A survey of isolation techniques

Draft Copy

Arun Viswanathan and B.C. Neuman
University of Southern California, Information Sciences Institute

Abstract

The general purpose computer has become pervasive and is supporting an increasing number of functions, including music, video, gaming, communications, banking, business, process control, and critical infrastructure. The use of a single computer for multiple functions, and the interconnection of multiple computers through a common network have reduced the isolation that protected such functions in the past. If we are to use common systems for multiple functions, we need mechanisms that provide the isolation needed to protect each function from interference by others. Without such isolation, vulnerabilities and mis-trust in any part of the system can propagate and compromise the rest of the system. Isolation techniques form an integral part of security in systems and networks. This work surveys isolation techniques for operating systems and networks and describes systems built using those techniques. An intuitive taxonomy is proposed for organizing these techniques. The paper aims to provide a critical understanding of what already exists and what needs to be done with respect to isolation security for building next-generation secure systems.

1. Introduction

The unprecedented growth of the network-enabled personal computer in a variety of form-factors, including the laptop, desktop, mobile, handheld devices, when combined with the increase importance of the internet has changed the dynamics of security. Computers which were once used only for number-crunching and electronic data storage are now used for functions not previously envisioned. Among these functions are music, video, gaming, communications, banking, business, process control, and critical infrastructure.

Unfortunately, this increased importance of the network-enabled personal computer has enabled new mechanisms for attack through the easy propagation of *malware*[1]. Malware works on the premise that infecting one part of the system gives easy access to other parts of the system and in most cases access to the network too. As computers have become more connected to one another, the malware threat has increased. It is essential that instead of just relying on defense techniques, the next generation of system software must be designed from the ground-up to provide stronger isolation of functions. As an example, consider the case of an employee logging remotely into a corporate network. If his system is infected with viruses introduced by malicious online games he may have played, the infection can now easily spread into the employees' corporate network. Thus, isolation becomes an essential building block for providing security in today's systems and networks. One must note here that use of isolation as a building block is not limited to only security but finds uses in other areas of software engineering like providing failure isolation for components, component modularity, improving system structure and easing system evolution. This work focuses primarily on using isolation for security.

The need for isolation is not a new requirement in Computer Security. It was articulated clearly almost 35 years ago in *Computer Security Technology Planning Study Report* [1] by James Anderson, much before the advent of the Internet and the widespread proliferation of personal computers. The report identifies that resource-sharing between users is the key cause of security and privacy issues and that execution of programs must be controlled to build a secure resource sharing system. The notion of a *Reference Monitor* was then proposed as a building block for designing a secure resource sharing system. As defined in [1], *the function of the Reference Monitor is to validate all references (to programs, data, peripherals etc) made by programs in execution against those authorized for the subject (user, etc). The Reference Monitor is also responsible for assuring that the references to shared resource objects are of the right kind (read, read/write etc)*. It can be argued that most of the isolation research presented in this paper is some variation on the generic concept of a reference monitor.

---

1 The term *malware* in this paper is used to refer to all types of existing bad software including worms, viruses, rootkits, Trojans, spyware etc.

In current operating systems, the notion of *isolation of functions* is supported at a minimum by operating system *processes*. The operating system kernel provides this isolation by using abstractions of virtual memory provided by the hardware. This simple isolation has proven extremely inadequate in dealing with the various penetration techniques used by malware which allow an adversary to access resources otherwise not meant to be accessed by a process. Research over the years has focused on providing the necessary isolation for mitigation of these issues. The well-known mitigation techniques include language-based protection provided by type-safe languages and certifying compilers, sandboxing-based protection as provided by different kinds of reference monitors, kernel-based protections, hardware-based protection and the more recently popularized hypervisor-based protection. This paper surveys the currently employed isolation techniques and proposes an intuitive taxonomy to organize the techniques. Around thirty different systems which implement those techniques (single or a combination of techniques) are then analyzed and categorized as per the taxonomy.

Unfortunately, techniques that have been very useful in improving the security of individual computer systems do not extend very well into the network environment. Controls on the flow of information within an operating system are easily circumvented when a process communicates unconstrained with processes outside the local system. Most of the current techniques treat the system in isolation and do not really concern themselves with the network aspect. But, in the face of emerging threats as outlined above, entities involved in a transaction over a distributed network require stronger guarantees of isolation than currently provided.

The paper aims to provide a critical understanding of what already exists and identify new challenges in isolation mechanisms for building next-generation secure systems. To that end, this paper serves as a bibliography of isolation techniques and should provide a single reference point for researchers in this area. The rest of the paper is structured as follows: Section 2 introduces required terminology and formulates the isolation problem in a generic way. Sections 3 and 4 present taxonomy for categorizing the isolation techniques. Section 5 provides a brief description of systems built using isolation techniques. Section 6 presents observations made using the survey and Section 7 concludes the paper.

2. The isolation problem

This section introduces a generic model for isolation in systems. Two examples are presented to highlight the necessity of isolation in individual systems and networked systems. The terminology introduced below is then used to compare the systems that are described in section 4.

2.1 Terminology

Task - A task is an abstraction for a piece of software that consumes resources to perform a specific function. Examples of tasks can be any piece of software such as web browsers and FTP server.

Shared Resource - A shared resource is at least one of a CPU, storage, or a network. Tasks perform their functions by sharing resources with other tasks. While sharing of resources helps in efficient utilization of resources, it is also one of the root causes of security issues.

Protection Domain - A *Protection Domain* is a logical container for task(s) and shared resources. The protection domain enforces the protection boundary policies using isolation techniques. An example of a protection domain is an operating system which allows multiple tasks to run while competing for CPU/Memory/Network resources or a virtual system [1] which abstracts resources from one or more systems and presents them as if part of a single system.

Trusted Computing Base (TCB) – As defined by Lampson [2], TCB is *a small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security.*

With the above definitions in mind we can define the *task isolation problem* as, *the problem of separating*

and protecting tasks from other executing tasks within a protection domain and from tasks in other protection domains.

Illustration 1 provides a visualization of the terms just described. The tasks are represented by ovals and they run within protection domains. The protection domains enforce policies on the running tasks. The thickness of the protection domains implies the extent to which the TCB is required to enforce the protection domain. The isolation model also allows for *recursive protection domains,* that is, there could be protection domains within protection domains as shown in the figure.

