Towards Trustworthy Virtualisation Environments: Xen Library OS Security Service Infrastructure

Melvin J. Anderson, Micha Moffie, Chris I. Dalton
Trusted Systems Laboratory
HP Laboratories Bristol
HPL-2007-69
April 30, 2007*

trusted computing, virtualization, Xen hypervisor

New cost effective commodity PC hardware now includes fully virtualisable processors and the Trusted Computing Group's trusted platform module (TPM). This provides the opportunity to combine virtualisation, trusted computing and open source software development to tackle the security challenges modern computing faces. We believe that leveraging this technology to partition critical operating system services and applications into small modules with strictly controlled interactions is a good way to improve trustworthiness.

To support the development of small applications running in Xen domains we built a library OS. We ported the GNU cross-development tool chain and standard C libraries to the small operating system kernel included with the Xen distribution, and wrote an inter-domain communications (IDC) library for communications between Xen domains. To confirm the usability of our library OS we ported a software TPM to run on it as a typical application. We evaluated the performance of our IDC system and showed that it has good performance for the applications we envisage.

We have shown that a lightweight library OS offers a convenient and practical way of reducing the trusted computing base of applications by running security sensitive components in separate Xen domains.

* Internal Accession Date Only Approved for External Publication
© Copyright 2007 Hewlett-Packard Development Company, L.P.

Towards Trustworthy Virtualisation Environments:
Xen Library OS Security Service Infrastructure

Melvin J. Anderson
Hewlett-Packard Laboratories
Filton Road, Stoke Gifford
Bristol BS34 8QZ, UK
melvin.anderson@hp.com

Micha Moffie*
Computer Architecture Research Laboratory
Northeastern University
Boston, MA 02115, USA
mmoffie@ece.neu.edu

Chris I. Dalton
Hewlett-Packard Laboratories
Filton Road, Stoke Gifford
Bristol BS34 8QZ, UK
chris.i.dalton@hp.com

February 1, 2007

Abstract

New cost effective commodity PC hardware now includes fully virtualisable processors and the Trusted Computing Group's trusted platform module (TPM). This provides the opportunity to combine virtualisation, trusted computing and open source software development to tackle the security challenges modern computing faces. We believe that leveraging this technology to partition critical operating system services and applications into small modules with strictly controlled interactions is a good way to improve trustworthiness.

To support the development of small applications running in Xen domains we built a library OS. We ported the GNU cross-development tool chain and standard C libraries to the small operating system kernel included with the Xen distribution, and wrote an inter-domain communications (IDC) library for communications between Xen domains. To confirm the usability of our library OS we ported a software TPM to run on it as a typical application. We evaluated the performance of our IDC system and showed that it has good performance for the applications we envisage.

We have shown that a lightweight library OS offers a convenient and practical way of reducing the trusted computing base of applications by running security sensitive components in separate Xen domains.

1 Introduction

Information Technology is moving towards a world where it is possible to deliver highly flexible and scalable solutions over virtualised and constantly evolving infrastructures [14]. Such a world throws up significant trust and security challenges. One example is the challenge of assuring that the isolation goals of an IT system built on top of software based machine virtualisation technology are being met. With non-shared dedicated physical hardware this is a fairly tractable problem. Once hardware is virtualised and shared amongst differing and perhaps competing parties the problem becomes much harder. Even the mundane

*Work carried out at Hewlett-Packard Laboratories

but critical requirement of being able to identify hosts reliably and robustly within a management domain, already difficult in the physical world, becomes significantly harder in an infrastructure where hosts are virtualised. The increased trust and security management burden posed by problems like these threatens to outweigh the benefits that flexible virtualised environments bring.

In this paper we present our library OS and related inter-domain communication mechanism work for the Xen virtualisation layer [1]. With the library OS an application can run directly on Xen without requiring a full blown OS such as Linux underneath it. The motivation of the work is two-fold. Firstly, to support the reduction of the overall Xen trusted computing base by allowing the restructuring of the privileged Xen management and control plane applications. Secondly, to act as an implementation infrastructure for the provision of security services on top of the virtualisation layer. We consider these security services key in moving towards a fuller *trustworthy virtualisation environment* where the trust and security problems introduced by virtualised infrastructures can begin to be addressed [15].

In our view a *trustworthy* machine virtualisation layer will be a core component of next generation IT infrastructures. As a core component, a trustworthy layer will require robust, reliable and predictable behaviour. The amount of code that currently forms the trusted computing base (TCB) of Xen, i.e. the code that needs to be trusted to uphold the Xen isolation properties for example, is too large in our opinion for Xen to be considered a trustworthy virtualisation layer. The TCB of Xen includes both the VMM layer itself and also the necessary management functionality. At the moment, this management functionality is hosted on top of a full Linux OS running in a privileged Xen domain (domain 0). Our library OS work and associated inter-domain communications support the restructuring and reduction of the Xen trusted computing base leading towards a more trustworthy VMM.

A trustworthy VMM alone however will not be sufficient in addressing the trust and security challenges introduced by the move to virtualised environments. We believe that some form of trust management framework, i.e. a framework for securely bootstrapping management components on top of a trusted VMM along with a set of appropriate security services is also required [5]. As an example, an integrity security service[1] layered on top of a trustworthy VMM would support the running of multiple operating systems with varying trust levels concurrently whilst still allowing for the meaningful attestation and remote verification of the identity and trustworthiness state of individual operating systems and configurations running on that virtualised platform [15]. Our library OS work and associated inter-domain communications act as an excellent foundation for the implementation of security services on the Xen VMM.

The rest of the paper is organised as follows: In section 2 we go over the related work. We present the design and implementation of our new environment in section 3 and provide performance numbers in section 4. We report our experience porting the virtual TPM security service in section 5. We discuss further work that we plan to do based on the library OS in section 6, and discuss our approach and conclude in section 7.

