Protection Mechanisms for Application Service Hosting Platforms

Xuxian Jiang, Dongyan Xu
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA
{jiangx, dxu}@cs.purdue.edu

Rudolf Eigenmann
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907, USA
eigenman@ecn.purdue.edu

Abstract

The Application Service Hosting Platform (ASHP) has recently received tremendous attention from both industry and academia. An ASHP provides a shared high-performance infrastructure to host different Application Services (AS), outsourced by Application Service Providers (ASP). In this paper, we focus on the protection of ASHP, which has inherent requirement of sharing, openness, and mutual isolation. Different from a dedicated server platform, which is analogous with a private house, an ASHP is like an apartment building, involving the 'host' - the ASHP infrastructure and the 'tenants' - the AS. Strong protection and isolation must be provided between the host and the tenants, as well as between different tenants.

Unfortunately, traditional OS architecture and mechanisms are not adequate to provide strong ASHP protection. In this paper, we first make the case for a new OS architecture based on the virtual OS technology. We then present three protection mechanisms we have developed in SODA, our ASHP architecture. The mechanisms include: (1) resource isolation between AS, (2) virtual switching and firewalling between AS, and (3) kernelized intrusion detection and logging for each AS. For (3), we have developed a system called Kernort inside the virtual OS kernel. Kernort detects network intrusions in real-time and logs AS activities even when the AS has been compromised. Moreover, for the privacy of AS, logs are encrypted by Kernort so that the 'landlord' (namely ASHP owner) cannot view them without authorization. We are applying SODA to iShare*, an Internet-based distributed resource sharing platform.*

1 Introduction

An Application Service Hosting Platform (ASHP) consists of high performance hosts connected by high speed networks. The ASHP creates a shared platform for the hosting of multiple Application Services (AS), which are *outsourced* by their respective Application Service Providers (ASP). Examples of AS include e-laboratories, e-campaign, and on-line conference management. In an ASHP, AS are installed and invoked on-demand at the requests of their ASP, and ASHP resources are dynamically allocated to the AS according to their service load. Therefore, ASHP reflects the vision of *utility computing*: computational resources are supplied on-demand, and turned off when no longer needed. ASHP have recently drawn tremendous attention from both industry (eg. IBM, HP, and VERITAS) and academia (eg. Xen [10] and SHoP [21]).

Current research in ASHP mainly focuses on dynamic resource provisioning. Much fewer efforts have been devoted to the critical problem of *ASHP security and protection*. In this paper, we show that ASHP protection poses new challenges. The new challenges are due to an ASHP's inherent requirement of *openness*, *sharing* and *mutual isolation*. Unlike a dedicated server platform which is analogous with a private house, an ASHP is like an apartment building, involving both the 'host' - the ASHP infrastructure and the 'tenants' - the AS. From a tenant's point of view, the ASHP should be a safe place to 'live' with privacy and isolation. From the landlord's point of view, the tenants should not do any damage to the 'property' and should not bother other tenants. More specifically, we consider the following requirements:

- *ASHP host protection:* The ASHP hosts should be protected from attacks from outside the ASHP. If one ASHP host is attacked, *all* AS residing in it will be affected.
- *Confinement of AS:* The ASHP should prevent any damage by its tenants - the AS. It is desirable that the activities of each AS are strictly confined within their allocated space in the ASHP.
- *Isolation between AS:* It is equally important that the ASHP should provide isolation *between* the tenants:

Each AS should run as if it is in a dedicated environment. Between AS sharing the same ASHP host, isolation is desirable with respect to (1) *administration*: an ASP should have administrator privilege, but *only* within its own AS; (2) *fault and attack*: a crash or security breach of one AS should not affect other AS; and (3) *resources*: each AS should be guaranteed the 'slice' of ASHP host allocated to it, and it should not be able to launch a local DoS attack upon other AS.

- *Controlled communication between AS:* Sometimes neighbors do talk to each other: an AS may need to communicate with another AS to form a *composite* service. Therefore, the ASHP should enable controlled inter-AS communications.

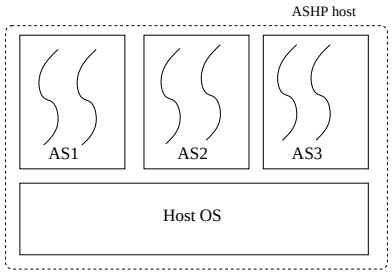
- *Untamperable intrusion detection and logging:* In each AS, intrusion detection and logging functions should be provided and should be untamperable. Furthermore, a log may contain sensitive information such as customer information. The ASHP should assure that the logs are *not* viewable by the landlord (i.e. ASHP owner), except during post-attack forensic analysis with authorization.