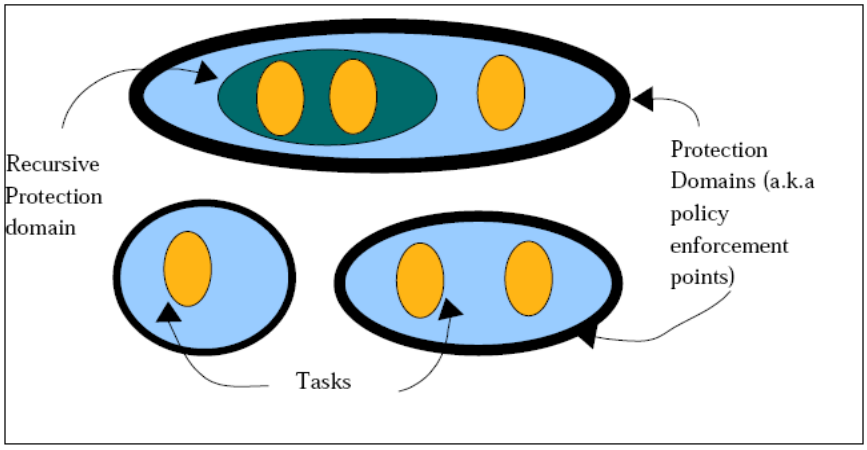

Illustration1: Isolation Model

2.2 Examples of protection domains

Consider an operating system (see illustration. 2) to be a protection domain comprising different tasks. The isolation problem for the OS is to provide separation between the tasks running on the same node and also prevent other outside tasks from inadvertently accessing tasks in its domain.

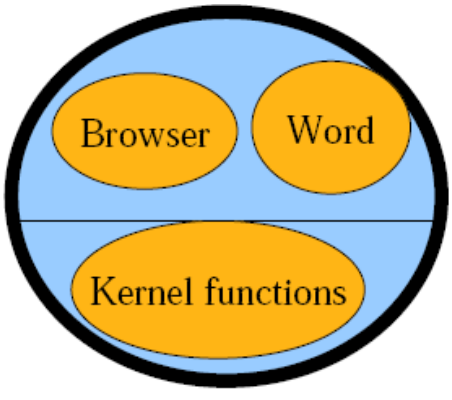

Illustration 2: Example of a local protection domain. The browser, word software and the kernel functions are all part of same protection domain.

Consider the distributed case (see illustration. 2), in which a user logs into his company network over VPN to access documents for editing. In this case, the protection domain comprises the companies file server, the user's VPN client and the user's word processing software. The protection domain policies should protect the tasks from each other and also prevent any unidentified task (like the browser in the example) from entering this virtual system and also prevents anything from within the domain from making illegal accesses outside the system. It is easy to see here that more is required of the TCB in the distributed case than in the single case.

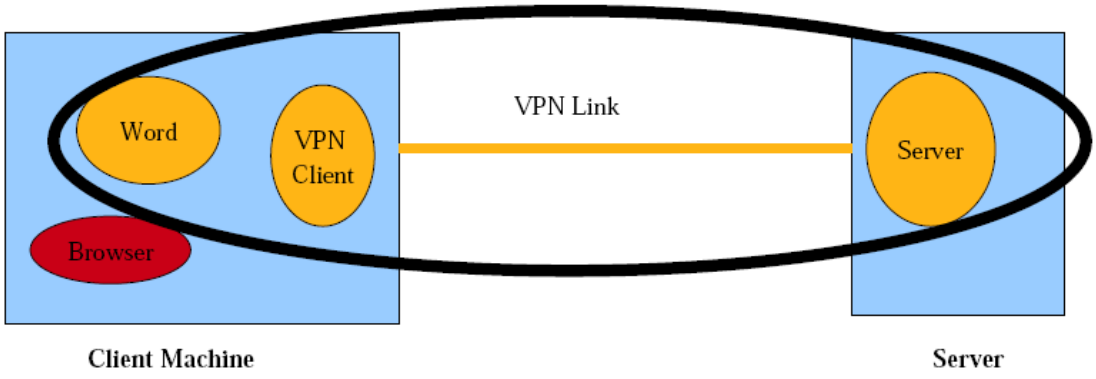

Illustration 3: Networked Protection Domain

3. Taxonomy of isolation techniques

As seen in section 2, there are two types of systems for which isolation can be defined: individual and networked. The distinction between individual systems and networked systems is important because the threat model is different in both cases and hence the isolation requirements change. Also, the TCB in a networked system is distributed and hence the isolation mechanism has to take into account the trustworthiness of the TCBs in the system. Though this is well understood, it is surprising to note the little attention that isolation in networked systems has received as compared to the plethora of work done in individual systems. Current techniques for providing isolation in networked systems rely mostly on encrypting the traffic flowing on the network. But, as the example in section 2 shows, this does not necessarily solve the isolation problem. This point is explored further in section 6. As there is not much reported work on isolation in networked systems, the taxonomy presented here in this section applies only to individual systems. Illustration 3 provides a visual of the taxonomy.

Isolation techniques for individual systems can be categorized as follows: (1) Language-based isolation (2) Sandbox-based isolation (3) Virtual Machine based isolation (4) OS-kernel based isolation and (5) Hardware-based isolation. Individual sections explore the categories further. The individual categories are explored further below.

3.1 Language-based Isolation

Language-based isolation is isolation provided by programming languages, language compilers, assemblers and/or by runtime environments. They essentially force the programmer to comply with a set of rules that

enforce isolation between the program and other programs. For example, type-safe programming languages, such as Modula, Scheme, or Java, ensure that operations are only applied to appropriate values. As stated in [3], they do so by guaranteeing that programs can only access appropriate memory locations and that control transfers happen to appropriate program points.

Language based isolation can be further classified into two categories based on how it is enforced:

(a) ***Type Systems:*** Type-systems enforce isolation using either programming language semantics, compilers, run-time systems or a combination of the above. They make sure that programs can access only appropriate memory locations and that control transfers happen only to appropriate program points. Examples of type systems are programming languages like ML, Scheme, Modula-3, Java. Isolation boundaries are usually enforced using a combination of compile-time and run-time techniques.

As stated in [3], the key idea in type systems to enforce security policies is to shift the burden of proving that a program complies with a policy from the code recipient (the end user) to the code producer (the programmer). The programmer is forced to write the program in conformance with the type system; the end user need only type-check the code to ensure that it is safe to execute. Type-systems thus provide a lightweight way to enforce memory and control safety as opposed to the traditional ways of enforcing isolation using OS/hardware protection mechanisms like page-tables/segmentation etc.

Example Systems: SPIN, Modula-3, ML, Scheme, Java

(b) ***Certifying Compilers:*** As defined in [3], a *certifying compiler is a compiler that, when given source code satisfying a particular security policy, not only produces object code but also produces a certificate—machine-checkable evidence that the object code respects the policy.* A widely-used example is the *javac* compiler for Java which produces annotated byte-code from a high-level Java program. The annotated code (in JVML) becomes the certificate for the high-level code. This certificate is then used by a verifier to verify the type-safety of the byte-code.

