ARL-TR-8029 ● MAY 2017

US Army Research Laboratory

A Survey on Security Isolation of Virtualization, Containers, and Unikernels

by Michael J De Lucia

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-8029 ● MAY 2017

US Army Research Laboratory

A Survey on Security Isolation of Virtualization, Containers, and Unikernels

by Michael J De Lucia
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
May 2017	Technical Report	September 2016–October 2017

4. TITLE AND SUBTITLE
A Survey on Security Isolation of Virtualization, Containers, and Unikernels

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Michael J De Lucia

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-CIN-D
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-8029

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Virtualization, containers, and unikernels are the fundamental technologies that enabled the widespread use of the cloud; therefore, a comparison of their security isolation characteristics is necessary to understand the potential threats. Each of these technologies contains subtle differences in the methodology and software architecture to provide secure isolation between guests. All 3 of these technologies commonly provide the same functionality with varying degrees of overhead; however, the security isolation is based on a vastly different approach. This report first gives the background of each of these technologies followed by the security isolation aspects of each technology. A suggestion on metrics to further evaluate security characteristics of each technology is proposed to guide future evaluations.

15. SUBJECT TERMS
security isolation, virtualization, containers, unikernels, the cloud

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Michael J De Lucia
Unclassified	Unclassified	Unclassified	UU	18	19b. TELEPHONE NUMBER (Include area code) 410-278-6508

Contents

List of Figures

List of Tables

1. Introduction

Virtualization, containers, and unikernels are the fundamental technologies that enabled the widespread use of the cloud; therefore, a comparison of their security isolation characteristics is necessary to understand the potential threats. Each of these technologies contains subtle differences in the methodology and software architecture to provide secure isolation between guests. All 3 of these technologies commonly provide the same functionality with varying degrees of overhead; however, the security isolation is based on a vastly different approach. This report first gives the background of each of these technologies, followed by the security isolation aspects of each technology. A suggestion on metrics to further evaluate security characteristics of each technology is proposed to guide future evaluations.

2. Background and History

Virtualization of x86 systems is a fundamental technology that has been in existence for some time, whereas containers are rapidly being adopted and unikernels are just emerging. An x86 virtualization solution was initially released in 1998 by VMware to provide a software-based solution, and the initial release of the ESXi type 1 bare-metal hypervisor was developed in 2001.[1] In that same year, XEN released an open-source type 1 bare-metal hypervisor. At that time, all virtualization was achieved in software, resulting in slower performance and additional work for the hypervisor. In 2005, Intel introduced hardware virtualization extensions, commonly referred to as "hardware-assisted virtualization" (i.e., Intel VT-x and VT-d).[1] These extensions were added to the processor to simplify the tasks of a hypervisor and increase performance. In 2007, KVM released a Linux kernel module-based hypervisor, which leveraged hardware-assisted virtualization extensions.[1] Since 2013, the trend has moved from virtualization toward containers. However, containers are not virtualization but lend themselves to similar concepts. Containers assisted in the increased utilization of hardware within a cloud environment by reducing the server (e.g., web server, database server) footprint, resulting in an increased number of services a single physical host could run. Then, in 2014, the concept of a unikernel was introduced to make the server footprint even smaller and increase performance.[2] The evolution of each of these technologies improves performance and builds on fundamental security isolation at the kernel or processor level in different ways. To better understand these differences, a brief introduction and review of the x86 processor ring levels follows.

3. x86 Processor Ring Levels

To comprehend the security aspects of virtualization, container, and unikernel technology, an understanding of the x86 processor security-ring model is essential. The x86 has a concept called security rings (ring 0-3) to define the privilege level of instructions run within the processor. Conventionally, ring level 0 is the most privileged level and the only level that can communicate directly with the hardware, whereas ring level 3 is the least privileged. These security rings are enforced at the processor level as memory is accessed. Traditional operating systems (OSs) leverage these privilege levels with ring level 0 for kernel space, ring level 3 for user space, and ring levels 1 and 2 are not used.