Unfortunately, the traditional *single-level* OS architecture, in which one underlying *host OS* supports *all* AS running on top of it, is not adequate to meet the above requirements (to be discussed in Section 2). In this paper, we present a new *two-level* ASHP architecture, based on the *virtual OS* technology. In the two-level architecture, each AS runs on top of a virtual *guest OS*; while the guest OS runs on top of the *host OS*.

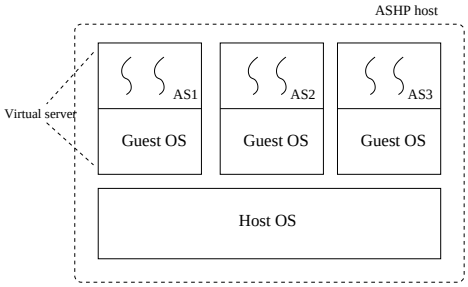
Although the virtual OS technology is not new [10, 13, 23], we have developed novel mechanisms for ASHP protection. Most notably, we have implemented (1) *resource isolation* (CPU, bandwidth, and memory) between AS, (2) *virtual switching and firewalling* between AS inside the same ASHP host, and (3) *Kernort*, a kernelized intrusion detection and logging system for each AS. The rest of the paper is organized as follows: Section 2 compares the single-level and two-level ASHP architectures. Section 3 presents an overview of our two-level ASHP architecture called SODA. Section 4 describes the novel ASHP protection mechanisms in SODA. Section 5 presents performance evaluation results. Section 6 outlines the application of SODA to *iShare*, an Internet-based resource sharing platform. Section 7 compares our work with related work. Finally, Section 8 concludes this paper.

2 Comparison of ASHP Architectures

The two different ASHP architectures are shown in Figure 1: Figure 1(a) shows the traditional *single-level* ASHP architecture; while Figure 1(b) shows the *two-level* architecture based on the 'guest OS/host OS' structure.



(a) Single-level architecture (host OS)



(b) Two-level architecture (guest OS/host OS)

Figure 1. Two different ASHP architectures (only showing one ASHP host)

In both architectures, multiple AS are hosted in one ASHP host. In the single-level architecture, all AS run directly on top of the host OS. In the two-level architecture, each AS runs within a *virtual server*, which is physically a 'slice' of the ASHP host. Inside the virtual server, the AS software runs on top of a virtual *guest OS*.

We argue that the traditional single-level architecture is not adequate for ASHP protection, in the following aspects:

- *Administration isolation*: It is desirable that each ASP has full administrator privilege only *within* the corresponding AS, so that the ASP can perform AS-specific management tasks such as data/software upgrade. However, if the administrator privileges of all ASP are at the same (host OS) level, access control will become complicated and may lead to security holes.

- *Installation isolation*: Different AS may require the same library, but of *different* versions, or their service daemons may require the *same* port binding. In the single-level architecture, such conflicts are difficult to resolve and can potentially lead to local DoS attacks

(by port exhaustion, for example) between AS. On the other hand, the two-level architecture naturally eliminates such conflicts.

- *Fault/attack isolation*: If all AS run at the same host OS level, any fault or security breach in one AS will affect the host OS and therefore other AS. For example, *ghttpd* [19] is a light-weight web server run by the root. However, one known attack upon *ghttpd* is: a malicious packet is sent as an HTTP request, causing buffer overflow to bind a shell on a certain port. Then the attacker can remotely log in using the port, and run a remote shell! On the other hand, in the two-level architecture, since the root that runs *ghttpd* is the root of the *guest OS*, not the host OS, the attack will *not* affect the host OS as well as other AS.

- *Crash recovery and forensics*: In the single-level architecture, to recover from an attack/crash of an AS, the entire ASHP host will have to be rebooted. As a result, other AS in the same ASHP host will be affected. On the other hand, in the two-level architecture, the recovery of one AS has *no* impact on the normal operations of other AS: the ASHP administrator can simply restart the virtual server; and the image of the attacked virtual server is dumped to a file and sent to an off-line site for forensic analysis.