In general, certifying compilers allow for a very versatile way of specifying security policies as compared to simple type-safety notions provided by programming languages. Javac compiler, TAL (Typed Assembly language) and PCC (Proof Carrying Code) are examples of certifying compiler approaches. TAL [5] or Typed Assembly Language extends the notion of JVML byte-code to machine language of real machines. As stated in [5], the basic idea behind TAL is to encode high-level typing abstractions and security policies from high-level language constructs to typed machine language. The obvious advantages of doing so are that it removes the dependence on virtual machines, high level languages and aids faster execution on the native platform. But, TAL still can only enforce the traditional type-safe security policies.

The most widely recognized example of Certifying Compilers is PCC [4] or Proof-Carrying Code. PCC [4] uses formal proofs represented in a meta-language to represent the safety and correctness requirements of code. The code-receiver on the other end uses a theorem-prover to validate the proof attached with the code. The TCB in PCC is very small and it does not impose run-time penalties. A big advantage of using PCC is that it has a very expressive logic to construct desired policies and thus is more powerful than simple type-safety.

Examples: Proof Carrying Code [4], TAL [5]

3.2 Sandbox based isolation

Sandboxing was first introduced by Wahbe et.al.[14] and was defined by them as a technique for software

encapsulation of untrusted code such that it may not escape its *fault domain*[2]. Their technique involved modifying the program binary to insert additional checks around each store or jump so that the program could only make jumps into its own code segment and write to data only in its data segments. The definition of sandboxing as adopted in this paper is not restricted to the definition presented in [14] but instead is defined more generically as *"a technique for creating confined execution environments for running untrusted programs on the same machine"* A simple example of a sandbox is the UNIX *chroot* jail which is a very simple way to provide remote users a restricted and virtual view of the file system.

There are three techniques that may be used to implement sandboxes based on how a program's activity may be restricted. One can use the simple technique of applying access controls like file ACLs/user ACLs to restrict activity or restrict behavior at the instruction level or at the system call level. It is easy to see that each level of monitoring has its own advantages and disadvantages and they all help protect against specific threats. Based on the above three techniques, sandbox-based isolation can be further classified as:

(a) ***Instruction Set Architecture based (ISA based):*** Any sandbox technique that restricts activity of programs at the instruction level falls under this category. One of easiest ways in which this type of sandboxing is implemented is by binary rewriting where additional instructions are added before existing code (esp. jumps and stores) to check for memory access violations. One of the problems with this technique is that it is architecture dependent and more so it may also be dependent on the type of the instruction set (RISC or CISC). Earlier techniques in Software Fault Isolation (SFI) suffered from being only RISC capable but recent projects like PittSFIeld [15] have shown that SFI can be applied to CISC architectures too.

Examples: SFI [14], Program Shepherding [10], Inline Reference Monitors [58], PittSFIeld [15].

(b) ***Application Binary Interface based (ABI based):*** The ABI is the interface between an application program and the operating system or between the application and its libraries. In this technique, a sandbox is constructed by controlling the ABIs that an application uses to restrict its behavior. A common way of specifying restricted ABIs for an application is via a configuration file.

Examples: Janus [12], MAPBox [13], Consh [20], SLIC [25]

(c) ***Access control based (ACL based):*** In access control based sandboxing, the restriction of activity is provided via explicit permissions that are applied to accesses by programs. In this class, the access control can be applied to files, network, processes, pipes, devices etc. UNIX *chroot* is the simplest example of this type of sandboxing, where the remote users' view of file system is constricted to a directory by controlling accesses to other directories in the file system. The difference between ACL-based and ABI-based sandboxing is that the ABI-based sandboxing relies only on preventing system calls while the ACL-based method is a little more generic in its applications. Another subtle difference is the fact that in ACL-based systems, the system calls may themselves be modified to implement policies while in ABI-based systems the system calls are prevented from executing.

Examples: UNIX Chroot-jail, TRON [17], Sub-Operating Systems [15], SubDomains [21], Consh [20], FreeBSD Jails [21], Chakravyuha [16], SBOX [22], One Way Isolation [18]

3.3 Virtual Machine based isolation

Virtual Machines in the simplest sense are software abstractions of real machines. They provide a virtual platform for running tasks. Virtual machines have been employed to provide various features like emulation, optimization, translation, isolation, replication etc [36]. This paper will only consider virtual machines from an isolation perspective. As defined in [36], a *virtual machine can support individual processes or a complete*

[2] Fault Domains as defined in [14] are logically separate portions of an applications address space.

system depending on the abstraction level where virtualization occurs. Some VMs support flexible hardware usage and software isolation, while others translate from one instruction set to another. Based on this observation we can classify Virtual Machine based isolation into 4 categories as:

(a) ***Process Virtual Machines:*** Process Virtual Machines support individual processes or a group of processes and enforce isolation between the processes and operating system environment. Process virtual machines can run processes compiled for the same ISA[3] or for a different ISA as long as the virtual machine runtime supports the translation. Isolation policies are provided by a runtime component which runs the processes under its control. Isolation is guaranteed because the virtual machine runtime does not allow direct access to the resources that the underlying real system provides. Earlier process virtual machines like the Java Virtual Environment (JVM) supported only single processes but research projects like Alta [8] have made it possible to run multiple processes within the same virtual machine. Similarly, dynamic binary optimizers like DynamoRIO [11] which have been extended to provide isolation also fall under this category.

Examples: DynamoRIO with Program Shepherding extensions [10], Java VM [39], MS Common language runtime [40], Alta [8], PeaPod [57].

(b) ***System Virtual Machines (Hypervisor Virtual Machines):*** System virtual machines provide a full replica of the underlying platform and thus enable complete operating systems to be run within it. The virtual machine monitor (also called the hypervisor) runs at the highest privilege level and divides the platforms hardware resources amongst multiple replicated guest systems. All accesses by the guest systems to the underlying hardware resources are then mediated by the virtual machine monitor. This mediation provides the necessary isolation between the virtual machines. System virtual machines can be implemented in a pure-isolation mode [36] in which the virtual systems do not share any resources between themselves or in a sharing-mode in which the VM Monitor multiplexes resources between the machines. Pure-isolation mode virtual machines are as good as separate physical machines. Examples of such systems are the IBM's PR/SM system [37]. Such systems, though highly secure, are not practical for desktop-like environments. Systems like XEN [51] and KVM [53] have commercialized the sharing hypervisor approach in desktop operating systems.

Examples: XEN [51], sHYPE [56], PR/SM [37], Terra [33], Nizza [35], Nexus [32], SVGrid [31], VMware GSX Server [41]

(c) ***Hosted Virtual Machines:*** Hosted Virtual Machines are built on top of an existing operating system called the host. The virtualization layer sits above the regular operating system and makes the virtual machine look like an application process. One can then install complete operating systems called guest operating systems within the host virtual machines. The VM can provide the same instruction set architecture as the host platform or it may also support a completely different instruction set architecture (ISA), like running Windows IA-32 OS on a Mac running on the PowerPC platform. VMware GSX Server is an example where the host ISA and guest ISA are same. Isolation in hosted virtual machines is as good as the isolation provided by the hypervisor approach except that the Virtual Machine Monitor in the case of the hosted VM does not run at the highest privilege. The processes running inside the Virtual machine cannot affect the operation of processes outside the virtual machine. System emulators are also loosely classified under hosted virtual machines [refer section 4.3].