2 Related Work

2.1 Virtualisation

Virtualisation is a technology that has been around at least since the VM/CMS in the 1960's – but it is only recently that commodity processors from AMD and Intel have had hardware extensions to support virtualisation. Before then VMMs such as Xen2 have used para-virtualisation to obtain reasonable performance. Xen3 has support for the new generation of virtualisable processors, allowing unmodified guest operating systems and their applications to run under Xen.

[1]Another example is a virtual network service [7]

2.2 Containment

There have been a number of examples of operating systems featuring controls designed to offer application containment properties above and beyond normal process based isolation. Several have been based on the Bell-Lapdula information flow control model [2]. Some more recent systems have attempted to cope with the changing nature of applications and services and are more network control centric [6, 18]. However, in all these examples the large body of operating system kernel code has to be trusted. In principal, a VMM can be a fairly small and tight code base with much less of an attack surface than a typical OS, making the VMM a more reasonable placeholder for our trust.

2.3 Open Trusted Computing

With the recent additions to commodity hardware (hardware virtualisation, trusted platform module (TPM)) we can run legacy guest operating systems within a virtual machine to support legacy applications, while running para-virtualised operating systems within parallel virtual machines which are aware of and can take advantage of the new security features.

The Open Trusted Computing (OpenTC) project [15] manages risk by compartmentalisation, using the Xen hypervisor [1] and L4/Fiasco [16] micro-kernel to enforce controlled sharing between compartments. The Private Electronic Transaction (PET) demonstrator described in [15] runs a financially sensitive application in a trusted compartment, while other applications run in less trusted compartments. The financial application runs in an environment with limited connectivity to reduce exposure to sources of threat, and runs trusted code to minimise vulnerabilities. The untrusted compartment only runs applications whose compromise would have less impact.

2.4 Microkernels and Hypervisors

The principle of compartmentalisation can be applied to components of applications, not just complete applications. Hohmuth et al [12] show how untrusted components can be used in a trusted system. They propose extending VMMs with IPC primitives to provide efficient, secure communications between trusted and untrusted components.

The designers of hypervisors such as Xen, and micro-kernels such as L4 have chosen different styles of interface to support their differing intended use. Xen runs complete operating systems in separate domains—either unmodified (when running Xen3 on the latest range of processors from AMD and Intel supporting full virtualisation), or using a para-virtualisation interface. L4 emphasises IPC, and aims at a particularly fast implementation. This allows efficient implementation of finer-grained protection domains, and allows operating system services to be implemented as servers accessed through IPC.

Hohmuth et al view traditional separation VMMs and micro-kernels as the extreme ends of a spectrum. We agree that the design space between VMMs and micro-kernels is an area worthy of investigation. The promise is to be able to run legacy applications and operating systems without any modification alongside new security critical applications, where the TCB of the new application can be minimised by partitioning it into cooperating components.

Hohmuth et al propose adding IPC primitives to a VMM. Xen's para-virtualisation interface gives system-level code access to hypervisor calls which extend the hardware interface with operations on virtual CPUs, shared memory and event passing between virtual CPUs. It provides operations for shared memory and event passing that allow IPC to be implemented as a library within domains without needing explicit changes to Xen. This is the approach we take in this paper.

3 Design and Implementation of the Library OS

Our aim is to minimise the TCB of applications, which means that we wish to minimise the system code in critical domains to just what is needed by the domain's application code. This suggests that we should implement a library operating system, like that proposed for the Exokernel architecture [8].

This section describes how we designed and implemented a library OS to run as a para-virtualised guest operating system within a Xen domain. We implemented the infrastructure needed to run small secure servers under Xen as a building block for constructing applications. The objective was to package the development tool chain, standard libraries and inter-domain communications (IDC) so that it would be as easy to prepare image files to load into a Xen virtual machine as it is to compile applications to run under Linux.

The aim is to allow domains to maintain shielded internal state that can only be accessed and updated in a controlled manner. In the terminology of object oriented programming, we are implementing a secure infrastructure for type managers.

The API provided to a guest OS by the Xen hypervisor is very close to the underlying hardware API. This means that although Xen provides secure sharing of low-level resources, any services or abstractions that the application requires must be provided by libraries within it's domain, or IPC calls to servers in other domains.

The library OS consists of:

- a cross-development environment based on the GNU tool chain,
- a set of libraries including the Red Hat "newlib" C library,
- a simple kernel based on Mini-OS—a simple example kernel supplied with the Xen source distribution, and
- an inter domain communication mechanism (IDC) layered on Xen's shared memory and event mechanisms.

Figure 1 shows an overview of the library OS and application interfaces. The application, library and kernel all run in a single address space within a single protection ring (ring 1 in the x86 version of Xen). Applications are free to use any interfaces, but it is recommended for maximum portability that the standard C library functions [13] provided by Red Hat `newlib` are used.

3.1 Development Tool Chain

The GNU binary utilities and GCC compiler were the natural choice of development tools. GCC is a mature compiler that produces good quality code. It is used to to compile Xen and Linux on the x86 architecture, so it integrates well with the other software components.

The aim is to achieve maximum source code compatibility between cross-compiling for the library OS and compilation to run in a Linux process. There are differences between the two environments and the wide range of libraries available to Linux applications are not available to stand-alone Xen domains. However, even limited compatibility will allow easier application development by using Linux as an environment for initial debugging. In particular, no special command line options should be needed—appropriate header files and libraries should be available without needing to be specified explicitly on the command line.