In the two-level architecture, due to the 'guest OS/host OS' indirection, AS performance slow-down is inevitable. In other words, to achieve the same AS service quality as in the single-level architecture, the two-level architecture requires more CPU capacity. With the rapid advances in processor speed, such cost is getting affordable.

3 SODA: Our Two-Level ASHP Architecture

We have developed a two-level ASHP architecture called SODA [16], a Service-On-Demand Architecture. Details about the non-security aspects of SODA can be found in [16]. Currently, SODA adopts Linux as the host OS of physical SODA hosts. For the guest OS running in each virtual server, we leverage and extend an open-source virtual OS project called UML [13], or User-Mode Linux. Unlike other virtual machine techniques such as VMWare [6], a UML runs directly in the unmodified *user space* of the host OS; and processes within a UML will be executed in the virtual server exactly the same way as they would be executed in a native Linux machine. A special thread is created to intercept the system calls made by all process threads of the UML, and redirect them into the guest OS kernel. The following features are enabled by leveraging the 'UML (guest OS)/Linux (host OS)' structure:

- The host OS has a separate *kernel space* from the guest OSes (UMLs), therefore preventing any harm done by individual guest OS to the host OS.

- An IP address is assigned to each virtual server, so that it has full internetworking capability just like a physical server.

- A virtual server can be frozen/restarted without affecting other virtual servers: the images of both the UML and the AS on top of it can be copied to a file, and be conveniently backed up and restarted. Such feature enables easy fault/attack recovery and forensic analysis.

However, current Linux (as host OS) and UML (as guest OS) are not sufficient to achieve SODA security. We have extended both of them by implementing a number of protection mechanisms as described in the next section.

4 ASHP Protection Mechanisms in SODA

In this section, we present three ASHP protection mechanisms in SODA: (1) *resource isolation*, (2) *virtual switching and firewalling*, and (3) *Kernort, a kernelized intrusion detection and logging system*. All mechanisms aim at meeting the ASHP protection requirements in Section 1. (1) and (2) are implemented inside the host OS, while (3) is implemented inside the guest OS.

Although far from being a complete suite of SODA security solutions, these mechanisms form the basis on which more complicated security mechanisms and policies can be implemented.

4.1 Resource Isolation between Virtual Servers

Resource isolation not only provides performance guarantee to the AS running in each virtual server, but also prevents an ill-behaving or malicious AS from launching local DoS attacks upon other AS in the same ASHP host. Currently, our SODA implementation supports CPU, network bandwidth, and memory isolation. SODA resource isolation mechanism has been presented in [16]. A summary is as follows for the completeness of this paper.

- *CPU capacity isolation* is achieved by implementing a coarse-grain CPU proportional sharing scheduler in the Linux host OS. The scheduler enforces the CPU share allocated to each virtual server. The CPU share of a virtual server is decided when the corresponding AS is admitted to the SODA platform. Within one virtual server, all processes bear the same user (AS) id. The host OS CPU scheduler then enforces CPU proportional sharing among all processes based on their user ids.

- *Network bandwidth isolation* is similarly achieved by implementing a traffic controller inside the Linux host OS. The traffic controller enforces the outbound bandwidth share allocated to each virtual server. Recall that each virtual server has its own IP address. The traffic controller achieves bandwidth isolation between virtual servers based on the IP addresses of outgoing packets generated by these virtual servers.

- *Memory isolation*: Memory is critical to the performance of virtual servers (and therefore that of AS). SODA simply leverages the *memory usage limit* feature of UML: the maximum amount of memory available to a virtual server (both AS and guest OS) can be specified as a parameter when UML, the guest OS, is started.

Note that resource isolation only prevents DoS attacks *between* AS. To prevent intra-AS DoS attacks launched by the clients of an AS, other methods (such as client puzzles [9]) still need to be installed as part of the AS software.

4.2 Virtual Switching and Firewalling

SODA uses virtual switching to connect the virtual servers to the outside world as well as between the virtual servers. More importantly, virtual switching achieves strong protection for the SODA host: the physical SODA host itself will have *no* IP address. Therefore, the host OS is totally 'invisible' and therefore un-attackable from the Internet.

To realize the seemingly conflicting goals of virtual server networking and SODA host *in-visibility*, we have implemented a software switch module inside the host OS (Figure 2). This solution is inspired by a real layer-two switch connecting physical NICs, yet the switch itself does not have an IP address. Similarly, we create a software switch in the host OS, connecting multiple virtual NICs of the virtual servers and one physical NIC of the SODA host.