Examples: VMWare Workstation[], Microsoft Virtual PC, Qemu [48], Simics [59]

(d) ***Hardware Virtual Machines:*** Hardware virtual machines are virtual machines built using virtualization primitives provided by the hardware like processor or I/O. The advantage of hardware level virtualization is tremendous performance improvements over the software based approaches and guarantees better isolation between machines. The isolation provided by the hardware assisted

3 ISA stands for *Instruction Set Architecture.* Examples are x86, PPC etc.

virtualization is more secure than that provided by its software counterpart for obvious reasons. This form of virtualization has been exploited in KVM [53] which is based on the virtualization instruction set of the Intel VT-x [60] and AMD-V processors.

Examples: Intel VT-x [59], AMD-V, KVM [53]

3.4 OS-kernel based isolation

OS-Kernel based isolation is the most traditional form of isolation. The operating system kernel has been always regarded as the most trusted component in the system and is thus entrusted with enforcing policies that are required for isolation between applications and between applications and the kernel. For a long time, the isolation guaranteed by the operating system's notion of 'process' was the only isolation that was provided in most mainstream operating systems.

Much research has focused on reducing the size of the kernel because a large kernel implies a larger TCB and hence a larger set of security problems. These efforts have resulted in the Microkernel [42] and Exokernel [43] based operating system kernels as opposed to the traditional monolithic kernels. While the design philosophy may differ between the types of the kernels, the core requirement of being a secure resource manager is still satisfied by all the types. That is, all kernels account for resources used by the processes, tasks or domain and guarantees isolation between them. Monolithic kernels provide an isolation guarantee by using the Memory Management Unit (MMU) of the processor while the Exokernel provide it by implementing fine access controls on the resource accesses by the applications.

The various kernels differ in their design and their requirements but all of them provide the basic isolation between the applications running on top of them. Thus, this category of operating systems is not further subdivided into the various types of kernels. Instead, the different kernels are the examples of this category.

Examples: Monolithic Kernels, Mach Microkernel [42], Exokernel [43], Hypervisors [51], Singularity [34], Perseus [49]

3.5 Hardware-based isolation

Isolation guaranteed by way of hardware controls is hardware-based isolation. This is the strongest form of isolation as it is not easily circumvented by software at runtime. This form of isolation is provided either by the processor or by special devices which work in conjunction with the processor. Most of the processors provide a Memory Management Unit (MMU) which helps in assigning different virtual spaces to different processes and thus provides isolation between them. Similarly, IOMMUs, IO Memory Management Units, are hardware devices that translate a device DMA addresses to physical addresses. As stated in [45], an isolation capable IOMMU restricts a device so that it can only access parts of memory it has been explicitly granted access to. IOMMUs increase system availability and reliability by preventing malicious devices from performing arbitrary DMAs. As stated in [45] [46], operating systems can utilize IOMMUs to isolate device drivers; hypervisors utilize IOMMUs to grant secure direct hardware access to virtual machines. ARM TrustZone [47], and Legba [55] are two other examples in this category.

Examples: MMU, Calgary IOMMU [45], DART IOMMU [46], ARM TrustZone [47], Legba [55]

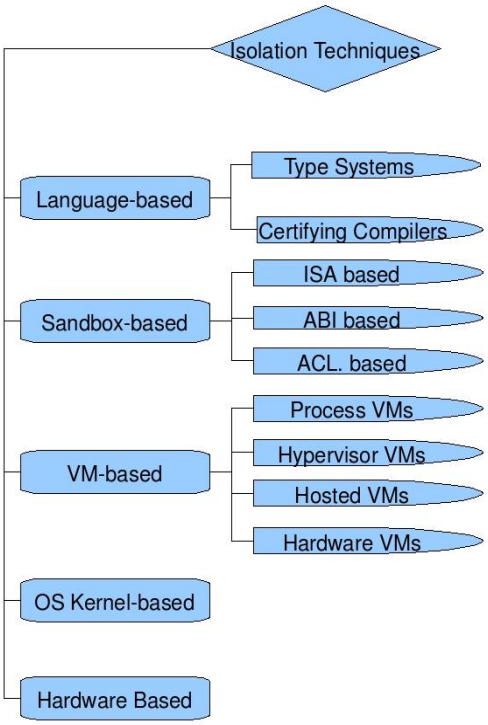

Illustration 1: Taxonomy of isolation techniques

4. A note on the taxonomy

The taxonomy has a few categories which seem to overlap in their definitions. But there are few subtle differences which justify their separate existence. This section describes the subtle differences which distinguish the categories in the taxonomy.

4.1 Hypervisors vs. OS kernels

At first glance, hypervisors or virtual machine monitors don't look any different than OS-kernels. They especially bear a striking similarity to the concept of microkernels. As suggested in [47], VMMs were born out of a necessity to improve system utilization by facilitating time-sharing of machines. The time-sharing aspect also meant that multiple users may own different virtual machines and thus strong isolation guarantees were essential. Microkernels on the other hand were born out of a desire to create small OS kernels which would enable easier validation and porting. Other kernels like Exokernel [43] were born out of the need to enable application level resource control. Thus, as far as the isolation problem is concerned, the VMM or the hypervisor is the only technique that handled the isolation problem. Other kernels also provide isolation, but it is not their major requirement as it is for the hypervisor. It was thus deemed inappropriate to classify hypervisors as just another kernel.

4.2 Virtual Machines vs. Sandboxes

Virtual machines have been loosely referred to as sandboxes in some literature. They can be loosely referred to as sandboxes as they provide a confined execution environment. But the striking difference between the sandbox category presented in the taxonomy and the virtual machines is that the sandbox provides a confined execution for untrusted code on the *same machine*. Virtual machines provide a completely different machine (albeit a virtual machine) for code to execute. Thus, even though they provide a confined execution environment, they are stronger in their isolation guarantees than simple sandbox techniques. Sandboxes are essentially additional patches on top of existing systems to separate trusted and untrusted code.

4.3 Where do System Emulators fit?

System emulators provide a complete software emulated processor. Traditional emulators execute every instruction in software and thus are very slow in their performance. They have nevertheless been used extensively for system testing, debugging and educational purposes. The main difference between a VMM and an emulator is that a VMM executes all the instructions directly on the underlying hardware instead of emulating. Thus VMMs are more practical techniques of isolation. Recently emulators like Simics [59] and Qemu [48] have started supporting a virtualized mode of execution where they try to behave like a VMM instead of a full system emulator. System emulators can thus be considered to be a part of Hosted Virtual Machines.