Within most OSs, kernel space contains the drivers and fundamental components that communicate directly with the hardware, whereas the user space contains the applications running on the OS. For example, because a user application such as a Web browser needs to access the hardware, a system call into the kernel is made, and during this time a context switch will happen. The context switch from the user space to the kernel space occurs and allows the kernel to communicate directly with the hardware on behalf of the Web browser. In addition, traditional hardware allows each process to operate on pages within the main memory. Each running process contains a page table to convert the virtual memory address to the physical memory addresses within main memory. These page tables also contain access rights and the associated ring level of the process. The page tables can only be modified by the kernel (ring 0 process) and cannot be modified by a user-level process. Every process contains its own page table of the memory space available to it to prevent user processes from interfering with one another. These ring levels combined with the hypervisor ("kernel") are what provide isolation between virtual machines (VMs) and are key to virtualization security isolation concepts. The ring levels are leveraged in virtualization by a hypervisor, which is either implemented as a separate hypervisor kernel or as a Linux Kernel Module.

4. Virtualization

In virtualization, the hypervisor provides a layer between the guest OS and the physical hardware. There are 2 different types of hypervisors: type 1 (e.g., VMware ESXi, KVM, and XEN), which is bare metal, and type 2 (e.g., VMware Workstation, and Virtual Box), which runs on a host OS. The hypervisor is responsible for communicating directly with and sharing the hardware by scheduling central processing unit (CPU) time for each guest VM and allocating virtual memory from the physical memory to each VM. As discussed earlier, only ring level 0 holds the privileges necessary to communicate directly with the

hardware. Therefore, the hypervisor must run at ring level 0. As a result, the guest VM kernel must be run at a lower privilege level, which does not allow direct communication to the hardware, resulting in a kernel failure.

The traditional software approach to virtualization is to either modify the guest VM kernel code or transparently trap these privileged instructions. VMware pioneered the approach of transparently trapping privileged instructions and scanning memory of the guest VM and then rewriting these instructions to redirect through the hypervisor.[1] Another approach to this problem is called paravirtualization, which relies on modification of the guest VM kernel. This technique requires patches to be applied to the guest VM kernel to communicate indirectly through the hypervisor to the hardware. Although these solutions provide an adequate approach to virtualization, each requires some sort of modification to the guest VM kernel. To further improve performance and decrease the complexity of the hypervisor, hardware-assisted virtualization was developed to provide a hardware-based solution, which extends the processor instruction set.

Hardware-assisted virtualization extends the x86 processor instruction set to allow virtualization within hardware, allowing a reduction of complexity of the hypervisor and overhead, leading to an increase in performance. Although both AMD and Intel have similar solutions, the focus of this discussion is Intel. In 2005, Intel introduced hardware-assisted virtualization support to their processors with VT-x and VT-d. The extended instruction set has support for a guest and host domain, which is commonly referred to as ring level -1 (Fig. 1). These extensions allow the hypervisor to run in the host domain, whereas the guest VMs run in the guest domain, thus allowing the guest VM kernel to run at ring level 0. Similar to the traditional case, ring level 0-3 exists in both guest and host domains. In addition, the extensions assist with memory management for each guest VM and can also allow guest VMs direct access to a dedicated network card and other peripheral devices. Hardware-assisted virtualization is offered as an option in each of the hypervisor vendors surveyed for this report.