An advantage of the GNU tool chain is that it supports cross-compilation, and is widely used for embedded applications. There are two ways to use the tool chain for cross-compilation, the easiest being to use the tool chain installed as part of Linux distributions. The disadvantage is that command line options

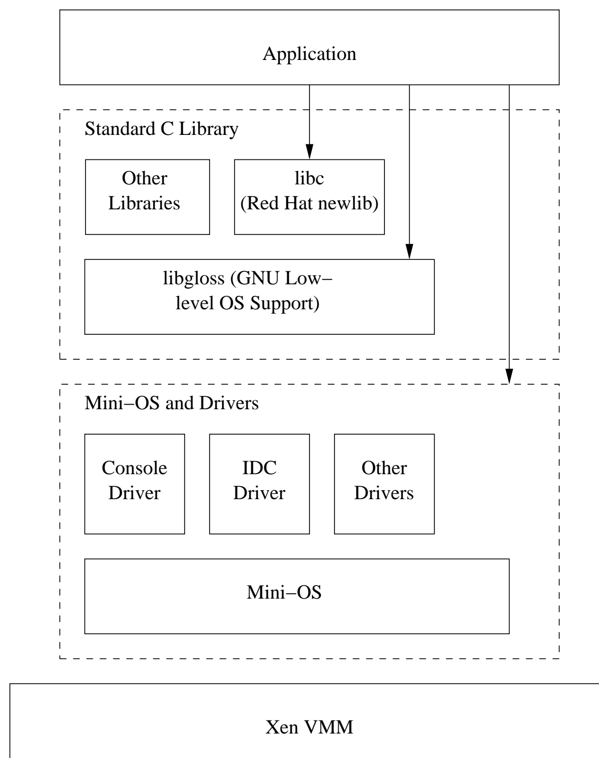

Figure 1: Overview of application and libraries

are needed to specify the locations of headers, libraries, initial code and loader directives, which makes it difficult to use existing Makefiles without major modifications.

The alternative approach used in this project was to modify the GNU compiler and binary utilities to support a new target architecture, a so-called "fully defined" target. This allows the compiler to be called explicitly as "`i386-minios-gcc`", and it selects the correct headers, libraries, initial start up code and loader directives. This convention works well when compiling existing libraries, particularly those packaged using "`automake`" and "`autoconf`", which have integrated support for cross-compilation. It is interesting to note that the developers of the embedded library uClibc [3] have made the same decision for similar reasons—see "Do I really need to build a uClibc tool chain" at [4].

It was difficult to locate the information needed on porting the GNU tools. There is a lot of documentation, but parts of it have not been updated to describe the current version of the tool chain. The build process is confusing, being based on the so-called "Cygnus tree". We were able to piece together sufficient understanding by reading [20] and reviewing mailing list archives to fill in the gaps.

The main changes to the GNU tool chain were to add the new target architecture to the build scripts based on existing `i386` targets, and to modify the default linker scripts, include file and library search paths, and initial code.

3.2 The Library OS

The C language specification [13] specifies library functions that application writers can expect to be available to C programs. There are at least three widely available open source implementations of this library: GNU glibc [17], uClibc [3], and Red Hat `newlib` [19]. The GNU glibc library is comprehensive, but large, and is not well suited for small stand-alone applications. Red Hat `newlib` is mature code, has been used for many embedded projects, and integrates well with the GNU tool chain, and was therefore chosen for this

project. uClibc has not been investigated, but may also be an alternative worth considering.

As well as adding the new target architecture identifier to the Red Hat `newlib` library configuration scripts, it was necessary to write low-level Posix-like functions on which `newlib` depends. Some of these functions, such as `fork` and `exec`, are null place holders as the equivalent functionality is not provided by our library OS.

There are a number of options when choosing a kernel, such as writing one from scratch, porting an existing kernel to Xen, or extending the Mini-OS example kernel which is part of the Xen distribution. The choice was made to use the Mini-OS example kernel because it already supports the Xen para-virtualisation interfaces, and any bug fixes or extensions can usefully be fed back to the Xen team.

The application, libraries and kernel all run in a single address space in supervisor state (which in the i386 version of Xen is actually ring 1), so there is no distinction between kernel and library code. This is an appropriate choice for running small trusted services, but not general user code.

The aim was to minimise changes to the Mini-OS example kernel, preferring to add code in the interface between Red Hat `newlib` (a library called `libgloss`—GNU low-level operating system support) or additional libraries. There are some small changes to Mini-OS which are generally useful and do not significantly increase its size or complexity.

3.3 Inter-domain Communication

Servers built using the library OS need to be able to communicate with other domains. Conventional operating systems run as guests in Xen domains already have well developed networking support, so it is possible to use protocols such as TCP/IP over virtual network interfaces for inter-domain communications. For the small servers we expect to run over the library OS a full TCP/IP protocol stack adds unnecessary code size and complexity. Hohmuth et al [12] propose adding IPC hypervisor calls to a VMM. We are using the existing para-virtualisation interface provided by Xen instead to implement inter-domain communications as a library component layered over Xen's shared memory mechanisms. This has the advantage of needing no changes to the Xen VMM itself, and ensures that any mandatory security policy applied by Xen security modules also applies to our IDC mechanism.

We developed a kernel module for Linux and a module for the library OS which exposes the communications API to the applications. We have striven to develop a simple API which is compatible (as much as we can) to known ones. This API allows applications running either on our library OS or on para-virtualised Linux kernels to communicate.

The IDC API is similar to file IO with a hint of resemblance to Socket IO. The core API includes 6 functions:

Init Initialise the channel,

Close Closes the channel,

Create The Create operation allows the user to write to the channel. The user is expected to specify the remote domain id,

Connect The Connect operation enable the user to read from a remote domain (with a valid ring already created). The user supplies the remote domain id as well as a grant reference associated with the ring created in the remote domain,

Write Write a buffer to the channel, and

Read Read to a buffer from the channel.

