
        num_bins = int(math.ceil(total_bytes / MB))
        return np.zeros(num_bins, dtype=np.int64)

    # Compute bin index for each page
    byte_pos = df["page"].to_numpy(dtype=np.int64) * page_size
    bin_idx = (byte_pos // MB).astype(np.int64)

    total_bytes = total_pages * page_size
    num_bins = int(math.ceil(total_bytes / MB))

    bins = np.zeros(num_bins, dtype=np.int64)
    np.add.at(bins, bin_idx, df["tag_count"].to_numpy(dtype=np.int64))
    return bins


def save_pdf(path: Path):
    plt.tight_layout()
    plt.savefig(path, format="pdf")
    plt.close()
    print(f"[saved] {path}")


def normalize_bins(
    bins_by_state: dict,
    norm_mode: str,
    scale: str,
```

```
    clip_q: float | None,
):
    transformed = {}
    for s, arr in bins_by_state.items():
        a = arr.astype(np.float64)
        if scale == "log":
            a = np.log1p(a)
        transformed[s] = a

    if norm_mode == "global":
        base = np.concatenate(list(transformed.values())) if transformed else np.array([0.0])
        if clip_q is not None and base.size > 0:
            clip_val = np.quantile(base, clip_q)
        else:
            clip_val = None

        base2 = base if clip_val is None else np.minimum(base, clip_val)
        global_max = float(base2.max()) if base2.size else 0.0
        global_max = max(global_max, 1e-12)

        out = {}
        for s, a in transformed.items():
            a2 = a if clip_val is None else np.minimum(a, clip_val)
            out[s] = np.clip(a2 / global_max, 0.0, 1.0)
        return out, global_max

    out = {}
    max_used = 0.0
    for s, a in transformed.items():
        a2 = a.copy()
        if clip_q is not None and a2.size > 0:
            clip_val = np.quantile(a2, clip_q)
            a2 = np.minimum(a2, clip_val)

        m = float(a2.max()) if a2.size else 0.0
        m = max(m, 1e-12)
        max_used = max(max_used, m)
        out[s] = np.clip(a2 / m, 0.0, 1.0)
    return out, max_used


def make_white_to_red_cmap():
    return mpl.cm.get_cmap("YlOrRd")

```

```
114
115 def plot_density_pdf(
116     bins_by_state: dict,
117     normed_by_state: dict,
118     out_pdf: Path,
119     show_colorbar: bool = True,
120 ):
121     max_bins = max(len(v) for v in bins_by_state.values())
122     x = np.arange(max_bins)
123
124     cmap = make_white_to_red_cmap()
125     norm = mpl.colors.Normalize(vmin=0.0, vmax=1.0)
126
127     fig_w = 16
128     fig_h = 2.8
129     fig, ax = plt.subplots(figsize=(fig_w, fig_h))
130
131     y_positions = {"running": 2, "shutdown": 1, "reboot": 0}
132     y_labels = ["VM reboot", "VM shutdown", "VM running"]
133
134     for s in STATES:
135         y = y_positions[s]
136         vals = normed_by_state[s]
137         if len(vals) < max_bins:
138             vals = np.pad(vals, (0, max_bins - len(vals)), constant_values=0.0)
139
140         colors = cmap(norm(vals))
141         ax.vlines(x, y - 0.38, y + 0.38, colors=colors, linewidth=0.6)
142
143     # X axis as GB positions
144     # 1 bin = 1 MB. Convert bin index to GB.
145     total_gb = (max_bins * MB) / (1024**3)
146     tick_gb = np.arange(0, math.ceil(total_gb) + 1, 3) # every 3 GB
147     tick_pos = (tick_gb * (1024**3) / MB).astype(int)
148     tick_pos = tick_pos[tick_pos <= max_bins]
149
150     ax.set_xticks(tick_pos)
151     ax.set_xticklabels([f"{g} GB" for g in tick_gb[: len(tick_pos)]])
152     ax.set_yticks([0, 1, 2])
153     ax.set_yticklabels(["VM reboot", "VM shutdown", "VM running"])
154
155     ax.set_xlim(0, max_bins)
156     ax.set_ylim(-0.7, 2.7)
157
```

```
    # Light grid for orientation
    ax.grid(axis="x", alpha=0.15)

    # Colorbar
    if show_colorbar:
        sm = mpl.cm.ScalarMappable(cmap=cmap, norm=norm)
        sm.set_array([])
        cbar = plt.colorbar(sm, ax=ax, fraction=0.035, pad=0.02)
        cbar.set_label("Relative tag density per 1 MB region")

    save_pdf(out_pdf)
def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--out", default=str(OUT_DIR / "hyperv_tag_density.pdf"), help="Output PDF
        path.")
    ap.add_argument("--scale", choices=["linear", "log"], default="log",
                    help="Scale counts before coloring (default: log).")
    ap.add_argument("--norm", choices=["global", "per_state"], default="global",
                    help="Normalization across states (default: global, recommended).")
    ap.add_argument("--clip", type=float, default=0.99,
                    help="Quantile clip in [0,1] to reduce saturation (default: 0.99). Use 0 to
                        disable.")
    ap.add_argument("--fallback-gb", type=float, default=16.0,
                    help="If dumps missing, assume this host RAM size for x-axis length (default:
                        16GB).")
    args = ap.parse_args()

    clip_q = None
    if args.clip and 0.0 < args.clip < 1.0:
        clip_q = args.clip

    fallback_total_pages = int((args.fallback_gb * 1024**3) // PAGE_SIZE)

    # Load and bin for each state
    bins_by_state = {}
    total_pages_ref = None
    for s in STATES:
        df = read_page_counts(s)
        total_pages = total_pages_from_dump(s, fallback_total_pages)
        total_pages_ref = total_pages_ref or total_pages

        bins_by_state[s] = pages_to_1mb_bins(df, total_pages=total_pages, page_size=PAGE_SIZE)

    normed_by_state, max_used = normalize_bins(
```

```
        bins_by_state=bins_by_state,
        norm_mode=args.norm,
        scale=args.scale,
        clip_q=clip_q,
    )

    out_pdf = Path(args.out)
    plot_density_pdf(
        bins_by_state=bins_by_state,
        normed_by_state=normed_by_state,
        out_pdf=out_pdf,
        show_colorbar=True,
    )

if __name__ == "__main__":
    main()
```