Improving Xen Security through Disaggregation

Derek G. Murray Grzegorz Milos Steven Hand
University of Cambridge Computer Laboratory
[Firstname.Lastname]@cl.cam.ac.uk

Abstract

Virtual machine monitors (VMMs) have been hailed as the basis for an increasing number of reliable or trusted computing systems. The Xen VMM is a relatively small piece of software – a *hypervisor* – that runs at a lower level than a conventional operating system in order to provide isolation between virtual machines: its size is offered as an argument for its trustworthiness. However, the management of a Xen-based system requires a privileged, full-blown operating system to be included in the trusted computing base (TCB).

In this paper, we introduce our work to *disaggregate* the management virtual machine in a Xen-based system. We begin by analysing the Xen architecture and explaining why the status quo results in a large TCB. We then describe our implementation, which moves the *domain builder*, the most important privileged component, into a minimal trusted compartment. We illustrate how this approach may be used to implement "trusted virtualisation" and improve the security of virtual TPM implementations. Finally, we evaluate our approach in terms of the reduction in TCB size, and by performing a security analysis of the disaggregated system.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Information flow controls

General Terms Design, Security

Keywords Disaggregation, trusted computing base, virtual machines

1. Introduction

Many researchers have proposed using virtual machine monitors (VMMs) to improve the reliability, security or assurance of computing systems [10]. They argue that the small code-base of a VMM can be used as justification for trusting its correct operation; this same argument has been applied to microkernels [31]. The assumption is made that the integrity of the lower layers (including the VMM) is axiomatic, therefore it is possible to infer the integrity of components (individual virtual machines) running above [3, 7]. We aim to use the Xen hypervisor [4] as the basis of an "open trusted computing" system [20]. Such an approach extends the purview of trusted computing hardware into virtual machines (VMs); it also allows the user discretion over when to use this hardware, and enables the use of trusted and untrusted software side-by-side.

VEE'08, March 5–7, 2008, Seattle, Washington, USA

In order to evaluate the trustworthiness of a software system, it is necessary to identify its trusted computing base (TCB). The integrity of the TCB may then be measured, and this unforgeable measurement can be used to generate a key that may be used for attestation, encryption and decryption operations [33]. However, the TCB of a current Xen-based system comprises, in addition to the VMM, a fully-fledged operating system (known as *Dom0*) and a set of user-space tools. These tools are used to perform several tasks that require elevated privileges, including the creation of new VMs. Due to the inclusion of user-space software in the TCB, the size of the TCB is practically unbounded, as it can include any software that may be run by the administrator of the physical platform. Furthermore, any measurement of the TCB cannot be used to guarantee the integrity of a virtual machine, because the system administrator may run arbitrary privileged code at any time. In effect, this implies that the system administrator must be trusted, which impairs the usefulness of Xen in utility computing scenarios [26].

In this paper, we demonstrate the use of *disaggregation* to shrink the TCB of a virtual machine in a Xen-based system. We have transferred the VM-building functionality into a small, trusted VM that runs alongside Dom0. We had two main goals in this work. The primary goal was to reduce and bound the TCB of a Xen-based system, in order to improve its security. In particular, by removing Dom0 user-space from the TCB, we would remove the requirement that the administrator of the physical platform be trusted. We anticipate that this will be relevant for the users of virtualisation in utility computing. A closely-related goal was to demonstrate that a VM running on a disaggregated system maintains the same confidentiality and integrity properties as a physical machine, assuming the inviolability of the TCB. This should allay many concerns over switching to a virtualised approach.

We begin by introducing the Xen architecture, the domain building process, and how these relate to the TCB of a Xen-based system (Section 2). We then describe our framework for selecting a TCB for the disaggregated solution, and introduce new criteria for evaluating a TCB that should be considered in addition to the raw number of lines of code (Section 3). We discuss the implementation of our disaggregated domain builder design, which has also necessitated the development of a lightweight inter-VM communication mechanism (Section 4). We evaluate our disaggregated approach against our primary goals by enumerating the contents of the TCB and performing a security analysis (Section 6). Finally, we consider related work (Section 7) and draw conclusions (Section 8).

2. Background

The existing Xen VMM architecture and privilege model have implications for security, and enable an administrator to take full control of all virtual machines running on the same host. It is therefore impossible for the users of a virtual machine to trust in the confidentiality and integrity of that virtual machine. This

may inhibit the uptake of virtualisation, especially in the class of applications where confidentiality is required by law.

In this section, we provide a brief overview of the Xen architecture (Subsection 2.1), describe the domain building process (Subsection 2.2), and discuss the design of Xen with respect to its trusted computing base (Subsection 2.3).

2.1 Xen architecture

The traditional representation of a Xen-based system shows several virtual machines, drawn separately and side-by-side on top of the Xen VMM (hypervisor), which itself is positioned above the platform hardware [4]. A single VM, *Dom0*, is distinguished as a management or control VM, which is always booted by the hypervisor, and contains the control plane software. However, this picture ignores the trust relationships between Dom0 and the remaining VMs. In our work we aim to reduce and bound these trust relationships, so that Dom0 can accurately be positioned side-by-side with other VMs.

Xen makes extensive use of *paravirtualisation* in order to achieve good performance. This process entails porting VM kernels to run on a new machine architecture that corresponds to the support provided by the VMM. On the x86 architecture, four processor rings are used to define privilege levels. In most common operating systems, the kernel runs in ring 0 (the highest privilege level), and user processes run in ring 3 (the lowest privilege level). Xen must have full control over and protection from the VMs that it hosts, so it runs in ring 0. VM kernels are paravirtualised in order to run in ring 1. User processes continue to run in ring 3.

Xen provides many mechanisms for communication. The most primitive is the *hypercall*, which is an invocation of the hypervisor by a guest VM. A hypercall is analogous to a system call in conventional operating systems, and is also implemented using a software trap to transfer control to the hypervisor. Hypercalls are leveraged to build higher level communication primitives. In particular, in order to facilitate inter-VM communication, Xen provides *event channels* and a mechanism to establish shared memory regions [4]. Shared memory can be established via the *grant table* interface or using *direct foreign mappings* [14].

Event channels are a lightweight event delivery mechanism that may be used both to virtualise interrupts and enable communication between VMs. They may be used to send only a single bit of information, and are typically employed to synchronise producers and consumers in virtual device drivers.

A grant reference is the index of an entry in a VM-owned grant table. Each entry contains the physical address of the page to be granted, the ID of the VM to which access is granted, and access control flags. When the grantee wants to map a granted page, it makes a hypercall, specifying the grant reference, granting VM ID, and instructions for where to map the granted page.

Direct foreign mappings are only possible in a privileged VM, typically only Dom0, and are used for platform management. A privileged VM can request that the hypervisor maps a specific physical frame of memory from a specific VM into one of its page tables. It is possible to make direct foreign mappings into both the kernel and user-space. User-space mappings are made by a privileged command driver (`privcmd`), the use of which is typically restricted to the administrator using file permissions.