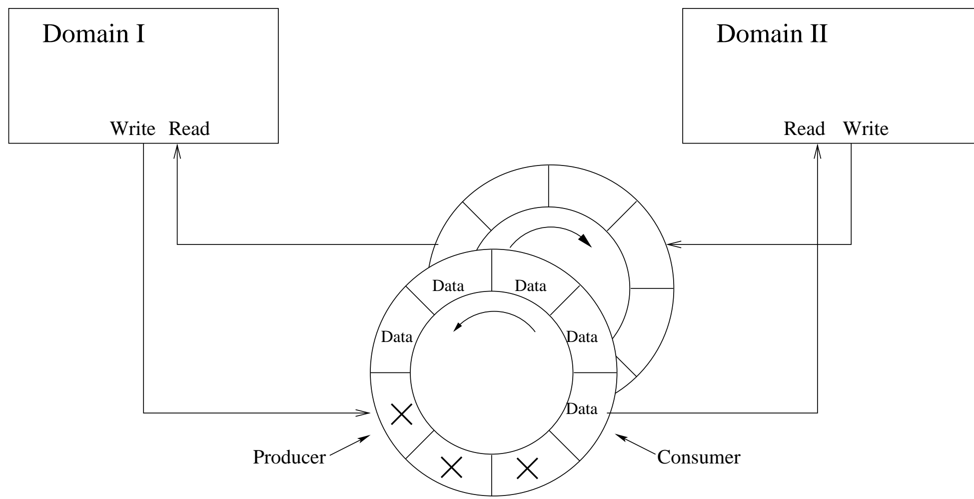

Figure 2: Inter-domain communications using a shared ring

A client program may create a one way channel to write only (create) or read only (connect), or a two way channel (create and connect).

Figure 2 shows an outline of the implementation of inter-domain communications. We use separate ring buffers for write and read channels, each ring buffer supporting one way communications. Each ring buffer is allocated in memory shared between domains. Short messages are passed in ring buffer entries, while larger messages are passed using separate physical pages which are pointed to from the ring buffer entry.

4 Inter-domain Communication Performance

The aim of performance measurement is to confirm that the cost of inter-domain communication is not excessive for the class of application envisaged. It is expected that two styles of use will predominate: request-reply transactions and bulk data transfers. Performance for request-reply transactions requires low message passing latency, while for bulk data transfers overall throughput is important. These observations influence the tests performed: firstly a check for scalability over total amount of data transferred to ensure that our measurements are not being influenced by cacheing or other effects, secondly a test to measure throughput against different ring buffer sizes, and finally a measurement of round trip delay. These measurements were run between domains running Linux rather than the library OS so that meaningful comparisons could be made with TCP/IP performance as a reference. The bulk of the IDC implementation is similar for the Linux driver, and the library OS implementation.

For our tests we have used an AMD Athlon(TM) 64 dual core processor with 1 MB cache per core and 1GB of main memory. Our domain 0 was configured with two virtual CPUs and 256MB of memory. The user domains had one virtual CPU and 256MB of memory available. All domains were running Fedora Core 5 GNU/Linux [10].

4.1 Scalability

We first investigate the scalability of the communication channel in relation to different number of transactions. Figure 3 shows the impact of different number of transactions (where each transaction is a write of a 4KB block) on the performance of the channels. In our test we use Enhanced ttcp (ettcp) [9], a program based on ttcp which allows to measure network throughput. We have modified ettcp to allow us to use IDC

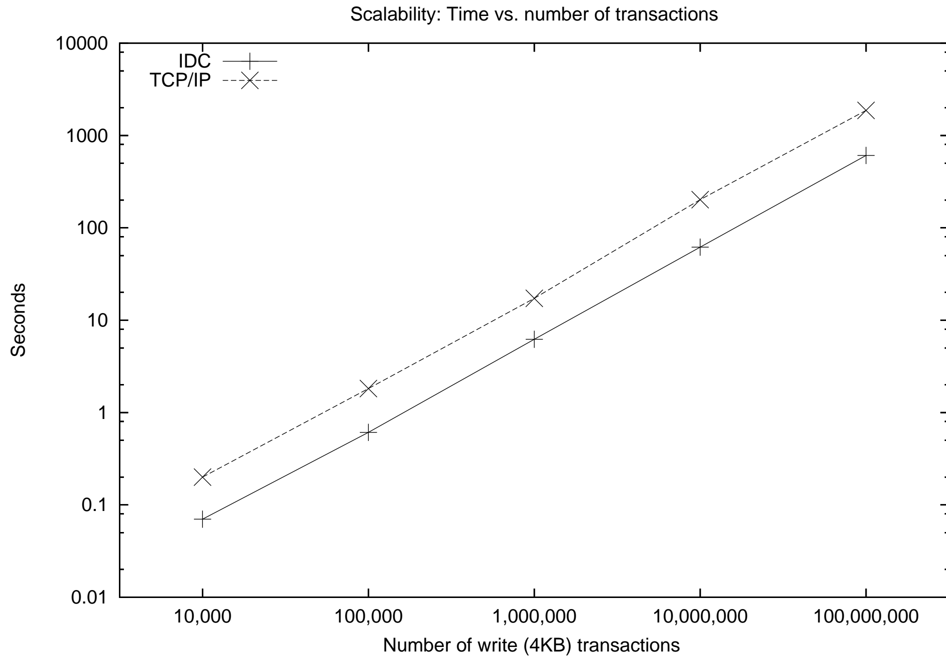

Figure 3: The impact of the number of transactions

as well. In a user domain, we run ettcp which continually writes 4KB blocks to the channel and use ettcp on domain 0 to continually read the blocks.

In the scalability test we used a program in a user domain to write and another program in domain 0 to read. We also tested whether the direction of data had an impact on the performance. Figure 4 shows the effect of the source and destination on the throughput. On the left side of the figure we write (considering a different number of transactions) from domain 0 to a user domain. The right side of the figure is a repeat of the test where we write from a user domain to domain 0.

The figure clearly shows that when using the TCP/IP protocol family writing from a user domain to domain 0 is faster than writing from domain 0 to a user domain. This is due to an additional copy of the data when we transfer from domain 0 to a user domain. The IDC channel does not have this effect, the throughput is not dependent on data direction, although some start-up effects can be seen.

