

Separation Virtual Machine Monitors

John McDermott, Bruce Montrose, Margery Li, James Kirby, Myong Kang
Center for High-Assurance Computer Systems
Naval Research Laboratory
Washington, DC, US 20375
<firstname>.<lastname>@nrl.navy.mil

ABSTRACT

Separation kernels are the strongest known form of separation for virtual machines. We agree with NSA's Information Assurance Directorate that while separation kernels are stronger than any other alternative, their construction on modern commodity hardware is no longer justifiable. This is because of orthogonal feature creep in modern platform hardware. We introduce the separation VMM as a response to this situation and explain how we prototyped one.

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection—*security kernels, verification*

General Terms

Security

Keywords

virtualization, hypervisor, virtual machine monitor (VMM), open source

1. INTRODUCTION

Use of *virtual machine monitors* (VMMs, using the hardware vendor's term for what are also called *hypervisors*) to share hardware between virtual machines is rapidly increasing. For security, VMMs are an attractive alternative to hardware separation. Conventional VMMs are typically designed to provide secure separation of their exported virtual machines but the strongest known separation for shared execution environments is provided by a *separation kernel* [26, 22]:

the task of a separation kernel is to create an environment which is indistinguishable from that provided by a physically distributed system: it must appear as if each regime is a separate, isolated machine and that information can only flow

This paper is authored by an employee(s) of the U.S. Government and is in the public domain. Non-exclusive copying or redistribution is allowed, provided that the article citation is given and the authors and agency are clearly identified as its source.

ACSAC '12, December 3-7, 2012, Orlando, FL
ACM 978-1-4503-1312-4/12/12

from one machine to another along known external communication lines.

(Rushby used the term *regime* to denote the hardware abstraction a separation kernel provides to a VM.) The separation kernel is a concept proven by practical application. We have a great deal of knowledge about how they can be constructed and used. Separation kernels are already available as, for example, the emerging MILS platform [2, 6] and the Green Hills Integrity® VMM [1]. Separation kernel software is small and simple, so it makes a small target that can be completely analyzed and verified mathematically.

Separation kernels represent the upper bound of practical software separation strength, but they are no longer justifiable on commodity hardware. The National Security Agency's Information Assurance Directorate (IAD) discusses this issue in its guidance for separation kernels on commodity workstations [31]. Their essential conclusion is that strict separation kernels are still the strongest known means of separation, but that there is no rationale for building strict separation kernels because of the increasing *orthogonal feature creep* of modern commodity hardware. Orthogonal feature creep makes commodity hardware too complex for separation kernels. The IAD report states

Ultimately, the problem with commodity desktop platforms comes down to the fact that too many developers and vendors are interdependent. Each organization involved in the creation of a desktop workstation has an economic interest in adding features to distinguish their version from the competition and often needs a particular, unique, and potentially powerful access. This prevents a unified strategy for security from emerging.

For commodity hardware, we need to find and understand the new practical upper bound for separation strength. It is critical to understand that we are not proposing to change the meaning of high robustness. Instead, we want to find a better approach to measuring, building, and using VMMs that have the greatest robustness that is justifiable for commodity hardware.

The first consideration would seem to be how such VMMs might be used but actually this begs the question of what is the practical upper bound of assurance that can be justified. This is the goal of the Xenon project, to investigate this new practical upper bound of assurance for modern commodity hardware.

If we take Rushby's separation kernel concept as an approach rather than a specific architecture, we can make sig-

nificant progress towards finding and understanding a justifiable upper bound. Lacking full mathematical verification, the alternative will not separate as robustly as a strict separation kernel. However it will provide stronger separation than commercial/open source best practice VMMs, for modern commodity hardware.

As an approach, the separation kernel combines the following concepts:

- partition all computation into VMs,
- isolate the VMs to use a small number of well-understood communication paths, and
- verify the security properties of the VMM to the highest level possible.

If we relax the assurance of the separation kernel from complete mathematical verification to the highest level commensurate with modern commodity hardware, but retain the strict isolation concept, we get a *separation VMM*. A separation VMM will

- run on modern commodity hardware,
- virtualize modern commodity operating systems,
- be smaller and simpler than a conventional VMM,
- use fewer and simpler communication paths, and
- have the highest assurance justifiable for its modern commodity hardware.

2. VMM SECURITY ALTERNATIVES

There are several general approaches to increasing the run-time security of a VMM:

- add an external run-time integrity verification mechanism,
- add an internal run-time integrity verification mechanism,
- add further self-protection mechanisms, to increase tamper resistance,
- reduce the size and complexity of the software, to decrease the number of residual security flaws, or
- use formal methods to decrease the number of residual security flaws.

Various combinations of these measures are possible as well. Research into these approaches has significantly increased our understanding of how VMM security can be improved. Separation kernels represent an extreme combination of the last 2 approaches; separation VMMs are a less extreme combination of the same.