2.2 Building a new VM

In the Xen architecture, the creation of new VMs (known as *domain building*) is carried out by management software in Dom0, which uses its special privileges in order to access the address space and processor state of the new VM. The original design decision was that the domain building code is relatively complicated and should be omitted from the hypervisor for reasons of compactness and robustness [4]. However, the delegation of this task to user-space processes exposes the special privileges to all user-space processes in Dom0, which has implications for the trusted computing base (see §2.3). In this subsection, we describe the process of building a paravirtualised VM. Thanks to paravirtualisation, many of the low-level steps involved in a conventional x86 OS boot process (such as interactions with the BIOS, and the use of 16-bit real mode) can be elided.

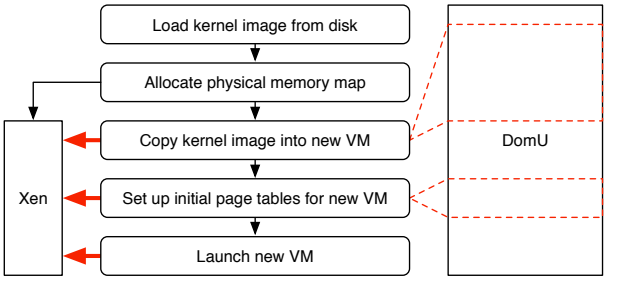


Figure 1. Outline of the steps involved in building a new VM (DomU). Privileged operations are shown as bold arrows; direct foreign mappings are shown as dotted lines.

The main steps in the domain building process are shown in Figure 1, with privileged operations (as defined later in §3.1) shown as bold arrows. Firstly, the kernel image (and, optionally, an initial ramdisk) are loaded from disk into the domain builder's memory. The image is parsed in order to extract the executable code region, and obtain any parameter values. Following this, the builder requests that the hypervisor allocates physical memory for the new VM. This is not depicted as a privileged operation, because it does not affect the confidentiality or integrity of any VM, but it can lead to denial of service, so its use should be restricted. The kernel image is then loaded into the new VM, using a direct foreign mapping. Following this, the initial page tables for the new VM are calculated and written into the new VM, using multiple direct foreign mappings. Finally, the new VM is launched, by setting the virtual CPU registers to a start-of-day configuration, and unpausing the VM.

The domain builder is implemented by the `xc_linux_build` function and its callees in `libxc`, a C library that provides low-level control over VMs. It is implemented in an object-oriented fashion, passing a `struct xc_dom_image` object between steps to communicate intermediate results. Direct foreign mappings are implemented by the `xc_map_foreign_range` function, and setting the virtual CPU registers is performed by the `xc_set_vcpucontext` function. The `libxc` library is linked into the `xend` management daemon, which is implemented in Python. `xend` implements the Xen API, which is an XML-RPC-based API that several tools use to manage a Xen-based system [35].

2.3 The TCB of a guest VM

We analyse the trusted computing base (TCB) of Xen for two reasons. Firstly, it is received wisdom that a smaller TCB corresponds to more trustworthy code [15, 17, 30]; therefore demonstrating the large size of the TCB gives motivation to our work. Secondly, it is necessary to enumerate and measure the TCB in order to build trusted virtualisation systems, which we discuss in Section 5.

The term "Trusted Computing Base" is defined variously in the literature, so it is useful to agree on a definition at the outset of this discussion. Hohmuth *et al* refer to "the set of components on which a subsystem S depends as the *TCB of S*." [17]. We refine this and define the TCB as "the set of components which a subsystem S trusts not to violate the security of S". We draw this distinction because, although S may depend on the piece of code which, for

example, loads it from disk into memory, that piece of code may be unable to violate the confidentiality or integrity of S, and therefore should not be considered part of the TCB.

We now turn to analysing the TCB of a guest virtual machine. The TCB must include Xen itself, because the hypervisor runs in a more privileged processor ring than the guest, and so it may access all memory and processor state. The Dom0 kernel is included in the TCB, because Dom0 is privileged, and the kernel can make privileged hypercalls. The domain builder process must also be included in the TCB, because it writes to the address space of the guest VM and configures its processor state: if it should malfunction, the security of S could be compromised. The domain builder code is dynamically linked into the Python interpreter that runs `xend`, so `xend` and the interpreter must be included in the TCB. However, the domain builder accesses the guest address space using the privileged command driver, which is exposed to all processes that run with administrator privileges in Dom0. Therefore, all such processes must be included within the TCB.

This poses a challenge when building trusted virtualisation systems. A trusted computing system relies on a chain of integrity measurements that inductively guarantees system integrity [3]. Therefore, it is necessary to communicate the integrity of the physical platform to each VM. However, if arbitrary software can be run in Dom0 at the discretion of the platform administrator, the integrity chain may be undermined at any point while a VM is running. For example, the integrity of the virtual platform may be used to decrypt a secret, after which the TCB may be altered by running some malicious software in Dom0. The secret now exists as plaintext in the memory of the guest virtual machine, and the malicious software may read it by performing a direct foreign mapping of the relevant page frames. We discuss this example in more detail in Subsection 5.1.

Therefore, we have shown that the TCB of a guest virtual machine running on Xen is both indefinitely large and dynamic. This raises questions about the robustness of such a system, and further precludes the use of Xen for trusted virtualisation, where the system administrator is not trusted.

3. Selecting the TCB

The goal of our disaggregation is to reduce the TCB of a Xen-based system. Therefore, in order to make an appropriate partition of the code base, it is necessary to determine which sections of code must be trusted, and which may be untrusted. In this section, we provide definitions that motivate our discussion of the TCB of a system (Subsection 3.1). We then argue that the "lines of code" metric for determining trustworthiness is insufficient, and introduce two new criteria (Subsection 3.2). Finally, we illustrate the use of these definitions for our disaggregation of the domain builder in a Xen-based system (Subsection 3.3).

3.1 Definitions

We want to be able to identify trusted and untrusted code. Therefore, we make the following definitions:

Privileged Operation An operation that can be used to subvert the confidentiality or integrity of a system or subsystem. (This differs from the definition of a privileged operation in Xen, which is broader, and includes all management tasks, such as creating and destroying VMs.)

Sanitising Operation An operation that checks that all potentially-untrusted inputs it receives are valid.

Invokable Operation An operation that may be invoked (from a particular position in the code base) without indirection via a sanitising operation. An operation is invokable from point p if it is implemented in the same address space as p and in the same or a less-privileged protection ring. An operation is also invokable from point p if there is a call path to that operation that does not include a sanitising operation.

Using these the definitions, it is clear that privileged operations must be implemented in a separate address space and/or protection ring from untrusted code. The TCB may alternatively be defined as the set of code positions for which any privileged operation is an invokable operation.

3.2 Criteria

It is commonly held that the trustworthiness of a TCB is a function of the number of lines of code that it comprises [15, 17, 30, 31]. This assumption is typically based on studies that relate software size to the number of bugs [5, 24]. Furthermore, it is attractive because it yields a metric that can be used to compare solutions. Whilst we accept that a smaller code base can be easier to comprehend, we believe there are scenarios in which *adding* code can improve trustworthiness. Therefore, in this section, we introduce two new criteria for evaluating the trustworthiness of a TCB: the size of the interface and the size of the TCB state space.