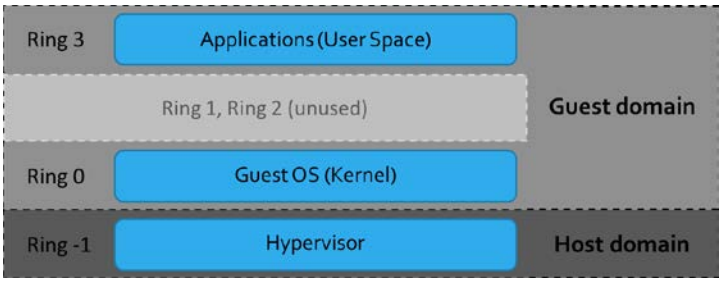

Fig. 1 Hardware virtualization ring levels

There are many different hypervisor platforms available, such as Microsoft Hyper-V, VMware ESXi, KVM, and XEN. This report focuses on both KVM and XEN with the use of hardware-assisted virtualization, because both platforms leverage hardware-assisted virtualization and are open source with a greater amount of documentation available. In addition, QEMU can be used on both hypervisors to provide hardware emulation. The purpose of QEMU is to allow an unmodified guest OS to share other hardware such as networking, storage (input/output [I/O]), and many other devices. Both XEN and KVM offer paravirtualization for I/O devices as an alternative to QEMU to increase performance at the cost of modifying the guest OS. XEN and KVM provide functionality of virtualization but they differ greatly in the design of the software architecture.

XEN's architecture was designed as a hypervisor microkernel (Fig. 2) to be separated from the drivers and administration, which are hosted within a privileged VM commonly referred to as dom0. The other unprivileged guest VMs running on the hypervisor are referred to as domU. The privileged VM, dom0, can communicate with all of the other untrusted VMs through a shared memory ring setup within the XEN hypervisor. In hardware-assisted virtualization, the XEN hypervisor is running within the host domain, whereas dom0 is running within the guest domain. Both the hypervisor and the guest VM OS kernel are running at ring level 0 within the host and guest domain, respectively. The dom0 OS is based on a customized Linux kernel. XEN offers the option of either using paravirtualization or QEMU to communicate from the guest OS to the physical hardware devices. Paravirtualization requires patching of the guest OS kernel, whereas QEMU does not require patching of the guest OS kernel. In addition, XEN allows for further separation of drivers into respective driver domains, similar to dom0. These driver domains provide for further isolation of specific drivers. For example, the network drivers could be separated into a separate network driver domain. It has been proven with the use of QubesOS[3] that these domains provide a strong amount of isolation to the specific driver. The strong amount of isolation between drivers is enabled by the microkernel design of XEN.

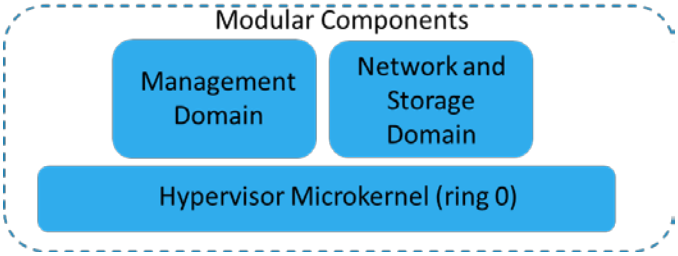

Fig. 2 Microkernel

In contrast, KVM is a monolithic (Fig. 3) design, leveraging the existing Linux kernel to provide administration and drivers, with a KVM kernel module composing the hypervisor functionality. The KVM hypervisor views each guest VM as a Linux-based process. Additional isolation between each guest VM is provided by SELinux, which restricts process privileges. From an architectural perspective, KVM can be considered a monolithic architecture because the hypervisor kernel module, drivers, and administration tools are all hosted within the Linux kernel. In hardware-assisted virtualization, the kernel module and drivers and the KVM user process are run within the host domain at ring levels 0 and 3, respectively. Paravirtualization for hardware I/O drivers could also be leveraged in KVM. Although, the drivers and the hypervisor kernel module are both located within the kernel, there is no isolation between them.

Fig. 3 Monolithic kernel

As discussed earlier, both XEN and KVM are architecturally different in terms of the hypervisor design. XEN contains a separation between the hypervisor kernel and the driver modules within dom0, resulting in a microkernel architecture. In comparison, KVM combines both the hypervisor module and the driver modules within the Linux kernel, resulting in a monolithic kernel architecture. In the XEN microkernel architecture, the hypervisor and drivers (dom0) run at different privilege levels of host and guest domain, respectively, with the use of hardware-assisted virtualization. In KVM and hardware-assisted virtualization, both the hypervisor and drivers run at ring level 0 within the host domain. It has been argued that a microkernel architecture provides better isolation at the cost of additional layers of separation and complexity and allowed seemingly trusted code to be run in an untrusted domain; however, monolithic architecture executed trusted code within a trusted domain, was less complex, but allowed the possibility of hypervisor compromise.[4]