4.2 Throughput

We evaluated the impact of the number of entries in the ring on the throughput of our communication channel. We compared different write buffer sizes (same number of transactions) and considered the effect of different ring sizes [2]. Figure 5 and 6 show the time needed for 100,000,000 transactions and 10,000,000 transactions respectively. In figure 5 we can clearly see the effect of the optimisation we implemented in the ring buffer: Enabling small enough data to be placed in the ring itself—instead of allocating a new page proved very useful. The figure clearly shows a jump in time when the buffer being written can't fit into the ring. This happens between 12 and 32 bytes for the 128 ring (since we can only fit 31 bytes), between 100 and 128 bytes for the 32 ring (can only fit 127 bytes) and between 500 and 512 bytes for the ring of size 8 (where we can only fit 511 bytes). One can see that the TCP/IP protocol handles this much more gracefully.

Figure 6 shows the performance when the buffer sizes are bigger. It is interesting to note that increasing

[2] We always "waste" one entry, a ring size of 8 has in effect only 7 usable entries

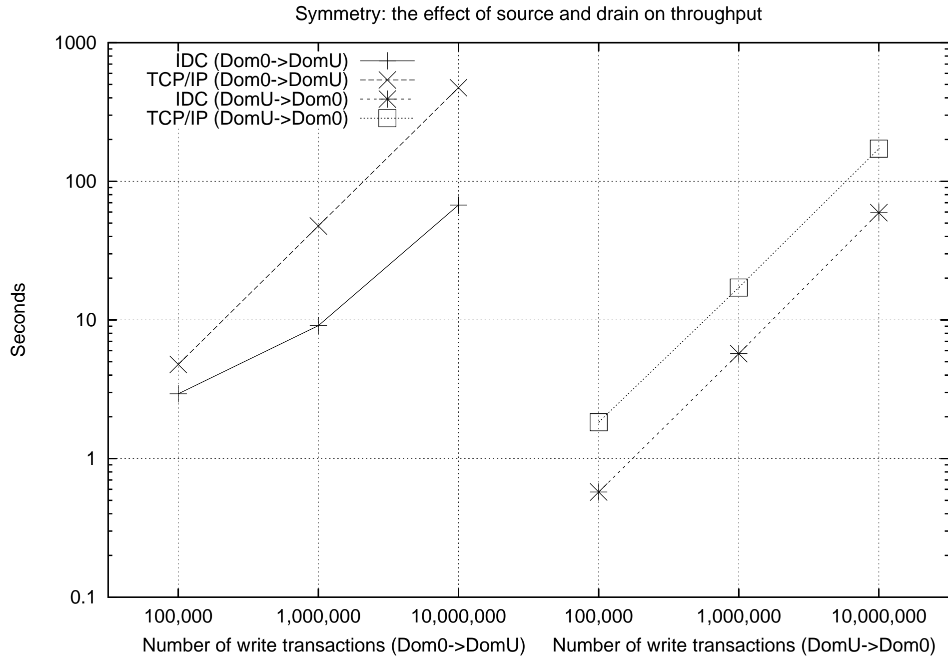

Figure 4: Impact of the direction of the write transactions

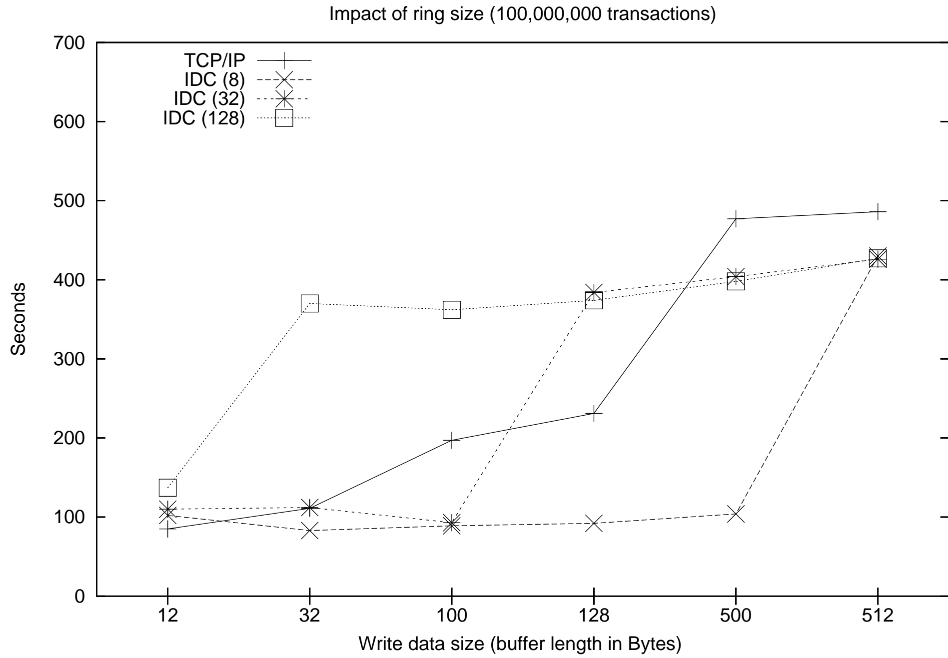

Figure 5: Impact of ring buffer size, 100,000,000 write transactions.