As we have stated in the previous subsection, it is necessary to sanitise the untrusted inputs to trusted, privileged code. As the number of interface functions in the TCB increases, so does the amount of sanitising code. Similarly, a less constrained type (e.g. a pointer to a region of memory that is used as an input) may require more complicated sanitisation than a more constrained type (e.g. an integer ID value).

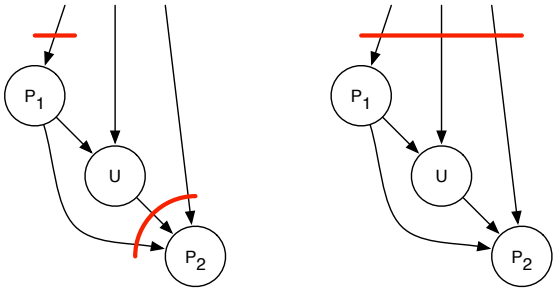
An error in the sanitising code is likely to allow a maliciously-constructed input to compromise the TCB. For example, Shankar *et al* observe a large number of security vulnerabilities that result from using an unsanitised string as the first argument to `printf` [29]. Therefore, minimising the complexity of the sanitising code is more important than doing the same for other trusted code. Basili and Perricone revealed the unexpected result that, as the size of a module increases, its error rate decreases [5]. They posit that this may be due to a uniform distribution of "interface errors" across all modules. This agrees with our intuition that the interface should be minimised, as interfacing code is the main source of critical errors. Furthermore, an increase in the size of the TCB should be tolerated, if it reduces the size or complexity of the interface.

Our second criterion is the size of the state space. We suggest that, where possible, TCB interface functions should be atomic operations, and, if they modify global state, this should not have an effect on the *behaviour* of subsequent invocations. In particular, we discourage a TCB design whereby it is necessary to pair operations together (such as lock and unlock operations), or where there is an assumed ordering to the operations. As such operations proliferate, the number of possible test cases increases factorially, and the likelihood of making an obscure error also increases. Therefore, if it is possible to create an atomic operation by including some previously-untrusted code in the TCB, we would prefer this to a smaller TCB.

We can illustrate our two criteria using the following simple example:

1. Invoke privileged operation P_1
2. Perform untrusted calculation, U, based on the result of P_1.
3. Invoke privileged operation P_2, based on the results of P_1 and U.

Should U be implemented inside or outside the TCB? Its size (number of lines of code) and ability to run arbitrary instructions should first be considered.



(a) U outside TCB (b) U inside TCB

Figure 2. Location of data paths (black arrows) and sanitising code (bold lines) depending on the position of U relative to the TCB.

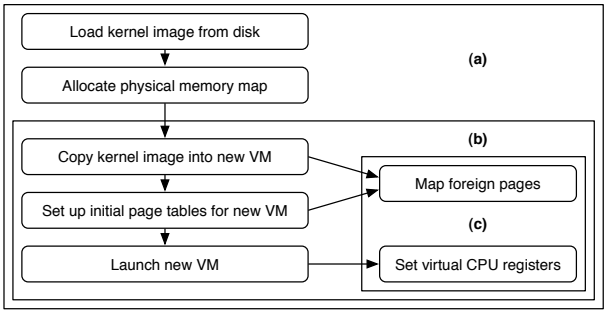


Figure 3. Three possible TCBs for the domain builder. **N.B.** This diagram only shows which parts of the domain builder (from `libxc`) are part of the TCB, and omits lower-level code for clarity.

Figure 2 illustrates the data paths between the three operations, and the location of sanitising code for each case. When U is outside the TCB (Figure 2(a)), it is necessary to sanitise the input to P_1, the external input to P_2 and the output of P_1 and U. The output of P_1 is generated by the TCB, and so we know that it is valid; however, it has left the TCB and must be sanitised, because the untrusted code could have modified it. If U is moved inside the TCB (Figure 2(b)), it is only necessary to sanitise the external inputs to P_1, U and P_2. The outputs of P_1 and U need no longer be sanitised, because they never leave the TCB.

In terms of the state space, our decision depends on whether P_1 and P_2 have an effect on the state of the system. For example, are all calls to P_1 and P_2 paired? What happens if P_2 is invoked before P_1, or if P_1 is invoked without invoking P_2? What happens if another operation is invoked between the invocations of P_1 and P_2? An incorrect implementation of mutual exclusion in this context could lead to liability inversion or deadlock. If the $P_1; U; P_2$ sequence were implemented as an atomic operation within the TCB, these problems would not arise.

Obvious tensions exist between the criteria that we have described here. For example, as one incorporates all possible atomic sequences of privileged operations into the TCB, the size of the TCB interface grows, and the number of lines of code in the TCB also grows. As the size of the TCB interface is reduced, the number of lines of code in the TCB may grow. Neither of our suggested criteria are intended as a panacea for choosing an appropriate TCB, but we believe that they should be considered as part of the whole picture when designing a secure system.

3.3 Redrawing the TCB of a guest VM

In this subsection, we present our redrawn TCB for a disaggregated Xen-based system. We concentrate on the domain builder, as it is the fundamental operation that has led to a large TCB for Xen-based systems. In order to reduce the size of the TCB, it is necessary to ensure that privileged operations are not invokable from untrusted code. In a VMM-based system, this is possible by moving the privileged operations to another VM, and interposing a sanitising operation on all call paths between untrusted and trusted code. In this subsection, we concentrate on the functionality that is disaggregated; Section 4 concentrates on the practical implementation of disaggregation.

By placing a sanitising operation (see §3.1) between Dom0 user-space and the privileged operations in the domain builder, it is possible to remove Dom0 user-space from the TCB. Therefore, since only the hypervisor, Dom0 kernel and part of the domain builder remain in the TCB, it is possible to measure the TCB at boot and be confident that it will not change. This is crucial for trusted virtualisation, which we discuss further in Section 5. The remaining question is where to place the sanitising code.

Figure 3 shows the domain builder process, annotated with three possible TCB demarcations. We determined that TCB (a), which places the entire domain building process (as described in Subsection 2.2) inside the TCB, was the best solution. The interface between trusted and untrusted code is the `xc_linux_build` function, which is called by the Python-based tool stack to perform the low-level build process (as opposed to the high-level configuration of virtual devices, which remains in Dom0).

This approach has two main advantages. First, the size of the interface is reduced to a single function, which has no arguments that are used as arguments to privileged operations (excluding the VM ID, which is straightforwardly validated). Therefore, the necessary amount and complexity of sanitising code is minimised, and, since errors here are critical, this improves our confidence in the security of the partitioning. Furthermore, the interface to the TCB exports a single, atomic function, which encapsulates all of its state either on its stack, or on the heap in regions referenced only by stack pointers. Therefore the TCB is effectively stateless, and it is not possible to use the interface to manipulate the TCB into an unexpected state.