The ability to leverage hardware-assisted virtualization with XEN and split drivers in separate driver domains allows trusted code to be run in trusted domains. For example, an attacker compromising a network driver domain will not be able to escape to an application or kernel of a guest VM and will not be able to compromise the hypervisor. The fundamental reason for this greater amount of protection is attributed to a greater degree of isolation provided by the XEN hypervisor design. Although XEN allows for a greater degree of isolation between the hypervisor and the drivers, KVM still provides a degree of isolation between the hypervisor and

guest VMs, which is increased by the use of hardware-assisted virtualization. Through the use of XEN, paravirtualization, and hardware-assisted virtualization, a far greater degree of isolation exists between drivers, hypervisor, and guest VMs as a result of a microkernel architecture. The use of paravirtualization allows for seemingly trusted code to be executed at a privilege level of 0 while still allowing for a microkernel architecture. However, there is still a large amount of overhead within the guest domains, with a large amount of redundant functionality (process scheduling, driver code, etc.) in the guest kernels. Although reduction of overhead can lead to increased performance, it can also help to increase security by decreasing the complexity. The greater the degrees of complexity and introduction of additional code lead to a chance of added vulnerabilities. Further work must be performed to provide a clear delineation between the boundaries and trust levels of each component (hypervisor, guest domains, and drivers) and eliminate the redundant overhead, perhaps by using unikernels or containers.

4. Containers

Containers provide similar functionality to virtualization, but there are several subtle differences. The concept of containers was developed to reduce the footprint, allowing developers to ease the transition from the development and testing phase to deployment in production. Containers enable further use of the tremendous computational power of modern hardware. The fundamental Linux construct of namespaces, cgroups, and capabilities makes the container concept possible. There are many container products available, such as Flockport LXC, Dockers, and many others. In addition, Microsoft has recently started offering Windows containers in Windows Server 2016 and Windows 10. While there are some differences between container solutions, this report focuses on Dockers, as the fundamental concepts are the same.

Each separate Docker application is a component called a microservice, which can be combined to compose a larger application. A Docker container is a specific instance of an image, which is a union read-only file system, combining several layers (different file systems) and contains the application and dependencies to be executed. These images are stored in a registry that can be either public or private. The Docker Engine is responsible for creating a fresh container instance from the images stored within the registry, setting network configurations and namespaces, and restricting the allowed Linux kernel capabilities for the container.

As seen in Fig. 4, the Docker Engine runs as an application with root privileges within the ring level 3 (user space), whereas the namespaces and Linux kernel capabilities are enforced by the Linux kernel at ring level 0. The results of an

investigation into Docker Security[6] concluded that Docker was secure in terms of isolation between each container and provided a significant amount of network isolation. The only negative finding was the vulnerability of traffic capture/sniffing because all of the traffic is sent over a bridge interface of the host system to each container, which is common within any networked environment.[6]

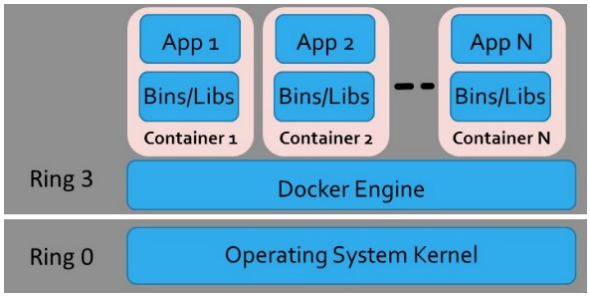

Fig. 4 Docker

However, because the Docker Engine is running within the user space, all containers share the same kernel (shown in Fig. 4). As a result, if the kernel is compromised, all containers running on the host will be compromised as well. In addition, there is no isolation between each of the drivers and the kernel namespace mechanism providing isolation between containers. Each container is simply a process running on the same kernel at the ring level 3 privilege. In addition, other user space applications (ring level 3) running on the host could allow the ability to interfere with the container processes that are running. Although containers provide an adequate amount of isolation between each other, unikernels were introduced and can provide even more isolation with a further reduced attack surface and a dramatic increase in runtime performance.