5. Survey of systems

This section surveys and categorizes 31 systems built using the above techniques or a combination of above techniques to provide isolation in various environments. There is a bias towards considering the security aspects of the system and other features are intentionally omitted. A tabular format is chosen over a more verbose format as the number of systems is very large. The tabular format gives a quick overview of the critical features of each system and provides pointers for further exploration.

For each surveyed system, the isolation mechanisms are listed as per the taxonomy along with a brief description. Systems are compared as per the terminology developed in Section 2. The column 'Tasks' lists the tasks that are being *protected* or *protected from* in the system. 'Protection Domain' (aka the container of tasks and resources) is the outermost protection boundary that is implemented by the system. Note that recursive protection domains are not mentioned. 'TCB' comprises of the minimal set of trusted components on which the security of the system relies. 'Policies' list the policy mechanism of the system. 'Year published' notes the year of publication of the paper describing the system.

Table of Systems

#	System	Isolation Mechanism Used	Brief Description	Tasks	Protection Domain (Container)	TCB	Policies	Year Pub.
1	SPIN [6]	Type System	Uses language features of Modula-3 to enforce boundaries and ensure isolation between code.	Kernel Extensions	OS	OS Core Services of memory and processor + External Modula-3 Type Checker	Static policies enforced by language type safety	1995
2	j-Kernel [7]	Type System + certifying compiler + Microkernel	Uses language features of java to provide multiple protection domains over a single JVM. Sharing between tasks is enabled by sharing capability objects.	Java Programs, Servlets	Java Virtual Machine	JVM + j-kernel library + Java interpreter and compiler	Static policies enforced by the language + policies specified by programmer	1998
3	Program Shepherding [10]	ISA-based sandboxing	Monitors control flow transfers in a process dynamically to enforce security policies. Existing process binaries do not require any changes.	OS Process	RIO Framework	RIO framework + Operating System	Statically specified within a policy file	2002
4	Janus [12]	ABI-based sandboxing	Monitors system call activity and applies policy restrictions to prevent execution of dangerous system calls.	OS Process	JANUS Framework	Janus Framework + Operating System	Statically specified within a policy file	1996
5	Sub Operating Systems [15]	ACL-based sandboxing	Tags each active data object like JavaScript, word files etc. with a different use rid to implement finer permissions on an object basis. This effectively creates a sandbox for active content.	OS Process	Operating System	Operating System + SubOS extensions in application.	Static or dynamic. Depends on how the application chooses to implement.	2000

6	PittSFIeld [30]	ISA-based sandboxing	PittSFIeld enforces security policies in CISC architectures by constraining memory accesses and control flow in untrusted binary code. The idea is to make sure that data and code accesses are all in safe regions.	OS Process	OS process with a reference monitor	Operating System + Binary Rewriter	Static policies as defined in the binary rewriter	2006
7	Terra [33]	Hypervisor Virtual Machine based	Terra is a flexible architecture for Trusted Computing which allows applications with varying security requirements to run simultaneously. Terra uses a Trusted Virtual Machine Monitor, a hypervisor, to partition the platform into multiple isolated virtual machines. All applications are thus completely isolated.	Virtual Machines	Terra Hypervisor Framework	Trusted Virtual Machine Monitor + TPM hardware	Static policies as enforced by hypervisor	2003
8	Nexus [32]	Microkernel - based + Processor based hardware isolation	Nexus is a trustworthy OS design which uses the Trusted Platform Module for trustworthy computing. Applications are run in isolated protected domains and secure memory regions are provided for storing sensitive data.	Isolated Protection Domains	Microkernel Operating System	Microkernel + TPM Hardware	Decentralized, credentials-based authorization using the Nexus Authorization Logic, which encompasses certificates attesting to provenance analysis, or rewriting as a bases to trust a components claims and requests.	2006
9	Singularity [34]	Type Systems + Certifying Compilers + Microkernel	Singularity is a microkernel-based OS which uses language features to provide memory safety and does not depend on hardware MMUs. The basic unit of isolation in singularity is called SIP (Software Isolated Process) which uses type and memory safety features of *Sing#* to create closed and verifiable spaces for code. The communication between SIPs happens via contract channels.	Software Isolated Processes (SIPs)	Singularity Kernel	Singularity Kernel + Sing# Language Compilers + Runtime	Static policies as offered by type-safe sing# and the singularity kernel	2007

10	Nizza [35]	Microkernel based isolation + Sharing Virtual Machine Based Isolation + Language based isolation	Nizza is a secure system architecture that promises a smaller TCB. Nizza supports legacy applications by way of language based VM's or paravirtualized VMs or platform emulating VMs. Isolation is Nizza is provided by a lightweight L4 microkernel. The kernel provides fine-grained protection between the domains.	Either a language based VM or a paravirtualized VM or a platform emulating VM	L4 Microkernel	Fiasco (L4) Microkernel + Secure Platform Layer (Loader + Trusted GUI etc)	Static policies as enforced by the microkernel interface	2005
11	Secure Virtual Enclaves [52]	ACL-based sandboxing (but over a network)	A secure virtual enclave is a collaboration infrastructure which allows multiple organizations to share information with each other but still maintaining local administrative control over their own data. SVE extends ACL based sandboxing over a network.	Operating System Processes	SVE Middleware	SVE Middleware + Operating Systems	Static policies specified in different enclaves	2000
12	XEN on HVM processors [51]	Hypervisor virtual machine based isolation + Hardware Virtual Machine	XEN is a paravirtualized virtual machine architecture and supports virtual domains on top of a thin hypervisor layer. Virtual machines running on top of XEN provide very strong isolation. XEN runs drivers within a virtual domain which provides additional isolation against driver faults.	Paravirtualized kernels or Unmodified kernels running on HVM enabled processors	XEN Hypervisor	XEN Hypervisor + XEN Domain0	Static policies as enforced by the microkernel interface	2003
13	KVM [53]	Hardware VM based isolation + OS based isolation	KVM is an extension to the standard Linux kernel to provide virtualization using hardware VM support. KVM supports running standard Linux processes as well as virtual machines over the standard Linux kernel. The isolation is provided by the hardware and the KVM module in the kernel.	Standard OS processes or virtual machines	KVM Module in kernel + Linux Kernel	KVM module	Static policies as enforced by hardware virtualization + KVM module	2006
14	Denali [54]	Hypervisor VM based isolation	Denali is paravirtualized VM architecture. It uses a thin hypervisor layer to multiplex different VMs' running on top of it and provides full isolation between the VMs'. The applications running on top of Denali are compiled with a guest OS library which provides an abstraction for the available resources.	Virtual Machines	Denali VMM	Denali VMM	Static policies as enforced by the VMM	2001