Note that we include the "Load kernel image from disk" step inside the TCB. At first, it appear that we must trust the system call, file system and block device code that fulfil such a request. However, it is possible to include this step inside the TCB by using a *trusted wrapper*, as described by Singaravelu *et al* [30], around the file-handling code. We use an approach called "I/O forwarding", which is described in Subsection 4.1.

We now consider the alternative approaches depicted in Figure 3. The domain builder process incorporates five main steps, of which three use privileged operations. TCB (b) places only these three steps inside the TCB, which is therefore smaller (in lines of code) than TCB (a). The TCB is then responsible for copying the kernel image into the new VM, building the initial page tables, and launching the VM. However, the second step in the build process ("Allocate physical memory map") generates a potentially-large amount of data that must be validated before it is used inside the TCB. These data originate in the (trusted) hypervisor, and ideally would not require validation. The process of validation would be so similar to the original process of retrieval, that it is redundant to perform this step outside the TCB.

Minimising the number of lines of code yields TCB (c) in Figure 3. Here, the TCB includes only the implementation of "map foreign pages" and "set virtual CPU registers", and the necessary sanitising code that is required to ensure that these functions are not abused. However, because of the flexibility of these functions, the

sanitising code would have to be almost as complex as the untrusted code from which it has been partitioned[1]. As we have stated, the amount and complexity of sanitising code should be minimised, as any error here could result in a security vulnerability.

In conclusion, choosing our disaggregation gives a useful example of the criteria for a trustworthy TCB that we presented in the previous subsection. We chose the largest of the three TCBs because this minimises the size of the interface (and hence the amount of sanitising logic), and introduces only a single, atomic operation to the TCB interface.

4. Implementation

In this section, we describe the implementation of the reduced TCB that we chose in the previous section. The principal component is the domain builder service (Subsection 4.1). In order to support the domain builder, we also implemented a lightweight remote procedure call mechanism (Subsection 4.2) and a user-space driver for accessing granted memory (Subsection 4.3).

4.1 The domain builder

We ported the domain builder to a minimal paravirtualised operating system, called MiniOS. MiniOS was developed as an example of how to implement the various features of paravirtualisation in Xen, such as event channels, granted pages and hypercalls. It includes no physical device drivers or file systems. A rudimentary virtual network driver is included, but there is no TCP/IP stack in the standard version. The small code base and the absence of many common exploit vectors make MiniOS a suitable basis for building a trusted service on Xen. However, the lack of functionality poses some problems, which we will address in this section.

The main challenge when implementing the disaggregated domain builder was that the existing code relies on file system access, in order to load kernel images and, optionally, initial ramdisks for a new guest. However, it is difficult to load a file into our domain builder, because, as we have already remarked, MiniOS includes no physical device drivers, file systems or networking stack. We have rejected adding any of these to MiniOS, because they are non-essential, complex pieces of code, which constitute an unnecessary burden on the TCB. Instead, we forward the file system calls, such as `fopen` and `fread`, to Dom0. Dom0 runs an I/O forwarding service, called `vfsback`, which receives the requests and executes them locally, returning the results to the domain builder.

Figure 4 illustrates how the various processes communicate in the disaggregated domain building process. To begin the process, `xend`, through the linked `libxc`, makes an IVMC request (described in §4.2) to the domain builder (known as *DomB*), specifying the filenames of the kernel and initial ramdisk, the domain ID and the amount of memory to be allocated. The domain builder loads the kernel and initial ramdisk by forwarding I/O calls to the `vfsback` daemon in Dom0. The domain builder allocates a physical memory map as before, then installs the kernel image, initial ramdisk and initial page tables in the new VM. Finally, the domain builder launches the new VM, and returns control to `xend` in Dom0.

4.2 Inter-VM Communication

In order for the management software to control the domain builder, it is necessary for Dom0 and DomB to communicate. However, as we do not want to introduce a TCP/IP stack into MiniOS, it is not possible to use existing network-based mechanisms to perform this communication. Instead, we have borrowed the concept of inter-process communication (IPC) from the microkernel community,

[1] It must effectively ensure that the sequence of invocations and parameters correspond to the correct domain-building procedure.

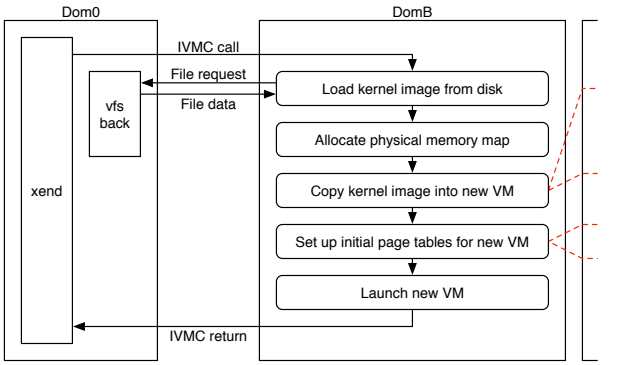


Figure 4. Communication relationship between the control software in Dom0 and the disaggregated domain builder (DomB). The IVMC mechanism is described in §4.2.

and implemented synchronous procedure call semantics using only the facilities present in a VMM-based system.

IPC is often highlighted as a method of communication between trusted and untrusted components in a microkernel-based system [17, 30]. The use of a VMM as the basis of a secure system is frequently criticised, because VMMs lack, as a primitive, any notion of inter-process communication. Indeed, it is suggested [17] that the only way to emulate IPC is using a remote procedure call (RPC) protocol [8] over a virtual network. Whilst this approach is feasible between full operating systems, it is not suitable for small-TCB applications that might lack a networking stack. This gives rise to our inter-VM communication (IVMC) implementation.

What do we mean by inter-VM communication? Clearly, a combination of the mechanisms in Subsection 2.1 can be used by two VMs in order to communicate: in fact, this is how the virtual devices in Xen are implemented [14]. We assume that the IPC advocated by Hohmuth *et al* is "message transfer between threads" [21], using which synchronous procedure calls can be developed.

We were inspired by the use of an Interface Definition Language (IDL) in the applications built on the L4/Fiasco microkernel [27]. The Dresden IDL Compiler (DICE) obviates the need to write low-level communication code, and presents a C or C++ procedural interface for IPC [1]. It contains front-ends for parsing DCE and CORBA IDL, and back-ends that generate code for the L4/Fiasco API and Linux Sockets. Although the interface to the domain builder is relatively small and simple, an IDL compiler enables rapid experimentation with different interfaces, and can be used in the disaggregation of other functionality. Furthermore, it raises the possibility of portable applications running on both Xen/MiniOS and L4/Fiasco.

We developed a new back-end for DICE that can be used on a Xen system. The generated code uses a generic communication object that encapsulates a shared buffer and user-provided notification function. The shared buffer is created by the client allocating a page and using its grant table to grant access to the server; the notification is handled using an inter-VM event channel.

An IVMC end-point is identified by the ID of the server VM, a grant reference for the shared buffer, and the ID of the inter-VM event channel. These numbers may potentially change between boots of the physical platform, and so some mechanism is required to bind the name "DomB" to the correct endpoint. In the simplest case, the identity of the DomB virtual machine is known to the VMM at boot. DomB can then make a hypercall to set the grant reference and event channel ID after it has booted. Dom0, seeking

the endpoint details, can then make a hypercall to discover these details. Since the VMM is part of the TCB, the correctness of the details can be trusted.