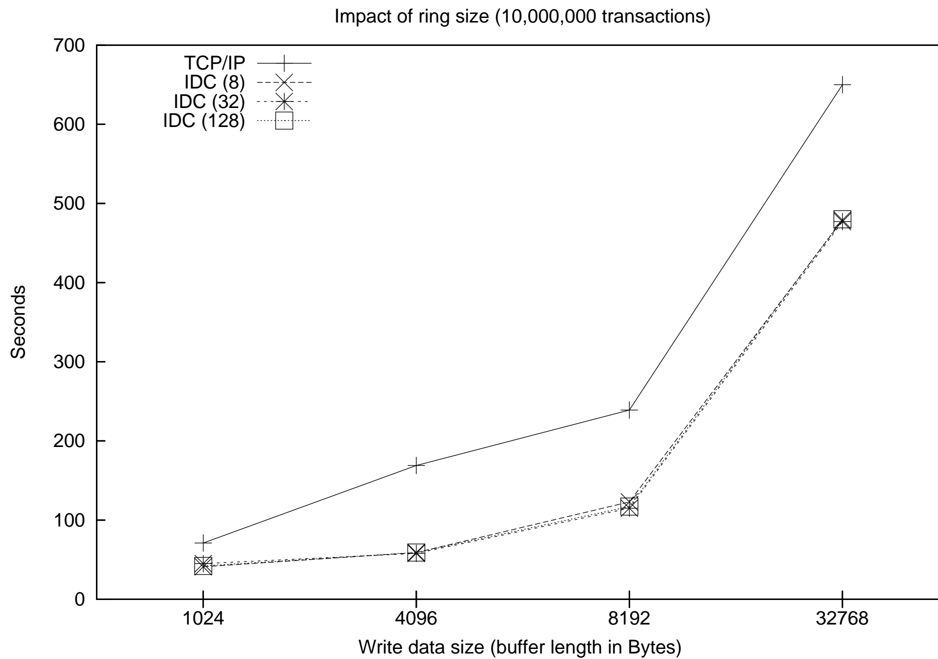

Figure 6: Impact of ring buffer size, 10,000,000 write transactions.

the number of descriptors in the ring doesn't have a significant effect on throughput for the conditions in this test.

4.3 Latency

In addition to throughput, latency is an important measure in many applications. We measured the latency (round trip of data) in figure 7. We compare the latency of different data sizes and different number of transactions. Each transaction here is defined here as a write and a read (and a corresponding read and write in a remote domain). The figure shows the relative performance of our IDC and TCP/IP when sending different buffer sizes.

4.4 Performance Implications

We have shown that our inter-domain communication mechanism has a significantly higher throughput and lower latency than using TCP/IP to communicate between Xen domains. Of course, this is not unexpected, considering the much greater functionality and complexity of the TCP/IP protocol stack. However, we believe that our IDC mechanism is well suited for communicating with small security-sensitive applications running in their own Xen domains. In order to validate this further, we ported a software implementation of the Trusted Computing Group's Trusted Platform Module (TPM), as described in the next section.

5 Case Study—Porting a virtual TPM

As a case study, to evaluate our design and implementation we chose to port the virtual Trusted Platform Module (*virtual* TPM or vTPM) to our system. *Physical* TPM (Trusted Platform Module) technology from the Trusted Computing Group (TCG) provides a firm foundation for enabling trust in a computing platform.

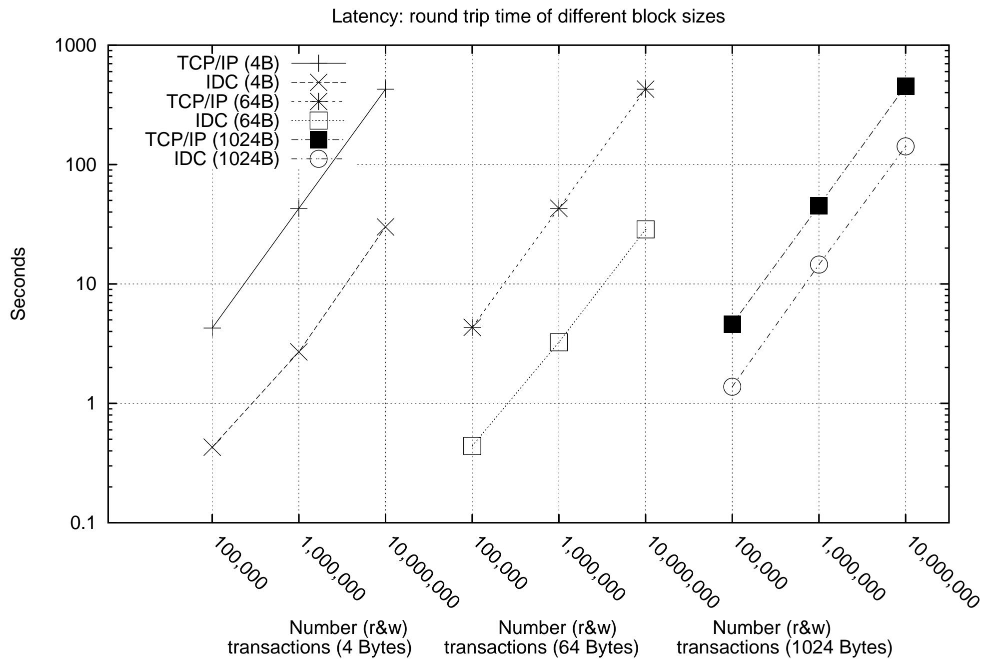

Figure 7: Impact of write and read size on latency

For example it can be used to ensure that a system boots with an operating system version and configuration that its owner considers trustworthy. Further, through attestation and verification, it allows for a platform to convey information about its trustworthiness state in an assured manner to other parties interacting or wishing to interact with the platform. However it is unlikely that a single OS version or configuration can be considered trustworthy or suitable for all uses or applications of a particular computing platform. For example, a configuration considered trustworthy enough for on-line banking may not be performant enough for computer gaming. Likewise, a platform configuration optimised for game playing may not be considered trustworthy enough for on-line banking.

A virtual TPM bound to a particular virtual machine instance can be used to attest and remotely verify the trustworthiness state of that particular VM and its associated operating system and configuration, much like a physical TPM can be used to attest and remotely verify the trustworthiness state of a physical machine and its associated operating system.