The software switch forwards packets to/from the virtual servers. Furthermore, we enhance the software switch to support firewalling *between the virtual servers*. A physical firewall can protect the SODA host from attacks from the outside, but it *cannot* handle attacks from one virtual server against another virtual server *inside* the same SODA host. Fortunately, our software switch provides an ideal venue to perform inter-virtual-server firewalling: The firewall module is plugged in on the packet path between the virtual servers, enforcing access policies ('who can connect to whom') between the virtual servers. Currently, the firewall implementation in SODA is based on the widely adopted *netfilter/iptables* suite. The software switch can easily accommodate more complicated firewalling rules such as those for *reverse* firewalls.

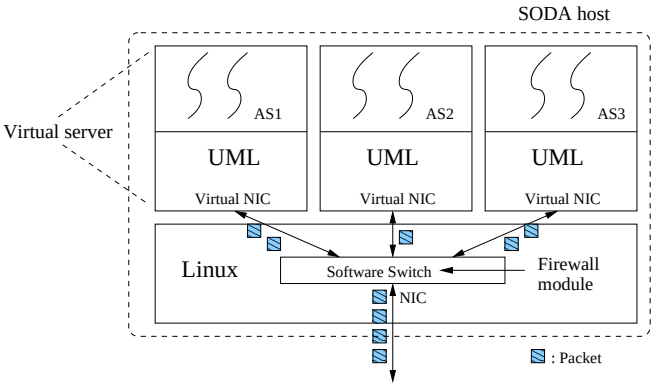


Figure 2. Virtual switching and firewalling: software switch inside the host OS (Linux)

With virtual switching and firewalling, the AS can communicate and collaborate in a secure fashion, making it possible to create composite and value-added application services. On the other hand, it is also possible to enforce communication isolation via the virtual firewall - for example, to prevent two competing AS (which sell the same products, for instance) from attacking each other.

4.3 Kernort: Kernelized Intrusion Detection and Logging

Intrusion detection is important to each AS hosted in SODA. It is interesting to determine where to install intrusion detection systems (IDS) in SODA: If an IDS is installed as an application-level system, it can be easily disabled by the intruder once the system is compromised. If an IDS runs inside the host OS, it will not be able to interpret any *encrypted* traffic which may be from an intruder-initiated *ssh* session, because the decryption will take place in the guest OS, not in host OS. As a result, we choose to install Kernort, our IDS system, inside the guest OS of each virtual server, as shown in Figure 3. To be tamper-free, Kernort is *not* a loadable kernel module. Instead, it is 'hard wired' as part of the guest OS kernel[1] compiled by a trusted *UML factory* (to be described shortly).

Kernort can be regarded as the *'kernelized'* and extended version of snort [4], an open source IDS. Driven by system calls and packet reception, Kernort performs signature based real-time intrusion detection. Since Kernort is located in the data path, one concern is the overhead it adds to the normal virtual server operations. Our measurement results in Section 5 show that for a small set of attack signatures

[1]To make the guest OS kernel un-damageable by its processes, we leverage the 'skas' mode of UML: the UML kernel is in a separate address space from it processes, making the UML kernel totally invisible to its processes.

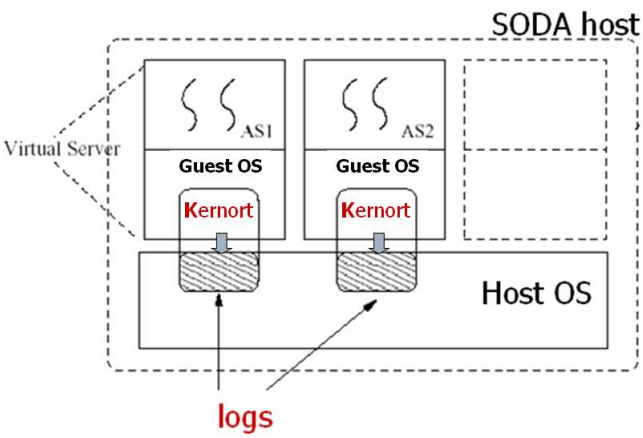


Figure 3. Kernort: the IDS in the guest OS kernel of each virtual server. Logs are generated by Kernort and pushed down to host file system

(we call them the 'top N most wanted'), Kernort incurs very little overhead.