4.3 The user-space grant table device

In order to complete our disaggregation, and disable the privileged operations in Dom0, it is necessary to remove all uses of the direct foreign mapping operation from Dom0 user-space. The grant table concept (see §2.1) provides a mechanism for controlled sharing – akin to capabilities – but this has hitherto only been available in the kernel. Therefore, we implemented a user-space grant table device (`gntdev`), which provides a means of mapping one or more granted pages directly into the user-space of a virtual machine.

We implemented gntdev as a Linux kernel driver, which is manipulated using the same `mmap` and `ioctl` system calls that are used with the `privcmd` driver. In addition, we provide library functions in `libxc` for performing common operations, such as mapping a single grant reference from another VM. These library functions are used to replace instances of the direct foreign mapping operation, and to map the shared buffer used for IVMC.

The principal challenge in developing `gntdev` was the transience of user processes, when compared with virtual machines. It was necessary to make a slight modification to the Linux virtual memory subsystem, adding a hook that is called when a page table is cleared, in order to unmap grant references correctly when a process crashes. Furthermore, it was necessary to address the case where pages are granted by a user-space process, mapped by another VM, and not unmapped before the process dies. In this case, we ensure that the granter operating system does not reuse such pages until they are unmapped, in order that no information about other processes is leaked.

5. Trusted virtualisation

Our work in this paper lays a foundation for *trusted virtualisation*. By bounding the TCB of a Xen-based system, we have made it possible for the hypervisor and Dom0 to take their place in a chain of trust that extends from the hardware Trusted Platform Module (TPM) into each VM. In this section, we address the problems with the status quo in trusted virtualisation (Subsection 5.1) and present our design for a trusted VMM using disaggregation (Subsection 5.2).

5.1 Problems with the status quo

Xen already incorporates virtual TPM (VTPM) software that may be used for trusted virtualisation. It was developed with the stated requirement of a "Strong association of the Virtual TPM with the Underlying TCB" [7]. The VTPM is implemented as a virtual driver, with a front end that matches the interface of a physical TPM, and a back end in Dom0 that performs TPM emulation and multiplexing of requests to the real physical TPM. For additional security, the authors show that the back end can be implemented on a secure coprocessor.

Two issues with the implementation of the virtual TPM undermine its effectiveness, however. The first is that there is no communication between the `libxc` domain builder and the VTPM software. Therefore, any attempt to measure the kernel and initial ramdisk, then store it in the VTPM, is vulnerable to a time-of-check-to-time-of-use (TOCTOU) attack [9].

The second issue is that the TCB may change at any time, due to the administrator running a new executable in Dom0. The issue is not mitigated by using a Dom0 operating system that is modified to measure all of its executables as they are run, such as that suggested by Sailer *et al* [28]. Consider the following series of events:

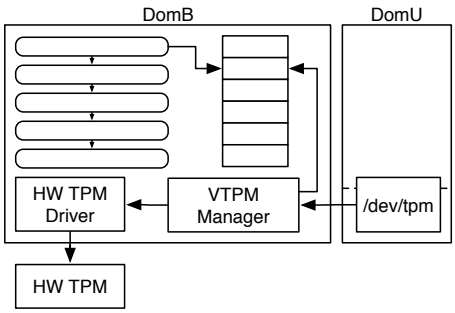


Figure 5. Design of a trusted virtualisation system, using the disaggregated domain builder (DomB).

1. The hypervisor and Dom0 are measured on boot, and the platform is brought into a trustworthy state. A new VM, DomU, is created.
2. DomU decrypts some data, based on the virtual platform configuration. The data now exists as plaintext in DomU's memory.
3. A new executable, `memorysniffer`, is run in Dom0, which modifies the physical platform configuration (and, hence, the virtual platform configuration of DomU) to a no-longer-trustworthy state.
4. `memorysniffer` maps each page of DomU's physical memory until it finds the unencrypted secret.

At no point in the above steps does any component malfunction, yet it is possible for Dom0 to compromise the confidentiality of DomU. A similar argument can be made for integrity.

5.2 Trusted VMM using disaggregation

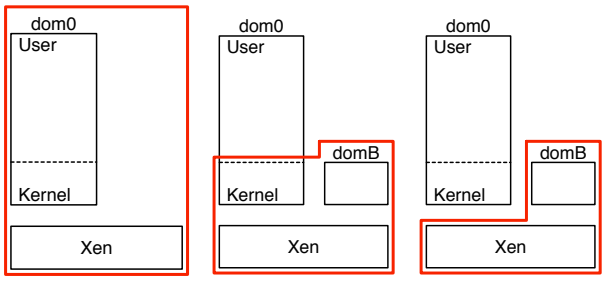
In this subsection, we present an improved design for virtualising the TPM, using the disaggregated domain builder.

Figure 5 illustrates our design for the trusted virtualisation system. DomB is expanded to include a VTPM Manager, a hardware TPM driver and virtual platform configuration storage. Anderson *et al* previously ported the VTPM Manager to MiniOS [2]. The VTPM front end (`/dev/tpm`) is unchanged, but now connects to the VTPM Manager in DomB.

The build process is modified so that an integrity measurement of the kernel, initial ramdisk and any configuration options is taken after the relevant files have been loaded into DomB's memory. At this point, a new VTPM instance is created for the domain, and the measurements are stored in one or more virtual PCRs. By performing the measurement here, the VTPM is not vulnerable to a TOCTOU attack, as the kernel that is measured is guaranteed to be the same as the one that is loaded. As in the existing VTPM implementation, the first nine physical PCRs are mapped into all virtual TPMs. In order to associate the virtual platform with the physical platform, a driver for the hardware TPM is included in DomB, and Xen gives DomB exclusive control over the device.

The physical platform configuration is measured by a trusted bootloader, such as the OSLO bootloader developed by Bernhard Kauer [18]. This loads and measures the hypervisor, Dom0 kernel and domain builder, and stores the measurements in the hardware TPM.

It is possible to refine this model in order to allow dynamic kernel module loading in Dom0. The Dom0 kernel must be modified to make an IVMC call to DomB before a new module is loaded. The IVMC call would contain a measurement of the loaded module, and DomB would update the physical platform configuration on Dom0's behalf. DomB would only perform the extension, and



(a) Existing (b) Disaggregated (c) Ideal

Figure 6. Illustration of the TCB contents for the existing, disaggregated and ideal Xen-based systems.