Each approach also brings with it some undesirable impacts and shortcomings. Adding software of any kind typically increases space and time overhead but it also increases the size of the code base, most probably introducing new security flaws. If we accept the motivation for adding the software, viz. security flaws in software are inevitable, then we must accept that an addition itself has flaws. None of the reported VMM security research on adding software discusses measures taken to reduce the residual flaw density of

the added software (or flaws in its interaction with the existing VMM software). The added software may also increase the attack surface of the VMM, by providing new ways to access its run-time internals.

Integrity verification mechanisms face additional challenges. All verification mechanisms face challenges of sampling rate and coverage. In modern commodity hardware, small but significant parts of the security state are either difficult to capture or change too rapidly to sample in a reliable way. It is difficult to show that there is no way for an attacker to bypass these kinds of defenses. External verification mechanisms are relatively tamper-resistant, but sample too slowly and cannot observe all of the security state. Internal verification mechanisms are better at sampling but are also relatively easy to tamper with. Integrity verification defenses that are non-trivial are hard to assure. Their security properties are complex, i.e. precise definition of all possible attacks and how the integrity mechanism is sure not to miss any of them.

Augmenting the existing self-protection mechanisms of a VMM is an intuitive response to VMM security. (To save space, we will assume that the reader is already familiar with hardware protection mechanisms such as no-execute, write-protect, and other paging attributes, segment registers, cache-line address space identifiers, and guard pages.) Advanced Micro Device's (AMD's) decision to remove the segment register protection mechanisms from the x86_64 instruction set architecture (ISA) may have been justified by the fact that current operating systems were not using it, but the security results of segment register protection in the Multics project still apply [4, 15]. There is no characteristic of modern operating systems that would rule out use of this strong and efficient mechanism.

This lack of proper segment registers (or alternative compartmentation mechanisms) in the widely-used x86_64 ISA is a strong motivation for adding further self-protection mechanisms to VMMs. These added mechanisms are designed to directly prevent tampering with code in memory, malicious diversion of control flow, and access to unauthorized address spaces. The challenge is to design software mechanisms that both satisfy the three reference monitor properties¹ but also are efficient. New protection mechanisms must also permit typical system programming techniques such as function pointers and self-modifying code.

Reducing the size and complexity of the software not only decreases the number of residual security flaws in a VMM, it also strengthens its reference monitor property of verifiability. There are four fundamental challenges to this approach. First, as the IAD guidance concludes, modern commodity hardware is growing more and more complex, and virtualization of all of this under complexity constraints is hard. Second, modern unmodified guest operating systems are even more complex than the hardware, but a practical VMM must support this complexity. Third, simplicity can introduce significant performance penalties. It can be difficult to design code that is both simple and fast; especially under the restrictions imposed by complex commodity hardware and guest operating systems. Fourth, it can be difficult to report research results on approaches that reduce size or complexity. Peer-reviewed venues prefer novel, powerful, and flexible security features, which tend to be large and

¹completeness, isolation, and verifiability

complex rather than simple and small. To achieve significant results, approaches that reduce the size or complexity of the VMM, for security, must run on hardware that is arguably a commodity, e.g. x86_64 or ARM, and support commodity operating systems such as the various forms of Microsoft Windows, Apple OS X, or an unmodified standard Linux distribution. A significant result for minimizing the hypervisor must also retain all of the features needed to support cloud computing.

Formal methods applied to small (less than 10,000 source lines of code (SLOC)) separation kernels or embedded software systems can raise security assurance to the highest level. Even successful approaches have placed significant restrictions on standard system programming techniques such as the use of pointers. In addition to the verification size challenge per se, that is, how to verify enough code to assure an actual product, formal methods approaches also face the 4 challenges of size/complexity reduction approaches: complex underlying hardware, support for complex guests, performance penalties, and the lack of significance entailed by toy projects. We agree with the IAD guidance that mathematically verified separation kernels are not justifiable on modern commodity hardware.

3. PROTOTYPING A SEPARATION VMM

We prototyped the separation VMM concept using the Xen open source VMM. Our prototyping is limited to the Xen VMM itself. The Xenon VMM will work with any of the conventional Xen control plane tools such as *xm*, *xl*, or *lib-virt*. The prototype is tested continuously against not only Windows and various Linux distributions but also Hadoop, Zookeeper, and Accumulo. This confirms that our extensive refactoring and re-design of the VMM internals has preserved Xenon’s support for commodity software.

It is important to understand that our separation VMM prototype is not a security enhancement to Xen, but a separate VMM code base that does not necessarily conform to Xen community practices. The VMM security features that we describe here are neither recommended as changes to Xen, because they do not conform to Xen community goals and practices, nor, for the same reason, are they to be viewed as criticisms of Xen’s design.

Our prototype is based on Xen 4, starting with a 4.0.1 basis but then forward porting to a Xen 4.1.2 basis (to measure the difficulty of a forward port). While justifiable assurance is a key question of our work, that assurance must have a reasonable target. So the Xenon prototype not only explores separation VMM construction but also acts as a target for investigation of justifiable assurance.