Kernort also performs logging for each AS, which is necessary for off-line intrusion detection (based on a much larger attack signature set) and auditing. However, logging poses the following dilemma: If it is performed inside the guest OS, the log may be tampered with or even erased by an intruder, who tends to do so first thing after breaking into a virtual server. On the other hand, if logging is performed by the host OS, two problems will arise: (1) the privacy of the tenants (AS) is violated, because the landlord can view the AS log and (2) it is difficult or even impossible to log activities that happen *inside* the virtual server.

In Kernort, we adopt a novel strategy to solve the above problem as shown in Figure 3: the log data are *generated* by Kernort in the guest OS kernel, but they are *stored* in the host file system. Kernort is capable of taking snapshots of AS execution, as well as collecting system-wide (within the virtual server) log data such as those from *syslogd* and *klogd*, and verbatim record of user console (local and remote) input. The log data are immediately *pushed down* to the host file system for storage. Recall that the host OS is un-attackable from both inside and outside, and therefore a much safer place to store the log data.

Furthermore, to conserve the privacy of the AS, Kernort will *encrypt* the log data before pushing them down to the host OS. For an AS, the key to encrypt the log data is *generated and compiled* into the UML kernel by a *trusted* authority called the *Trusted UML Factory*, before the AS is created in the SODA host. As shown in Figure 4, the Trusted UML Factory obtains the image of the AS from the ASP, and builds a customized guest OS (UML) kernel with both Kernort and the key compiled in it. The images of the AS and guest OS will then be downloaded to the SODA host. The key is nowhere to be obtained in the SODA host. Instead, it will be kept by the Trusted UML Factory and by the ASP. In the event of a crash or an attack, the key will be used to decrypt the log data for forensic analysis.

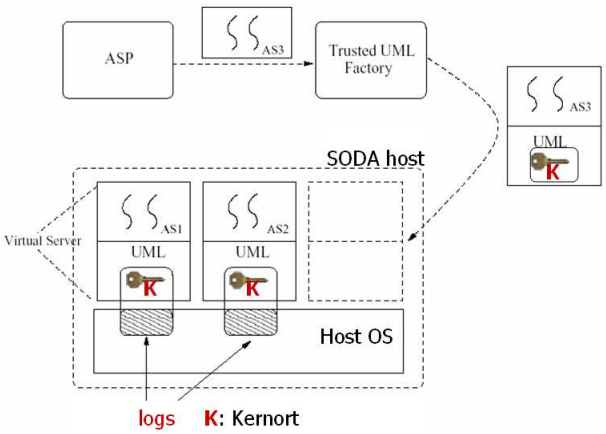


Figure 4. Guest OS with Kernort and log encryption key compiled by Trusted UML Factory; log data are first encrypted before getting to the host file system

5 Experiments and Performance Evaluation

We have deployed a local testbed of SODA to demonstrate and evaluate the protection mechanisms in Section 4. The SODA hosts in the testbed are Dell PowerEdge 2650 servers each with a 2.6GHz Intel Xeon processor and 2GB RAM, connected by 100Mbps LAN. Performance of the resource isolation mechanisms has been presented in [16].

We first demonstrate Kernort in action: we launch a known attack called *Lion Worm* against one virtual server running in a SODA host. When the attack occurs, Kernort issues an intrusion alarm in real-time as shown in Figure 5. As another demonstration, Figure 6 shows a screenshot that looks very much like a regular session. In fact, it is *not*. It is instead a *replay* of an earlier *ssh* session, based on the log recorded by Kernort.

To evaluate the overhead added to each virtual server by Kernort, we have performed a number of experiments using *LMbench* [17], a suite of benchmarks for UNIX system performance comparison. More specifically, we compare the performance of a virtual server with and without Kernort running in its guest OS. The virtual server runs under otherwise the same configuration in the same SODA host. Kernort contains a set of 10 attack signatures. Tables 1-3 show the comparison results: Table 1 shows