	Component	1000s of lines of code		
		C	ASM	Python
	Hypervisor	98	3	–
	Dom0 kernel	1500	9.6	–
	Dom0 drivers	$\leq$ 2400	$\leq$ 2.6	–
+	DomB	9.2	0.5	–
−	libc	690	15	–
−	Python	220	–	140
−	libxc	9.9	–	–
−	xend	2.4	–	17
+	Added	9.2	0.5	–
−	Removed	920	15	160

Table 1. Changes to the TCB between the existing and disaggregated approaches. Rows beginning with + indicate where code was added to the TCB; rows beginning with − indicate where it has been removed.

return a successful response, if no guest VMs were currently running on the physical machine. Otherwise, loading the module could potentially compromise the running VMs, so the extension would not be performed, and Dom0 would be instructed not to load the module. As the Dom0 kernel is part of the TCB, it could be trusted to respect the decision of DomB.

6. Evaluation

In this work, we aimed to improve the trustworthiness of a Xen-based system. In this section, we evaluate our solution both quantitatively and qualitatively, by enumerating the contents of the TCB before and after disaggregation (Subsection 6.1), performing a security analysis of a disaggregated system (Subsection 6.2), and discussing the limitations of our approach (Subsection 6.3).

6.1 Reducing the TCB

Figure 6 shows three illustrations of the TCB in a Xen-based system, representing the existing, disaggregated and ideal cases.

The existing case (Figure 6(a)) shows the TCB of a system running the current version of Xen (version 3.1). The TCB of every guest VM includes the hypervisor, the Dom0 kernel, and all user-space code that the administrator runs in Dom0. This includes a Python interpreter that runs the `xend` control daemon, the `libxc` low-level control library, and a C library (`libc`).

The disaggregated case (Figure 6(b)) shows the current state of our work. The TCB of every guest VM includes the hypervisor, the Dom0 kernel, and a MiniOS-based version of the `libxc` low-level control library. The Dom0 kernel is included in the TCB because it is responsible for interaction with physical input/output devices; however, it no longer exports privileged operations to user-space. The `xend` control daemon and the Python interpreter are no longer part of the TCB, because they cannot be used to undermine the confidentiality of a VM, and any attempt to undermine integrity would be detected (see §6.2 for details).

Table 1 compares the TCB contents for the existing and disaggregated cases. The following software was analysed: Xen version 3.1 (for the hypervisor, Dom0 kernel (version 2.6.18-xen), Dom0 kernel drivers, libxc and xend), Python version 2.4.4 and glibc version 2.6. We used version 2.26 of the SLOCCount tool [34] to perform the analysis. We analysed only platform-independent code and platform-specific code for the `x86_32` architecture. We did not analyse testing code. The table does not include other user-space code in Dom0 that would be counted as part of the TCB in the existing version, but instead concentrates on a minimal Dom0 configuration.

In addition to the amount of code in the TCB, the size of the interface must be considered. The existing version includes the `privcmd` driver, which enables user-space processes run by the administrator to make privileged hypercalls. By removing this feature from the driver, it is no longer possible for user-space processes to perform a direct foreign mapping, or alter the virtual CPU state. This step removes Dom0 user-space from the TCB.

Figure 6(c) illustrates the ideal TCB. At present, this is not feasible. Several authors have commented that a malconfigured device that can perform Direct Memory Access (DMA) can access any part of physical memory [14, 15, 17], and its driver must therefore be included in the TCB. The introduction of Input/Output Memory Management Units (IOMMUs), which in effect create virtual address spaces for DMA, should help to mitigate this problem [6]. It will be necessary to add code that controls the IOMMU, either to the hypervisor, or in a small, trusted VM that runs beside DomB. Without an IOMMU, we must trust the physical device drivers, and since these run within a Linux kernel, we must trust the kernel itself.

6.2 Security analysis

In this subsection, we evaluate a disaggregated Xen-based system in terms of two security properties: *confidentiality* and *integrity*. We consider two unauthorised sources of attacks: the administrator of the physical platform (hereafter abbreviated to Dom0-admin), and other unprivileged VMs on the same physical host.

We take the definitions of confidentiality and (data) integrity from the NIST Handbook on Computer Security [23]:

Confidentiality "A requirement that private or confidential information not be disclosed to unauthorized individuals."

Integrity "[A] requirement that information and programs are changed only in a specified and authorized manner."

In our analysis, we use a physical host as the benchmark for our security properties. We do not consider the *availability* of a VM, because, if it were a physical machine in a managed data center, the machine could be powered down arbitrarily, and it is not possible for software to prevent this. Therefore, the `xend` operations of creating and destroying VMs are not considered privileged operations. Similarly, we assume that all input/output channels and secondary storage are insecure, because on a physical machine these could be intercepted by, for example, wiretapping a physical cable or transplanting a hard drive into another machine. Finally, we assume that encryption is unbreakable and we do not consider timing attacks in this analysis.

We describe the results of our analysis for the virtual CPU (6.2.1), physical RAM (6.2.2), secondary storage (6.2.3), network (6.2.4) and kernel image (6.2.5).

6.2.1 Virtual CPU

Each VM has one or more virtual CPUs (VCPUs), which are stored in a hypervisor data structure when not in use by the physical CPU. The confidentiality of a VCPU may be attacked by a hypercall that gets the contents of that VCPU. The integrity of a VCPU may be attacked by a hypercall that sets the contents of that VCPU.

Neither Dom0-admin, nor an unprivileged VM has access to either hypercall, and so neither can undermine the confidentiality or integrity of a VCPU. Only DomB can use the "set" hypercall, as part of the build process.

6.2.2 Physical RAM

In order to map a page of RAM from another VM, it is necessary to create a mapping to that page in the attacking VM's page table. However, all active page tables are protected by Xen, and any update that refers to a page that belongs to another VM must be made using a hypercall. The confidentiality of physical RAM may be attacked by mapping a page for read access. The integrity of physical RAM may be attacked by mapping a page for write access.

An unprivileged VM may only update its page table to include pages that it owns, or to which it has been granted explicit access. Therefore, it cannot undermine the confidentiality or integrity of physical RAM.

The Dom0 kernel may, in effect, access any page of physical memory, because it controls devices that may perform DMA from any address. However, the Dom0 kernel does not propagate this control to user-space; therefore Dom0-admin cannot use user-space software in Dom0 to undermine the confidentiality or integrity of physical RAM.

6.2.3 Secondary storage

The contents of secondary storage may be held directly on a physical backing store, or in a file in Dom0. Because the virtual block device back-end is typically implemented there, Dom0 requires read- and write-access to the secondary storage. The confidentiality of secondary storage may be attacked by inspecting the content of the physical backing store or file. The integrity of secondary storage may be attacked by overwriting the content of the physical backing store or file.

An unprivileged VM has no direct access to hardware, and cannot directly access a file in Dom0 that contains another VM's secondary storage. Therefore, it cannot undermine the confidentiality or integrity of secondary storage, unless aided by Dom0.

Clearly, Dom0-admin can read any unencrypted contents of secondary storage, and can make arbitrary changes. In order to protect confidentiality, it is necessary for the guest to encrypt its data using, for example, the dm-crypt API in Linux [32] or the BitLocker feature in Microsoft Windows Vista [22]. Keys can be protected using the sealing functionality of the virtual TPM. Encryption can also be used to preserve integrity, though this does not rule out a destructive attack on the data in secondary storage.

We note that the protection given to secondary storage is equivalent to that provided by a physical host, as a physical hard drive can be transplanted into an untrusted computer, in order to circumvent access control.