15	VMware Workstation [41]	Hosted VM based isolation	Exports a full virtual machine as an application level process and allows installation of complete operating systems in the virtual machines. The process running the VM is completely isolated from the regular application processes.	Complete operating system running in the VM	Virtual Machine Monitor	Virtual Machine Monitor + Host Operating System	Static policies as enforced by the VMM	late 1990's
16	Legba [55]	Hardware based isolation	Legba is fine-grained memory protection architecture that enables strong isolation. It enables isolation by introducing object tagging to cache lines and providing protected procedure calls.	OS Processes	Legba enabled TLB architecture	Hardware + OS using the hardware features	Static policies as implemented by the OS using the hardware	2003
17	PeaPod [57]	ACL based sandboxing + ABI based sandboxing +Process VMs	Provides two key sandboxing abstractions: Process Domain (POD) and Process Encapsulation and Abstraction (PEA). PODs provide applications a virtualized view of the underlying OS and PEAs use system call interposition techniques to enforce restrictions on process restrictions. Together, the POD and PEA provide strong isolation between untrusted application processes and allow fine grained specification of policies on a per-application basis.	OS Processes	PeaPod Virtualization layer	PeaPod layer + OS	Static fine-grained policies as specified in configuration	2007
18	SVGrid [31]	Hypervisor based Virtual machine + ACL based sandboxing	SVGrid is a secure virtual grid environment to protect grid computers filesystem and networks from malicious code. SVGrid is based on XEN. All grid applications are run inside a Grid Virtual Machine (GVM) and all accesses to resources from GVMs are redirected to a Monitor Virtual Machine where access policies are applied.	XEN VM	Monitor Virtual Machine + Xen Hypervisor (Xen Domain0)	XEN Hypervisor + Monitoring VM (domain 0)	Static access policies as specified in a access file	2005
19	Consh[20]	ACL based sandboxing + ABI based sandboxing	Consh provides a semi-virtualized view of the filesystem and network to an application so that untrusted applications can run without comprising local resources. Consh also provides fine-grained protection to protect local system resources. It is based on Janus [12].	OS Process	Consh Framework	OS kernel + Consh Framework consisting of Janus and virtualization code	Static policies specified in configuration file	1998

20	SubDomain [19]	ACL based sandboxing	SubDomains is a kernel extension designed to provide least privilege confinement to Untrusted programs. It allows an administrator to specify the domain of activities the program can perform by listing the files the program may access. It also allows subprocesses (child processes) to be assigned separate privileges.	Processes and Sub-Processes (that is portions of a process)	Operating System	OS Kernel + SubDomain Kernel Extensions + Application code calling the SubDomains APIs	Static Policies as specified in configuration files	2000
21	SLIC[25]	ABI based sandboxing	SLIC is an extension system which uses the technique of interposition to insert trusted extension code to existing operating systems. These extensions enable existing OSes to provide tighter isolation environments for executing untrusted binaries.	Regular OS processes	Operating System	SLIC Extensions + Operating System	Static policies as applied by the SLIC extension layer	1998
22	TRON[17]	ACL based sandboxing	Process-level discretionary access control system. Allows users to specify capabilities for a process's access to individual files and directories. The enforcement is done by kernel wrappers.	OS Process	Operating System	OS kernel + Application	Static or dynamic. Left to the discretion of the process.	1995
23	MAPBox [13]	ABI based sandboxing	Classifies applications into classes according to behavior and provides pre-configured sandboxes for each class. Its call interception and policy enforcement mechanism are similar to Janus [12]	OS Process	MAPBox Framework	MAPBox Framework + Operating System	Statically specified within policy files.	2000
24	Chakravyuha [16]	ACL based sandboxing	Uses a Resource Capability List (RCL) to specify permissions and resources accessed by untrusted code. The RCL is attested by a third party. Clients enforce the RCL that is received with the code thus providing a sandbox around the resources	OS processes, active data like applets, scripts	Chakravyuha Framework	Chakravyuha Framework on client + RCL attester + Operating System	Statically specified within RCL files	1997
25	Alta [8]	Process Virtual Machine based isolation + Language based isolation	Alta is an operating system supporting nested processes within a Java Virtual Machine. The language features of java along with the VM provide isolation.	Java Processes	Alta Operating System	JVM	Static policies as provided by the language and JVM	1999
26	One way Isolation [18]	ACL-based sandboxing	Processes executing under this technique are allowed to make reads but their writes are redirected to a different area. This applies to filesystem and network. This creates a very simple sandbox and prevents malicious processes from modifying system data.	OS Processes	Operating System	Operating System Kernel + Isolation File System + Policy Enforcement Engine	Static policies	2005

27	FreeBSD Jails [21]	ACL based sandboxing	FreeBSD Jails allow partitioning of the OS into virtual environments with each environment supporting processes, file systems and network resources. The jail provides a restrictive environment for running untrusted applications.	OS Processes	Virtual Jail Environment	Operating System kernel (with the jail extensions)	Static Policies as defined by standard UNIX semantics	2000
28	Fine grained protection domains [28]	ABI based sandboxing	Combines benefits of both kernel level and user level sandboxes by placing a reference monitor in the same process address space as the sandboxed applications. The protection is provided at memory page level. The reference monitor intercepts system calls made by the application and can enforce its policies.	Operating System Process	OS Kernel	OS Kernel + Kernel Extensions to implement Fine Grained Protection + OS Loader	Static policies identified by programmer	2003
29	Flexibly controlling downloaded executable content [26]	ACL based sandboxing + Type Systems	Describes an elaborate architecture for controlling downloaded executable content which provides for authentication of remote sources, determining access control rights based on source and application and enforcement mechanisms for policies.	Active content like scripts, applets and operating system processes	Content Protocol Framework	OS Kernel + Trusted Browser + Security Managers	Static policies specified in ACLs	1996
30	Deeds [29]	ACL based sandboxing + Type Systems	Deeds implements a history based access control for mobile code. It maintains a selective history of accesses made by programs and uses this to discriminate between safe and unsafe programs.	Active content like scripts, applets and operating system processes	Deeds Framework	OS Kernel + Deeds Framework	Dynamic Policies because the system supposedly learns from histories and adapts its policies to provide security and ease of use.	1998
31	Perseus [49]	Microkernel OS based isolation + hardware based isolation	Perseus is a security framework for trustworthy computing. It is based upon the Fiasco microkernel and uses services of the trusted platform module to guarantee security. Isolation is provided by the microkernel using hardware assisted isolation.	Paravirtualized VM or OS Processes	Microkernel Secure Platform	Secure Platform + TPM Hardware	Static policies as enforced by the secure platform layer	2001

6. Observations

The taxonomy and the list of surveyed systems present a clear view of the plethora of research that has happened in isolation security. There are several observations that can be made about the evolution of isolation.