6. Unikernels

Unikernel technology was introduced by the University of Cambridge in 2013–2015 with the research and development of MirageOS. Since that time, several other alternatives to MirageOS have been introduced such as OSv and Rump kernels. All of these unikernels are based on the same concept of reducing the footprint of an application running in the cloud. For example, traditionally, each Web server application running in the cloud requires the extra overhead of the OS containing a monolithic kernel that carries unnecessary code and services for a single application (Fig. 5).

Fig. 5 Traditional VM vs. unikernel

Unikernel is a new approach to building a specialized kernel that contains the application code, runtime (i.e., Mirage Runtime in MirageOS), and necessary kernel dependencies. This specialized unikernel can then be run on a hypervisor. This concept again allows for the building of a mircoservices architecture and is sometimes referred to as a "library operating system".[2] Similar to containers, unikernels are only able to run a single process. Therefore, if multiple parallel instances are required, the unikernel must be duplicated and run several times on the hypervisor. The hardware consumption constraints and scheduling of hardware access is handled by the hypervisor. For example, the MirageOS system contains many different core fundamental library components such as the TCP/IP (Transmission Control Protocol/Internet Protocol) () module to leverage in building a single application.[2] In MirageOS, the application and all of the dependency components and Mirage Runtime are "compiled" into a single unikernel to run on the hypervisor.[2] The footprint of these unikernels are considerably smaller than a full VM containing an OS and application. In addition, the boot time is measured in milliseconds, and networking performance has been shown to be better than a traditional VM-based server.[2]

The increase in performance, with respect to networking, is partially attributed to the application running within the same address space as the "kernel components", alleviating a context switch between privilege levels for the user application and kernel. In a unikernel, a delineation between user and kernel space does not exist. Therefore, both the user application and core fundamental kernel components are running at the same privilege of ring level 0. In addition, the attack surface is reduced because unnecessary services and code are removed, leaving only necessary kernel dependencies. The isolation between each unikernel is provided by the hypervisor, similar to the VM process. Unikernel technology is still in its infancy and will require further research to streamline the ease of unikernel application development and deployment.

7. Isolation Comparisons

Each of these 3 technologies are similar in function but have subtle differences in security characteristics, making some more isolated and secure than others (see

Table 1 for comparisons). The greatest amount of isolation provided between each instance is virtualization. The isolation provided is attributed to the isolation by the hypervisor, which leverages the memory segmentation provided in hardware by ring levels and hardware virtualization extensions. However, although hypervisors provide good isolation, not all hypervisors provide the same amount of protection. Both XEN and KVM provide a good amount of isolation, but XEN benefits from separating the drivers from the hypervisor into a privileged VM. In addition, paravirtualization and the use of separate driver domains can be combined in XEN to provide an even stronger amount of isolation and protection from kernel and driver vulnerabilities. The use of paravirtualization requires patching of the guest OS kernel. Although patching is required, it is more secure than using QEMU to run an unmodified guest OS. The QEMU module is extremely complex because it emulates many hardware components, which leads to a higher possibility of vulnerabilities.[3] For example, the VENOM vulnerability found in 2015 leveraged a bug in the floppy drive emulation code within QEMU to break out of the VM to the host running the hypervisor.[7] The hypervisor microkernel is the most trusted component and should be well-isolated from other potential attack vectors. By separating the drivers from the hypervisor, the microkernel allows the attack surface to be reduced to the code of the hypervisor. Although many kernel drivers have been shown to contain vulnerabilities, it is imperative to separate them from the hypervisor.