6.2.4 Network

The virtual network driver is implemented as a virtual split device that has a front end in the guest VM and a back end in Dom0. In the most common Xen network topology, the back ends are connected to a software bridge, which is then connected to the physical network interface. The confidentiality of network traffic may be attacked by attaching a packet sniffer to the virtual network. The integrity of network traffic may be attacked by modifying the bridge software to change the contents of packets as they are forwarded, or injecting false packets into the network.

An unprivileged VM will only see packets that are addressed to it, and it cannot send packets with false headers that appear to be from another VM, because Dom0 will reject them. Therefore, an unprivileged VM cannot undermine the confidentiality or integrity of the network.

As is the case for secondary storage, all network traffic passes through Dom0. A packet sniffer, run in Dom0, could therefore be used to undermine the confidentiality of the network. An encryption scheme, such as TLS [11] or IPSec [19] may be used to protect confidentiality. In order to modify packet data, it would be necessary to modify the Dom0 kernel code that controls the software bridge: since this is part of the TCB, it would be reflected in integrity measurements. However, encryption provides an additional defence against attacks on the integrity of the network. The use of Virtual Private Networking [16] software in the guest VM would be sufficient to protect the confidentiality and integrity of the network from Dom0-admin.

We note that the protection given to network connections is equivalent to that provided by a physical host, when it is connected to an untrusted network.

6.2.5 Kernel image

The kernel image (and optionally, the initial ramdisk) exist as unencrypted files in the Dom0 file system, which are transferred to the domain builder when a VM is created. We do not consider the confidentiality of a kernel image, because this is incompatible with an "open trusted computing" approach: it should be possible to inspect the contents of the kernel and be confident that it is not carrying out any malicious activities [20]. The integrity of a kernel image may be attacked by modifying the relevant file or files.

An unprivileged VM cannot access the file containing the kernel image unless it is granted access by Dom0, and it cannot intercept the IVMC channel used to transfer the file to DomB, because this channel uses explicit granted access from DomB to Dom0.

Dom0-admin may modify the contents of the kernel image. However, the trusted VMM architecture (see §5.2) stipulates that the integrity of the kernel image is measured before boot, and this measurement is stored in the virtual TPM. If Dom0-admin compromises the integrity of the kernel image, the guest will be able to detect this by performing attestation, or trying to decrypt a secret that has been sealed to the platform configuration.

6.3 Limitations

As discussed in Subsection 6.1, one major limitation of our approach is that the Dom0 kernel must be included in the TCB, and it is by far the largest TCB component. It is included in the TCB because it controls physical hardware that may perform DMA, and, without an IOMMU, may therefore read or write to any location in physical memory. Why, in that case, did we not include the domain builder code in the Dom0 kernel, and retain the aggregation of privileges for Dom0? We chose disaggregation because, when IOMMUs are commonly available, the disaggregated approach will facilitate a switch to the "ideal" TCB depicted in Figure 6(c), whereas there would be no advantage if the domain builder were integrated in the Dom0 kernel.

Duflot *et al* demonstrated that it is possible to use the System Management Mode on x86 computers to undermine security policies [12]. The exploit is possible from user-space, when running as the administrator, even when using a "secure" OS, such as OpenBSD. In order to mitigate the attack, user-space access to

video RAM must be disabled. In this case, it is not possible to use an X Server, which relies on this access. This is not a concern in data centre use cases, but it has severe implications for desktop virtualisation. We are currently investigating the use of a lightweight GUI, such as Nitpicker, which is sufficiently small to be included in the TCB [13].

We have analysed the Xen tool stack and found several tools that make use of the `xc_map_foreign_range` function from libxc. Most of these map single frames for communication with guest VMs: we have introduced "third-party grants" that enable the domain builder to insert entries in a new VM's grant table in order to share these pages. The user-space grant table device (see §4.3) may be used to perform the mappings. The save and restore functions use direct foreign mappings in order to copy the contents of a VM's memory to and from disk, respectively. These may be implemented in a disaggregated fashion, using the same basis as the domain builder. A final category of mappings are those used for debugging: we do not attempt to enable these, because they would undermine the confidentiality and integrity guarantees that our approach makes.

7. Related Work

Disaggregation is similar to the work carried out by Singaravelu *et al* on the Nizza architecture [30]. This work involved extracting the security-critical components ("AppCores") of several legacy applications, and running these within a kernelised TCB, which runs on top of the L4 microkernel. Communication between the trusted and untrusted components was implemented using L4 interprocess communication (IPC). However, the intention of this work was to protect the security sensitive parts of an application from the much larger, untrusted segment. The authors did not attempt to protect applications from a malicious administrator.

Disaggregation is also an example of privilege separation, described by Provos *et al* [25]. They describe an approach whereby an application containing privileged and unprivileged components is divided so that each part runs with the least necessary privilege, and analyse their approach on OpenSSH. In their security analysis, they concentrate on minimising the number of lines of code in the privileged components. By contrast, we introduce additional criteria for qualitatively evaluating the TCB in Subsection 3.2, and base our separation upon these, in addition to the number of lines of code.

Hohmuth *et al* criticise the security claims made of virtual machine monitors and suggest using a small kernel, inter-process communication and wrappers around untrusted code, in order to reduce TCB size [17]. To this end, they demonstrate the ability to run legacy applications on an L^4Linux server, which runs on top of the L4 microkernel. They propose a new point on the VMM-microkernel continuum, namely "VM-enabled microkernels", which they conflate with "VMMs with microkernel-like features". We believe that the two are actually different, and, in this paper, we present an example of the latter. In particular, Hohmuth *et al* hold that, in a paravirtualisation-based VMM, such as Xen, "IPC needs device emulation", which thereby hinders the adoption of a disaggregated approach. In Subsection 4.2, we demonstrate that this is not the case.

The Terra architecture for Trusted Computing presents the concept of "closed box" virtual machines that cannot be inspected or altered by another virtual machine running on the same platform [15]. Such closed boxes could be provided by, for example, a online game manufacturer, who wants to ensure that players are unable to cheat whilst playing the game. Although the authors present an architecture that could enable closed box operation, they do not provide details for how these may be isolated from the host operating system (analogous to Dom0 on a Xen platform). Indeed, they acknowledge that their implementation, using VMWare GSX Server and a Debian Linux host OS, is not "suitably high assurance for a real TVMM [Trusted VMM]". We develop their work further by suggesting the mechanisms that can be used to develop a trustworthy VMM.

Fraser *et al* demonstrated the use of virtual machine isolation for providing safe and reliable access to hardware devices [14]. In their system, each hardware device and the least I/O privileges required to access it are assigned to an individual Xen VM. Therefore, the robustness of the system is improved when faced with a misbehaving driver. "I/O Spaces" are used to disaggregate the I/O port and memory privileges that were previously assigned to a monolithic Dom0. However, this work differs from ours in that it concentrates on driver isolation, and does not attempt to shield guests against a malicious administrator.