Observation 1: The current trend in systems design is to combine many isolation techniques into the complete system as can be seen in projects like Singularity [34] and Nizza [35]. This trend is clearly justified because of the evolving nature of the threats and threat vectors. It is unlikely that a single isolation technique would be capable of preventing all attacks.

Observation 2: There is a shift towards mandatory access control based systems from discretionary access control based systems. This is clearly visible due to the large number of systems that incorporate virtualization techniques or hardware-based techniques or language-based techniques. Mandatory access control techniques gives little power to the user to subvert a system due to the access mechanisms implicitly built into the system during its construction. For example, using type safe compliers like Java automatically removes buffer overflow vulnerabilities, using virtualization techniques confines program execution to a completely separate machine and thus is inherently stronger than basic protections provided by a process.

*Observation 3: A*ll systems allow some way of specifying static policies for the system. In some cases like language-based systems, the policies are very implicit while in other cases like the sandboxing-based systems the policies are explicitly specified. A system or a technique is more prone to configuration errors when it is explicitly configurable because it requires an intricate understanding of the policies and their dependencies. We, thus, also need systems that learn policies dynamically from the environment in which they are operating.

Observation 4: There is a growing trend towards using virtualization to provide isolation security. We believe that this is due to the fact that virtual machines provide an easy and fast way to configure a very secure environment. Virtual machines provide an easy way to securely wrap (in an attempt to contain) existing applications.

Observation 5: In spite of the work done for isolation in individual systems, there has been little work done for isolation in networked systems. We only found one system, Secure Virtual Enclaves [52], which implemented minimal isolation over a network. For reasons mentioned in section 2 of this paper, isolation in networked systems is becoming a very important challenge today. Networked systems today provide an easy infrastructure for supporting the notion of Virtual Systems [1]. Such virtual systems if deployed would require isolation mechanisms beyond those that are used for individual systems.

7. Summary

We have introduced taxonomy for isolation techniques in individual systems. The taxonomy comprises five major categories: language-based, sandbox-based, VM-based, OS kernel-based and hardware-based. A survey of 31 systems was presented with respect to the taxonomy. We note that the current trend in systems is to use a composition of techniques instead of relying on one technique. Virtualization has been adopted by the systems community as the technique of choice for providing isolation. There is very little work on isolation in networked systems. Next-generation systems must build in *isolation* as a requirement and not as an option.

References

[1]. Anderson, J. Computer security technology planning study. U.S. Air Force Electronic Systems Division Tech. Rep. (Oct. 1972), 73--51.

[2]. Butler W. Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. ACM Transactions on Computer Systems, 10(4):265--310, November 1992

[3]. Fred B. Schneider, Greg Morrisett, Robert Harper. A language-based approach to security. Informatics: 10 Years Back, 10 Years Ahead, Lecture Notes in Computer Science, Vol. 2000, Springer-Verlag, Heidelberg, 86-101.

[4]. George C. Necula. Compiling with Proofs. PhD thesis, School of Computer Science, Carnegie Mellon Univ., Sept. 1998.

[5]. Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A Realistic Typed Assembly Language. In the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software, pages 25-35, Atlanta, GA, USA, May 1999.

[6]. B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System. 15th ACM Symposium on Operating Systems.

[7]. T. von Eicken, C.-C. Chang, G. Czajkowski, C. Hawblitzel, D. Hu, and D. Spoonhower. J-Kernel: A capability-based operating system for Java. In Vitek and Jensen [45], pages 369–393.

[8]. P. A. Tullmann, "The Alta operating system," Master's thesis, University of Utah, Dec. 1999.

[9]. M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In 2nd Symposium on Operating Systems Design and Implementation, pages 213--227, 1996

[10]. Kiriansky, V., Bruening, D., and Amarasinghe, S. P. 2002. Secure Execution via Program Shepherding. In Proceedings of the 11th USENIX Security Symposium (August 05 - 09, 2002). D. Boneh, Ed. USENIX Association, Berkeley, CA, 191-206.

[11]. Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4), December 2000.

[12]. David A. Wagner. *Janus: an approach for confinement of untrusted applications*. Master's thesis, University of California, Berkeley, 1999.. Also available Technical Report CSD-99-1056, UC Berkeley, Computer Science Division. http://www.cs.berkeley.edu/~daw/papers/janus-masters.ps

[13]. Anurag Acharya and Mandar Raje. Mapbox: Using parameterized behavior classes to confine applications. In Proceedings of the 2000 USENIX Security Symposium, pages 1-17, Denver, CO,

August 2000.

[14]. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In Proceedings of the 14th ACM Symposium on Operating Systems Principles, pages 203--216, December 1993.

[15]. Sotiris Ioannidis and Steven M. Bellovin. Sub-Operating Systems: A New Approach to Application Security. Technical Report MS-CIS-01-06, University of Pennsylvania, February 2000.

[16]. A. Dan, A. Mohindra, R. Ramaswami, and D. Sitaram. ChakraVyuha: A sandbox operating system for the controlled execution of alien code. Technical Report 20742, IBM T. J. Watson Research Center, 1997.

[17]. Berman, A., Bourassa, V., and Selberg, E. 1995. TRON: process-specific file protection for the UNIX operating system. In Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings (New Orleans, Louisiana, January 16 - 20, 1995). USENIX Association, Berkeley, CA, 14-14.

[18]. W. Sun, Z. Liang, R. Sekar, and V. N. Venkatakrishnan. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In Proceedings of the 12th ISOC Symposium on Network and Distributed Systems Security (SNDSS), pages 265–278, February 2005.

[19]. Crispin Cowan, Steve Beattie, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Parsimonious Server Security. In USENIX 14th Systems Administration Conference (LISA), New Orleans, LA, December 2000.

[20]. A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations. Usenix, Dec 1998.

[21]. P.-H. Kamp and R. N. Watson. Jails: Confining the omnipotent root. In Proceedings of the Second International System Administration and Networking Conference (SANE), Maastricht, The Netherlands, May 2000.

[22]. L. D. Stein, "SBOX: Put CGI scripts in a box," in Proc. 1999. http://stein.cshl.org/software/sbox/

[23]. R. Balzer and N. Goldman. Mediating connectors: A nonbypassable process wrapping technology. In Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, June 1999.

[24]. T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 1999.

[25]. D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In Proceedings of the 1998 USENIX Annual Technical Conference, pages 39–52, June 1998.

[26]. T. Jaeger, A. D. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In Proceedings of the 1996 USENIX Security Symposium, pages 131–148, San Jose, Ca., 1996.