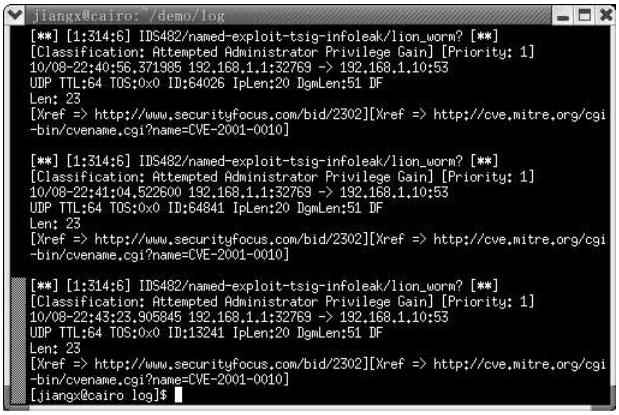


Figure 5. Screenshot of real-time intrusion alert raised by Kernort when the *Lion Worm* attacks a virtual server

process microbenchmarks; Table 2 shows *context switching time* between different number of processes with different working set sizes; and Table 3 shows the *file system and virtual memory system latency*. All results clearly indicate that Kernort in the guest OS incurs very low overhead to the virtual server.

Finally, to evaluate the networking overhead incurred by Kernort, we performed another set of measurements of TCP throughput and latency. Figures 7 and 8 show the comparison results under different TCP send buffer sizes. The throughput and latency penalty incurred by Kernort is low, as indicated by the results. In summary, all our experimental results show that Kernort is an effective and low-overhead mechanism for ASHP protection.

6 Application to iShare

As our ongoing work, we are integrating SODA into a fully decentralized resource sharing platform called *iShare* [2]. Designed as an Internet-sharing middleware, iShare facilitates the supply and request of shared computing resources, including programs and AS dispersed over the Internet. A key idea of the de-central structure is that resource/service providers can post their availability, together with any access rules, on web pages. iShare user nodes will then discover these resources and access them, if they agree with the published rules. iShare presents the discovered resources to end users in the form of *Cyberlaboratories*. For example, the Nanotechnology, Lifesciences, or Parallel Programming Cyberlaboratories include software tools as well as databases that are specific to the respective subjects.

To a resource/service provider or an advanced user, iShare provides four basic functions: The *iPublish* function allows the provider to post resource availability and define

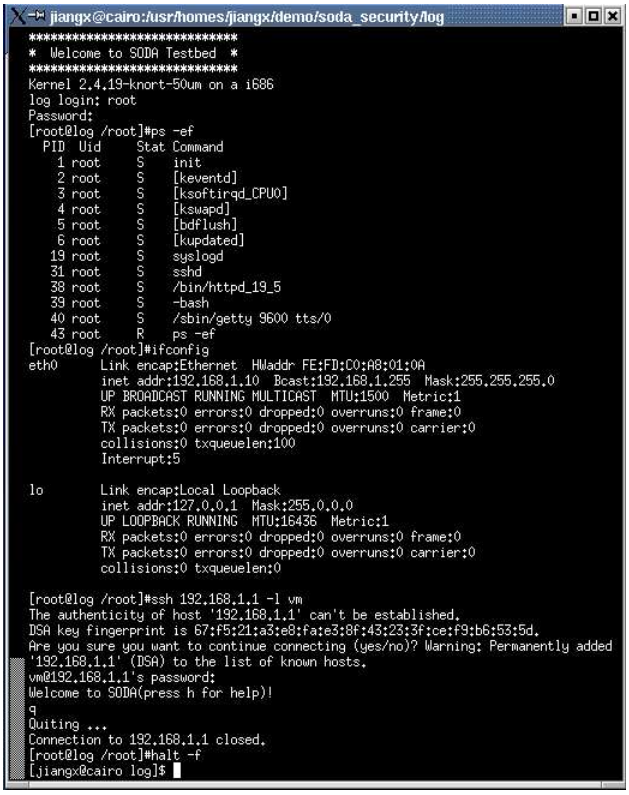


Figure 6. Screenshot of (off-line) replay of ssh session, based on the log generated by Kernort

access rules. *iDiscover* is capable of detecting published resources. The *iRun* function can remotely execute a program or AS on an available machine (or a *slice* of the machine), thereby mapping programs to machine resources. Finally, the *iCompose* function uses published resources as components and compose them into new entities, such as collaborative programs or composite AS.

Since iShare imposes no constraints on publishable resource properties, access protocols, and authentication methods, security becomes a concern for resource providers. It is desirable that the 'slice' of resource contributed by a provider to the community remains highly isolated from the remaining and private part of the resource. SODA and its protection mechanisms are expected to bring strong isolation and enhanced security to the iShare platform.