Table 1 Comparison of security characteristics

Type	Products	Ring level	Isolation provided by	Image size
Virtualization	XEN, KVM, Hyper-V, ESXi	Level 0 or -1	Hypervisor	Large
Container	Docker, Flockport LXC	Level 3, enforced at Level 0	Host Kernel	Medium
Unikernel	MirageOS, OSv, Rumpkernel	Level 0	Hypervisor	Small

8. Conclusions

Although virtualization provides good isolation, it is unrealistic and extremely inefficient to install a single application on an OS within a VM. A better alternative in terms of footprint and reduced overhead would be to use containers, but this does not offer strong isolation like virtualization. Nonetheless, the emerging concept of unikernels could provide a small footprint, reduced overhead, and strong isolation due to the use of the hypervisor; however, this is not the optimal solution due to its lack of privilege levels within the unikernel itself. The unikernel lacks the ability

to separate a seemingly trusted “kernel code” from the application code itself. This means the unikernel kernel code and application code are running at the same ring level of 0 while running on a hardware-assisted hypervisor. Because both application code and kernel code are running at the same privilege level, this alleviates the need for processor context switches, which partially contributes to the performance speedup. Although the combination of hypervisors and unikernels could be considered as an “adequate” solution, an “optimal” solution has not been developed yet.

An optimal solution, in terms of security isolation, requires the separation of trusted versus untrusted code, reduced overhead, and an increase in performance. A combined approach for an optimized solution will require the use of many of the best features of each of the 3 technologies. In the search for an optimal solution, it is imperative to have metrics to assess the solution. Proposed metrics include architectural differences (monolithic vs. microkernel), the use of privilege levels (ring 0-3), and attack surface measurements. These metrics will help guide the experimentation and uncover vulnerabilities that are not otherwise apparent. Further analysis and experimentation of all 3 of these technologies are required to advance the security state of applications. Each technology brings a security aspect to assist in the advancement of a secure, small footprint, and reduced overhead environment. A hybrid combination of all 3 technologies could show potential in advancing the security state of applications.

9. References

1. Understanding full virtualization, paravirtualization, and hardware assist. Vmware; 2008 Mar 11 [accessed 2016 Sep 23]. http://www.vmware.com/techpapers/2007/understanding-full-virtualization-paravirtualizat-1008.html.
2. Madhavapeddy A, Scott D. Unikernels: the rise of the virtual library operating system. Commun ACM. 2014;57(1):61–69.
3. Rutkowska J, Wojtczuk R. Qubes OS Architecture. Invisible Things Lab; 2010 [accessed 2016 Aug 15]. https://www.qubes-os.org/attachment/wiki/QubesArchitecture/arch-spec-0.3.pdf.
4. Shropshire J. Analysis of monolithic and microkernel architectures: towards secure hypervisor design. Presented at the 47th Hawaii International Conference on System Science; 2014 Jan 6–9; Waikoloa, Hawaii.
5. Getting started with LXC. Flockport; 2014 [accessed 2016 Sep 23]. https://www.flockport.com/lxc-guide/.
6. Bui T. Analysis of Docker security. Aalto (Finland): Aalto University School of Science; 2015 Jan 13 [accessed 2016 Sep 23]. http://arxiv.org/abs/1501.02967.
7. Venezia P. The venom vulnerability: little details bite back. InfoWord; 2015 May 18 [accessed 2017 Jan 9]. http://www.inforworld.com/article/2922315/virtualization/venom-security-vulnerability-little-details-bite-back.html.
8. Chisnall D. The definitive guide to the XEN hypervisor. Upper Saddle River (NJ): Prentice Hall; 2007.
9. Docker Overview. Docker; 2016 [accessed 2016 Sep 22]. https://docs.docker.com/engine/understanding-docker/.
10. KVM – kernel-based virtual machine. Redhat; 2015 Jan 20 [accessed 2016 Sep 29]. https://www.redhat.com/en/resources/kvm-%E2%80%93-kernel-based-virtual-machine.

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIO L IMAL HRA MAIL & RECORDS MGMT
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
11 (7 PDF, 4 HC)	DIR USARL RDRL SER P AMIRTHARAJ RDRL SER E DEL ROSARIO (1 HC) J WILSON J PENN (3 HC) R PROIE E VIVEIROS RDRL WML B F DE LUCIA