[27]. K. M.Walker, D. F. Stern, L. Badger, K. A. Oosendorp, M. J. Petkac and D. L. Sherman. Confining root programs with domain and type enforcement. In Proceedings of the 1996 USENIX Security

Symposium, pages 21–36, July 1996.

[28]. T. Shinagawa, K. Kono, and T. Masuda. Flexible and efficient sandboxing based on fine-grained protection domains. In Proceedings of the International Symposium on Software Security, pages 172--184, February 2003.

[29]. Edjlali, Guy, Anurag Acharya, and Vipin Chaudhary, "History-based Access-control for Mobile Code." To appear in Proceedings of the Fifth ACM Conference on Computer and Communications Security. San Francisco, CA, USA. November 1998.

[30]. McCamant, S. and Morrisett, G. 2006. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15* (Vancouver, B.C., Canada, July 31 - August 04, 2006). USENIX Security Symposium. USENIX Association, Berkeley, CA

[31]. Zhao, X., Borders, K., and Prakash, A. 2005. SVGrid: a secure virtual environment for untrusted grid applications. In Proceedings of the 3rd international Workshop on Middleware for Grid Computing (Grenoble, France, November 28 - December 02, 2005).

[32]. Emin Gun Sirer. Nexus: A New Operating System for Trustworthy Computing. TRUST (Team for Research in Ubiquitous Secure Technology) Winter Meeting, Washington, DC, January 2006. (Talk)

[33]. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum M., and Boneh, D. Terra: A virtual machine-based platform for trusted computing. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003).

[34]. Galen Hunt and James Larus. Singularity: Rethinking the Software Stack. Operating Systems Review, Vol. 41, Iss. 2, pp. 37-49, April 2007. ACM SIGOPS.

[35]. Hartig, H. et. al. The Nizza secure-system architecture. International Conference on Collaborative Computing: Networking, Applications and Worksharing, pp. 10, 21 Dec 2005.

[36]. James E. Smith, Ravi Nair, "The Architecture of Virtual Machines," Computer, vol.38, no.5, pp. 32-38, May, 2005

[37]. Certification Report for Processor Resource/System Manager (PR/SM) for the IBM eServer zSeries 900, BSI-DSZ-CC-0179-2003, 27 February 2003, Bundesamt fur Sicherheit in der Informationstechnik: Bonn, Germany. URL: http://www.commoncriteriaportal.org/public/files/epfiles/0179a.pdf

[38]. V. Bala, E. Duesterwald, and S. Banerjia, “Dynamo: A Transparent Dynamic Optimization System,” Proc. ACM SIGPLAN 2000 Conf. Programming Language Design and Implementation, ACM Press, 2000, pp. 1-12.

[39]. T. Lindholm and F. Yellin, The Java Virtual Machine Specification, 2nd ed., Addison-Wesley, 1999.

[40]. D. Box, Essential .NET, Volume 1: The Common Language Runtime, Addison-Wesley, 2002.

[41]. J. Sugerman, G. Venkitachalam, and B-H. Lim, “Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor,” Proc. General Track: 2001 Usenix Ann. Technical Conf., Usenix Assoc. 2001, pp.1-14.

[42]. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for UNIX development. Proc. Summer 1986 USENIX Conference, pages 93–112, July 1986.

[43]. Kaashoek, M. F., Engler, D. R., Ganger, G. R., Briceño, H. M., Hunt, R., Mazières, D., Pinckney, T., Grimm, R., Jannotti, J., and Mackenzie, K. 1997. Application performance and flexibility on exokernel systems. In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (Saint Malo, France, October 05 - 08, 1997). W. M. Waite, Ed. SOSP '97. ACM, New York, NY, 52-65.

[44]. Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski (2007-06-27). "Price of Safety: Evaluating IOMMU Performance" (PDF). Proceedings of the Linux Symposium 2007, Ottawa, Ontario, Canada: IBM Research

[45]. Utilizing IOMMUs for Virtualization in Linux and Xen, by M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L. Van Doorn, A. Mallick, J. Nakajima, and E. Wahlig, in Proceedings of the 2006 Ottawa Linux Symposium (OLS), 2006.

[46]. T. Halfhill. ARM Dons Armor: TrustZone Security Extensions Strengthen ARMv6 Architecture. Microprocessor Report, 2003. Document available at the URL: http://www.arm.com/miscPDFs/4136.pdf

[47]. Heiser, G., Uhlig, V., and LeVasseur, J. 2006. Are virtual-machine monitors microkernels done right? SIGOPS Oper. Syst. Rev. 40, 1 (Jan. 2006), 95-99.

[48]. Fabrice, B. QEMU, a Fast and Portable Dynamic Translator, USENIX 2005 Annual Technical Conference, FREENIX Track

[49]. B. Pfitzmann, J. Riordan, Christian Stüble, M. Waidner, A. Weber: The PERSEUS System Architecture; Verlässliche Informationssysteme (VIS) '01, DuD Fachbeiträge, Vieweg Verlag, pp. 1-18, Kiel, 2001.

[50]. Tullman, P. and Lepreau, J. 1998. Nested Java processes: OS structure for mobile code. In Proceedings of the 8th ACM SIGOPS European Workshop on Support For Composing Distributed Applications (Sintra, Portugal)

[51]. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. 2003. Xen and the art of virtualization. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (Bolton Landing, NY, USA, October 19 - 22, 2003)

[52]. Shands, D. et. al. Secure virtual enclaves: Supporting coalition use of distributed application technologies. ACM Trans. Inf. Syst. Secur. 4, 2 (May. 2001), 103-133.

[53]. KVM White Paper. URL : http://www.qumranet.com/art_images/files/8/KVM_Whitepaper.pdf

[54]. Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel, Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI 2002), Boston, MA, December 2002.

[55]. Adam Wiggins, Simon Winwood. Legba: Fast hardware support for fine grained protection. In Proceedings of the 8th Australia-Pacific Computer Systems Architecture Conference (ACSAC'2003)

[56]. R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, S. Berger: sHype: Secure Hypervisor Approach to Trusted Virtualized Systems. IBM Research report RC23511.

[57]. Shaya Potter, Jason Nieh, and Matt Selsky. Secure Isolation of Untrusted Legacy Applications. Proceedings of the 21st Large Installation System Administration Conference (LISA '07), pp 117-130.

[58]. Erlingsson, U.; Schneider, F.B., "IRM enforcement of Java stack inspection," Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on , vol., no., pp.246-255, 2000

[59]. Simics. URL : http://www.virtutech.com/

[60]. Intel Virtualization Technology: Hardware Support for efficient processor virtualization. URL: ftp://download.intel.com/technology/itj/2006/v10i3/v10-i3-art01.pdf

[61]. B. Clifford Neuman, The Virtual System Model: A Scalable Approach to Organizing Large Systems, Ph.D. Thesis, University of Washington, Department of Computer Science and Engineering Technical Report 92-06-04, June 1992