7 Related Work

Internet hosting has been moving from content hosting to application service (AS) hosting. The latter requires a higher degree of isolation in resource, administration, and

Configuration	null call	null I/O	stat	open close	slct TCP	sig inst	sig hndl	forc proc	exec proc	sh proc
w/o Kernort	11.0	11.3	119	146	23.8	11.9	28.5	4707	8016	15.K
w/ Kernort	11.0	11.4	120	147	24.3	12.0	29.0	4910	8221	16.K

Table 1. LMbench result: processes - time in μs

Configuration	2p/0K	2p/16K	2p/64K	8p/16K	8p/64K	16p/16K	16p/64K
w/o Kernort	9.1100	8.7500	9.6700	11.7	37.4	16.7	46.7
w/ Kernort	10.9	11.5	10.7	11.6	39.6	19.1	47.2

Table 2. LMbench result: context switching time in μs

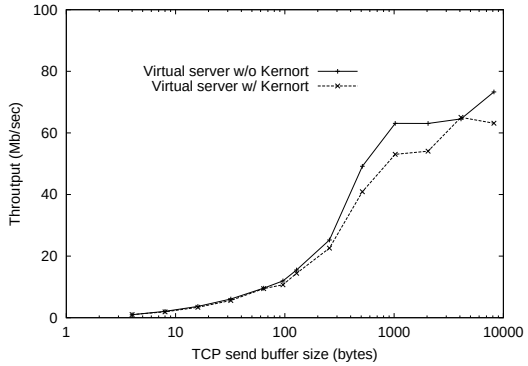


Figure 7. Comparison result: TCP throughput with and without Kernort

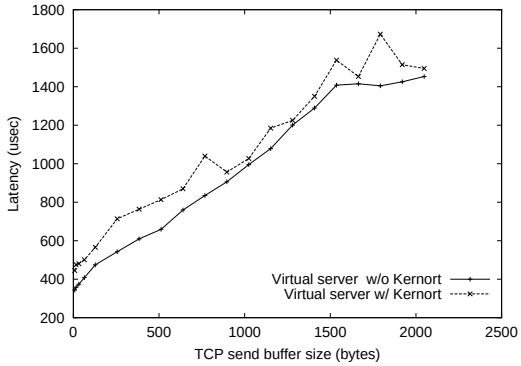


Figure 8. Comparison result: TCP latency with and without Kernort

fault/attack impact. Examples of ASHP platform include Oceano [8] of IBM, Utility Data Center [7] of HP, and Virtual Private Server (VPS) of Ensim [1].

Recently, virtual OS and isolation kernel have received significant attention. Representative projects include VMWare [6], Denali [23], Xen [10], UML [13], and UMLinux [12]. They all support the creation of virtual servers based on the guest OS/host OS architecture. It is noteworthy that virtual machine and networking technology has also been applied to Grid computing, such as in In-VIGO [5] and Virtuoso [20]. The protection and security mechanisms in SODA can be integrated into these systems.

It is noteworthy and interesting that the user-level mechanism for system call interposition in [15] is very similar to the way the UML [13] guest OS is implemented. They both exploit the *ptrace* mechanism provided by a UNIX-style OS (including Linux) for the surveillance of other processes. The purpose of [15] is intrusion detection and confinement, also similar to that of the UML.

The *honeypot* is "a computer system that is specifically designed to capture all activity ... of a criminal who has gained unauthorized access to the system" [11]. Recently, virtual OS (including UML) has also been applied to the deployment of honeypot [3], in order to achieve better attack confinement and log data capture. However, due to its different purpose from SODA, UML-based honeypot does *not* support resource isolation and virtual firewalling *between* honeypots. As to logging, UML-based honeypot supports untamperable logging of attacker's keystrokes. However, it is *not* capable of intrusion detection and AS-specific logging. Another project that addresses untamperable logging is ReVirt [14]. ReVirt is based on the UMLinux [12] guest OS. In ReVirt, the logging module is completely *outside* the guest OS. As a result, the log data of ReVirt may be less detailed than those captured by SODA's Kernort.