Two separate projects have implemented socket-like communications between virtual machines in a Xen-based system. Zhang *et al* created "XenSocket", which provides a "high-throughput interdomain transport" between Linux-based VMs [36]. Anderson *et al* have developed an inter-domain communication (IDC) library, which may be used to communicate between Linux- and MiniOS-based VMs. In addition, they have ported a minimal C library and development toolchain to MiniOS, which enables trusted applications to be built straightforwardly [2]. These approaches differ from ours, because they implement communication in the Linux kernel, whereas we use the user-space grant table device to implement communication in user-space. Furthermore, both approaches assume a trusted Dom0.

8. Conclusions

In conclusion, we have demonstrated that it is possible to improve the security of a virtual machine running on the Xen virtual machine monitor, using a process of disaggregation. Our approach yields a measurable TCB that can be used with established trusted computing techniques. In redrawing the TCB, we have asserted that the number of lines of code is not the sole determinant of trustworthiness in a TCB, and introduced two new criteria – interface size and state space size – for judging trustworthiness. We applied these criteria to the selection of a TCB for disaggregation.

In order to implement our solution, we have made three main contributions. The principal contribution is the disaggregated domain builder service, based on our TCB design. This domain builder is a small, trusted service, which is given the necessary privileges in order to build new virtual machines. It therefore obviates the need for a monolithic management domain (Dom0) to have full privileges. Our other contributions supported the development of the domain builder, but can also be used for other purposes. We developed an inter-VM communication mechanism and associated IDL compiler. This enables developers to create lightweight code that communicates between Xen guest VMs without using a networking stack, which is particularly useful when developing trusted services that run on a minimal operating system (such as MiniOS). The IDL dialect is compatible with that used for L4/Fiasco, and we hope that this will lead to an exchange of solutions between the VMM and microkernel development communities. Finally, we developed a user-space grant table driver, which replaces the use of direct foreign mappings in Dom0. The grant table permits controlled sharing, and the new driver may be used in future to develop secure user-space management software for Xen guest VMs.

Our design has been informed greatly by developments in microkernels, especially the privilege separation work that has been carried out using the L4 microkernel [17, 30]. We borrow the concept of IPC between protection domains, and, indeed, we make use of the same IDL compiler front-end that is used to integrate applications on L4 [27]. We believe that our approach, "VMMs with

microkernel-like features", leads to a satisfactory combination of security and functionality.

Acknowledgments

We would like to thank our colleagues for their comments and suggestions. This work was partially supported by EPSRC Grant reference EP/D020158/1 (XenSE), and the Open Trusted Computing project of the European Commission Sixth Framework Programme.

References

[1] R. Aigner. DICE User's Manual. Technical report, Technische Universität Dresden, 2007. `http://os.inf.tu-dresden.de/dice/manual.pdf`.

[2] M. J. Anderson, M. Moffie, and C. I. Dalton. Towards Trustworthy Virtualisation Environments: Xen Library OS Security Service Infrastructure. Technical Report HPL-2007-69, Hewlett-Packard Development Company, L.P., April 2007.

[3] W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 1997.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on operating systems principles*, pages 164–177. ACM Press New York, NY, USA, 2003.

[5] V. R. Basili and B. T. Perricone. Software errors and complexity: an empirical investigation. *Commun. ACM*, 27(1):42–52, 1984.

[6] M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L. V. Doorn, A. Mallick, J. Nakajima, and E. Wahlig. Utilizing IOMMUs for Virtualization in Linux and Xen. In *Proceedings of the 2006 Ottawa Linux Symposium*, 2006.

[7] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: virtualizing the trusted platform module. In *Proceedings of the 15th USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, 2006. USENIX Association.

[8] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[9] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[10] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.

[11] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246, IETF, Jan. 1999.

[12] L. Duflot, D. Etiemble, and O. Grumelard. Using CPU System Management Mode to Circumvent Operating System Security Functions. In *Proceedings of the 7th CanSecWest conference*, 2001.

[13] N. Feske and C. Helmuth. A nitpicker's guide to a minimal-complexity secure GUI. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society.

[14] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, 2004.

[15] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 193–206. ACM Press New York, NY, USA, 2003.

[16] B. Gleeson, A. Lin, J. Heinanen, G. Armitage, and A. Malis. A Framework for IP Based Virtual Private Networks. RFC 2764, IETF, Feb. 2000.

[17] M. Hohmuth, M. Peter, H. Härtig, and J. Shapiro. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th ACM SIGOPS European workshop: beyond the PC*. ACM Press New York, NY, USA, 2004.

[18] B. Kauer. OSLO: Improving the Security of Trusted Computing. In *Proceedings of the 16th USENIX Security Symposium*. USENIX Association, 2007.

[19] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, IETF, Dec. 2005.

[20] D. Kuhlmann, R. Landfermann, H. Ramasamy, M. Schunter, G. Ramunno, and D. Vernizzi. An Open Trusted Computing Architecture: Secure virtual machines enabling user-defined policy enforcement. Technical report, OpenTC consortium, 2006. `https://secure.opentc.net/otc_HighLevelOverview/OTC_Architecture_High_level_overview.pdf`.

[21] J. Liedtke. On micro-kernel construction. *ACM SIGOPS Operating Systems Review*, 29(5):237–250, 1995.

[22] Microsoft Corporation. BitLocker Drive Encryption, 2007. `http://technet.microsoft.com/en-us/windowsvista/aa905065.aspx`.

[23] National Institute of Standards and Technology. An Introduction to Computer Security: the NIST Handbook. Technical Report 800-12, National Institute of Standards and Technology, October 1995.

[24] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. *SIGSOFT Softw. Eng. Notes*, 27(4):55–64, 2002.

[25] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[26] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: Accountable execution of untrusted programs. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, page 136, Washington, DC, USA, 1999. IEEE Computer Society.

[27] L. Reuther, V. Uhlig, and R. Aigner. Component Interfaces in a Microkernel-based System. In *Proceedings of the 3rd Workshop on System Design Automation (SDA)*, March 2000.

[28] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th USENIX Security Symposium*, pages 223–238, 2004.

[29] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[30] L. Singaravelu, C. Pu, H. Hartig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of EuroSys 2006*, 2006.

[31] A. Tanenbaum, J. Herder, and H. Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.

[32] (Unattributed). dm-crypt - a device-mapper crypto target, 2007. `http://www.saout.de/misc/dm-crypt/`.

[33] (Unattributed). TPM Main Part 1 Design Principles. Technical report, Trusted Computing Group, 2007. `https://www.trustedcomputinggroup.org/specs/TPM/mainP1DPrev103.zip`.

[34] D. A. Wheeler. SLOCCount, 2007. `http://www.dwheeler.com/sloccount/`.

[35] XenSource. XenApi - Xen Wiki, 2007. `http://wiki.xensource.com/xenwiki/XenApi`.

[36] X. Zhang, S. McIntosh, P. Rohatgi, and J. Griffin. XenSocket: A high-throughput interdomain transport for VMs. In *Proceedings of Middleware 2007*, Secaucus, NJ, USA, 2007. Springer-Verlag New York, Inc.