When combined with a suitable trustworthy virtualisation layer and appropriate trust chaining [15], we have the capability to run multiple operating systems with varying trust levels concurrently whilst still allowing for the meaningful attestation and remote verification of the trustworthiness state of individual operating systems and configurations running on a platform. With such a system, a platform can safely and flexibly run both on-line banking and gaming applications simultaneously whilst still for example being able to convince a banking service that the platform meets the banks security requirements.

5.1 The current Xen vTPM Architecture

Figure 8 shows the software architecture of the vTPM as currently delivered with Xen 3.0 [21]. On the right, a TPM enabled user domain is running. A front end (FE) TPM driver is used to forward commands to domain 0. The back end (BE) driver in domain 0 forwards all commands to the vTPM manager. The manager demultiplexes the commands (which may originate from multiple domains) to vTPM daemons. A

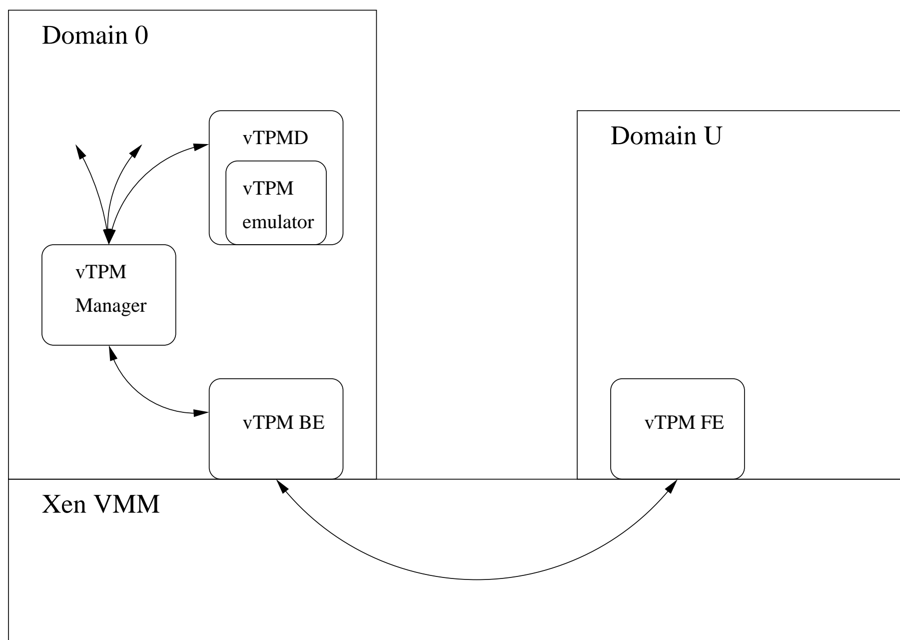

Figure 8: vTPM software architecture for Xen 3.0

vTPM daemon is responsible for executing commands on behalf of one and only one user domain, and each domain has a private instance of the vTPM daemon.

The vTPM manager runs in domain 0, and handles two types of command. The first type are virtual TPM commands which are handled by the manager itself—examples are load and save the TPM non-volatile memory. The second type of commands are real TPM commands which are forwarded to the vTPM daemon. The daemon can be viewed as a wrapper around a TPM emulator which actually handles the command. Currently the vTPM daemon is executed in domain 0 and communicates with the manager through a set of Unix pipes.

5.2 Porting the vTPM to run under the library OS

To port the vTPM we had to overcome two sets of obstacles. The first set was changing the communications. For simplicity we did not modify the communication channels between the vTPM daemon and the vTPM manager and allow all TPM and vTPM commands to proceed as before. We modified the vTPM daemon running in Dom0 to communicate with the TPM emulator running in our library OS. The vTPM daemon in domain 0 effectively became a proxy to the TPM emulator running in our system. We used our inter-domain communication channel and had to make only local and relatively simple modifications. Figure 9 shows the modified software architecture.

The second set of changes had to be done in TPM emulator. First we added a channel to allow us to communicate with the vTPM proxy running in domain 0. Secondly we cross-compiled the TPM emulator for our library OS. We used our new compilation tool chain and our ported libraries to compile the TPM emulator with very little effort. The TPM emulator requires multiple precision integer arithmetic functions provided by the GNU multiple precision library (GMP). We found that having made the effort to build the GNU tool chain for our library OS, all we needed to do was to modify the build files to add the new host architecture (`i386-minios`).

Overall we had to do very little modification to the code, makefiles etc. We have modified less than 200 lines on the vTPM daemon in domain 0, and less than 500 lines to port the TPM emulator to the library OS.

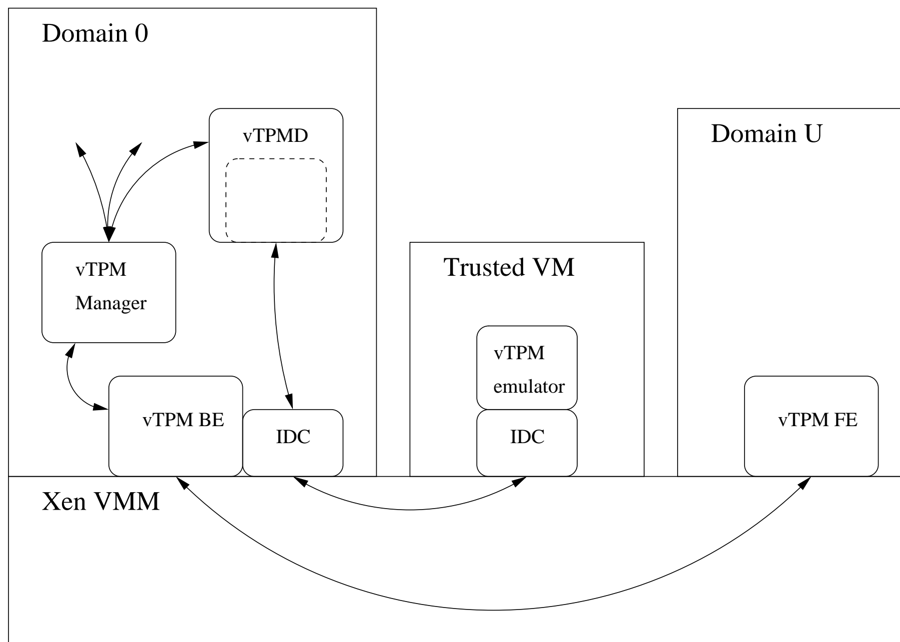

Figure 9: modified vTPM software architecture

6 Further Work

We intend to improve and extend the library OS, and to submit it for inclusion in the "extras" section of the Xen source tree.