A paradigm similar to the ASHP is the execution of mobile code on a foreign host platform. It also involves a two-way security relationship between the mobile code and the host: we need protection of the latter against the former [22] *and vice versa* [18]. However, the ASHP paradigm is different in that the host-tenant relationship in an ASHP usually lasts longer, and that the inter-tenant relationship

Configuration	Create/ 0K	Delete/ 0K	Create/ 10K	Delete/ 10K	Latency/ Mmap	Prot Fault	Page Fault	100fd selct
w/o Kernort	160.2	83.1	226.2	90.2	792.0	14.1	15.0	21.9
w/ Kernort	160.8	83.6	228.6	90.2	772.0	14.2	15.1	21.9

Table 3. LMbench result: file and virtual memory systems latency in μs

should also be considered.

8 Conclusion

In this paper, we show that the protection of Application Service Hosting Platforms (ASHP) poses new research challenges. Due to ASHP's inherent requirement of openness, sharing, and mutual isolation, novel OS architecture and mechanisms are needed to provide protection and isolation between the host (ASHP infrastructure) and the tenants (AS), as well as between different tenants. We argue that the traditional single-level ASHP architecture is *not* adequate for such protection, and that the two-level architecture based on virtual OS technology appears to be a promising candidate. Based on SODA, our two-level ASHP architecture, we have implemented a number of effective and efficient protection mechanisms, including resource isolation, virtual switching and firewalling, and Kernort for intrusion detection and logging. These mechanisms will serve as the basis for more complicated security mechanisms and policies for ASHP protection.

9 Acknowledgments

We thank Xiaojuan Ren from the iShare Project for helpful discussion. We thank the anonymous referees for their constructive comments and suggestions. This work was supported in part by a gift from Microsoft Research and a grant from the e-Enterprise Center at Discovery Park, Purdue University.

References

[1] Ensim Virtual Private Servers. *http://www.ensim.com*.

[2] iShare. *http://www.ecn.purdue.edu/ParaMount/iShare*.

[3] Know Your Enemy: Learning with User-Mode Linux. *Honeynet Project White Paper, http://project.honeynet.org/papers/uml*.

[4] Snort. *http://www.snort.org*.

[5] The In-VIGO Project. *http://invigo.acis.ufl.edu/docs/aboutInVigo.html*.

[6] VMWare. *http://www.vmware.com*.

[7] HP Utility Data Center. *HP Technical White Paper*, 2001.

[8] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, and M. Kalantar. Oceano: SLA based Management of a Computing Utility. *IFIP/IEEE Intl. Symp. on Integrated Network Management*, May 2001.

[9] T. Aura, P. Nikander, and J. Leiwo. DOS-resistant authentication with client puzzles. *Security Protocols Workshop 2000, LNCS*, 2133, 2000.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfi eld. Xen and the Art of Virtualization. *19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, Oct. 2003.

[11] C. Brenton. Honeynets. *Technical White Paper, Institute for Security Technology Studies, Dartmouth College*.

[12] K. Buchacker and V. Sieh. Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects. *IEEE Symposium on High Assurance System Engineering (HASE 2001)*, 2001.

[13] J. Dike. User Mode Linux. *http://user-mode-linux.sourceforge.net*.

[14] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *USENIX OSDI 2002*, Dec. 2002.

[15] K. Jain and R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confi nement. *Network and Distributed Systems Symposium (NDSS 2000)*, 2000.

[16] X. Jiang and D. Xu. SODA: a Service-On-Demand Architecture for Application Service Hosting Utility Platforms. *IEEE HPDC-12*, June 2003.

[17] L. McVoy and C. Staelin. LMbench: Portable Tools for Performance Analysis. *USENIX Technical Conference*, 1996.

[18] O. K. Onbilger, R. Newman, and R. Chow. A Distributed and Compromise-Tolerant Mobile Agent Protection Scheme. *International Conference on Intelligent Agents, Web Technology and Internet Commerce (IAWTIC 2001)*, 2001.

[19] G. Owen. ghttpd. *http://gaztek.sourceforge.net/ghttpd*.

[20] A. Sundararaj and P. Dinda. Towards Virtual Networks for Virtual Machine Grid Computing. *USENIX VM Symposium*, May 2004.

[21] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profi ling in Shared Hosting Platforms. *USENIX OSDI 2002*, Dec. 2002.

[22] V. N. Venkatakrishnan and R. Sekar. Empowering Mobile Code Using Expressive Security Policies. *10th New Security Paradigms Workshop (NSPW 2002)*, 2002.

[23] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. *USENIX OSDI 2002*, Dec. 2002.