We also plan to gain further experience of porting security services on top of the library OS. Our experience in porting the vTPM highlighted an interesting issue when running applications in a virtualised environment, and that was how to generate cryptographically secure random numbers for keys.

6.1 Random number generation

A physical TPM includes a cryptographic co-processor which implements several cryptographic operations such asymmetric decryption and encryption. In addition the co-processor includes a random number generator to generate keys.

There is currently no support for generating random numbers in the library OS so we simply forwarded random bytes from the vTPM proxy running in domain 0 on each request (using /dev/urandom). This satisfied the TPM emulator (which passes all the internal tests).

This solution shows—as a proof of concept—that the TPM emulator can be ported and executed in a separate domain. However, the method used for generating keys is not entirely satisfactory, and we plan to investigate alternative strategies for a virtualised environment.

7 Conclusion

We have developed a small and specialised environment based on the Mini-OS example running on Xen which allows us to execute sensitive applications. This environment is isolated from other domains and has a very small TCB. Both of these properties increase the trustworthiness of the application and environment executed.

The environment is not intended as a general purpose OS. In order to maintain a small TCB many features (including file system support and networking) are not present in our environment. Limited access to these resources may be provided through a communication channel—but will incur performance overhead. There is a tradeoff between making our environment more general and keeping the TCB as small as can be. Instead of starting with the minimum kernel and then layering only essential libraries on top, we could have taken a more sophisticated kernel such as Linux, and pared it down to what was necessary for the application. This could be a better approach where physical device drivers are to be moved to a separate Xen domain [11]. This decision should be made specificly for each security service taking into account the tradeoffs between security and functionality.

There is also a management issue: Pulling out services into separate domains can increase the complexity of managing many different VMs. For example, a vTPM which runs in a separate domain is conceptually attached to the domain using the TPM. If the domain using the TPM is being migrated to another physical machine the vTPM will need to be migrated as well. All these issues introduce complexity and overhead for managing the data centre.

The effort required porting security sensitive applications into our environment as well as the additional complexity of managing the data centre services poses a tradeoff between additional isolation and security against manageability. The vTPM example is a service with security needs that are worth the porting effort despite the increased management complexity—but this is not necessarily true for every security service. Careful consideration must be exercised before deciding to port and manage a security service in a separate trusted environment.

We believe our framework would prove useful for securing sensitive services. The library OS and tool chain can decrease porting effort. The resulting strong isolation and small TCB forms a corner stone for increasing the trustworthiness of security sensitive applications.

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[2] D. E. Bell and L. J. La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Hanscom Air Force Base, Bedford, Massachusetts, March 1976.

[3] A C library for embedded Linux. See website `http://uclibc.org/`.

[4] A C library for embedded Linux, frequently asked questions about the tool chain. See website `http://uclibc.org/FAQ.html#toolchain`.

[5] Serdar Cabuk and David Plaquin. Trusted component management for virtualized platforms. Technical report, Hewlett-Packard Laboratories, 2007. In preparation.

[6] Chris Dalton and Tse Huong Choo. An operating system approach to securing e-services. *Commun. ACM*, 44(2):58–64, 2001.

[7] Chris I. Dalton. Local and wide area virtual network support for Xen and VMware. Technical report, Hewlett-Packard Laboratories, 2007. In preparation.

[8] D. R. Engler, M. F. Kaashoek, and Jr. J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM Press.

[9] Enhanced ttcp. See website http://sourceforge.net/projects/ettcp/.

[10] Fedora core Linux. See website http://fedora.redhat.com/.

[11] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williams. Safe hardware access with the xen virtual machine monitor. In *Proceedings of the First Workshop on Operating System and Architectural Support for the on-demand IT Infrastructure*, October 2004.

[12] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop: beyond the PC*, page 22, New York, NY, USA, 2004. ACM Press.

[13] International Organization for Standardization. *Programming Language – C*. ISO/IEC 9899.

[14] Mahesh Kallahalla, Mustafa Uysal, Ram Swaminathan, David E. Lowell, Mike Wray, Tom Christian, Nigel Edwards, Chris I. Dalton, and Frederic Gittler. SoftUDC: A software-based data center for utility computing. *Computer*, 37(11):38–46, 2004.

[15] Dirk Kuhlmann, Rainer Landfermann, Hari V. Ramasamy, Matthias Schunter, Gianluca Ramunno, and Davide Vernizzi. An open trusted computing architecture—secure virtual machines enabling user-defined policy enforcement. Technical Report RZ 3655, IBM Research, 2006.

[16] Jochen Liedtke. Toward real microkernels. *Commun. ACM*, 39(9):70–77, 1996.

[17] Sandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper. *The GNU C Library Reference Manual*. GNU Press, Boston, MA, USA, 2001. Two volumes. For glibc version 2.2.x.

[18] P. Loscocco and S. Smalley. Meeting critical security objectives with security-enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001.

[19] Red Hat Inc. *The Red Hat newlib C Library*, 2004. Software and documentation available from ftp://sources.redhat.com/pub/newlib/index.html.

[20] Ian Lance Taylor. The GNU configure and build system. Technical report, Cygnus Solutions, 1998. Available from http://www.airs.com/ian/essays/.

[21] Xen user's manual for Xen version 3.0. Available from http://www.cl.cam.ac.uk/research/srg/netos/xen/documentation.html/.