Xenon’s code is not meant to be a special branch of the Xen code base. Xenon draws from Xen’s code, for selected new features and bug fixes, but separation VMM simplicity and assurance requirements rule out maintaining it as a branch of Xen. We validated this strategy by measuring the effort needed to keep the prototype synchronized with both the fixes and the new features of Xen [21]. The effort of analyzing, translating, vetting (not all bug fixes are applied to Xenon) and applying all of the Xen bug fixes created during a 6 month period was measured at approximately 100 hours from an experienced kernel coder [21]. The effort needed to synchronize new features from one Xen branch to another (4.0.1 to 4.1.2) was measured at approximately 200 hours.

Our changes to the Xen code have resulted in a VMM

Directory	Xen	Xenon
arch	64006	46520
common	15507	13204
include	3558	1691
crypto	1113	1072
drivers	19287	19256
xsm	6665	0
msm	0	269
Total	110136	82012

Table 1: Size Reduction of Xenon VMM in Non-comment Lines of Code. MSM is the Xenon replacement for XSM.

that is 26% smaller than conventional Xen, in terms of lines of code, with worst-case complexity [36] reduced from 2,450 to 70, a 3500% decrease. We report these values as approximate percentages because they change frequently, but downward over time. Table 1 shows specific values for Xenon change set 341 and its base Xen 4.1.2. Some simplifications also increase the total amount of code in the hypervisor, for example when a complex C function is broken up into many smaller functions.

4. SEPARATION VMM SECURITY

We wanted our separation VMM prototype’s enforcement mechanisms to be as verifiable as we could make them. We also wanted the security to be intuitive to use.

A virtual machine monitor and its guest virtual machines constitute a *virtual machine system* [25]. The combined security model for a VM system must define policies that are both consistent and intuitive. The need for consistency in security VM system models and policies is addressed by Rueda et al. [25] so we will not discuss it further here. The need for intuitiveness is less clear but no less important. Intuitiveness is related to *safety* [14]: a complex unintuitive security model and its policies may be described in a way that is theoretically tractable but leaves administrators of real systems unable to easily determine whether the configured policy satisfies their requirements. Ultimately, the policy enforced by a VMM must be easy to understand and to construct, otherwise users will disable or work around it. A complex resource-based policy expressed as rules about low-level platform-specific resources will be relatively difficult to translate into the high-level service agreements needed by cloud or SOA management. If a guest migrates to another VM system, it may need a different set of low-level rules on its new host VMM, because the devices and other peripherals it was using on the source have different names are not present on the destination host. We would like to have a policy that uses high-level rules that can be the same on all VMMs. We would like to have rules that cover equivalent resources, without being expressed in terms of specific resources.

Security in conventional Xen is enforced primarily by its Xen Security Module (XSM) but also by other small pieces of code at key points in the hypervisor. The XSM security model is based on SELinux (in turn based on Flask [29]) which provides flexible fine-grained type enforcement on individual resources. This flexibility comes at a price: the policies require many rules and it is difficult to translate the policy into an intuitive abstraction of what protection

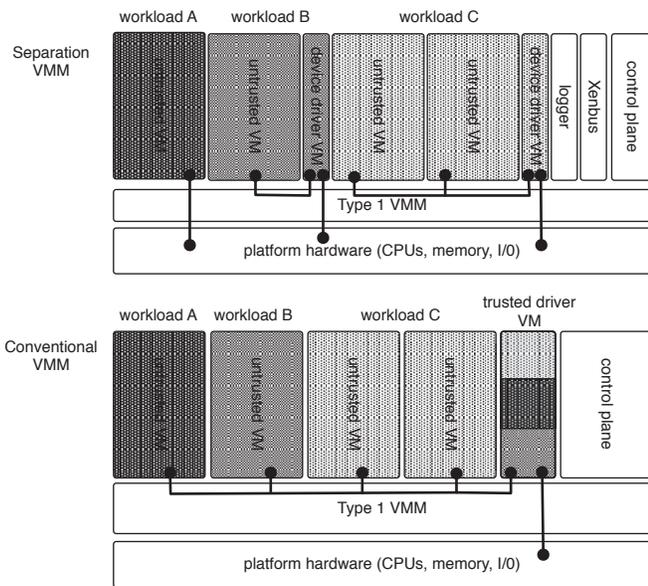


Figure 1: Separation VMM Security Mechanisms Are Simpler Because They Support Simpler Policies

a given policy specifies. The current practice for Xen is to enable security in a passive mode and record the violations, then create rules to address the violations one-by-one. While this approach yields a working policy, it cannot be used to construct a policy that is verifiably a refinement of a previously chosen high-level abstraction.

Xenon security policies are separation VMM policies with limited communication between guests. Xenon does not conform to the Xen code base. For these reasons we were able to replace Xen’s security implementation (269 versus 6665 lines of code) that protects all of the Xen code paths protected by Xen, but also provides additional VMM security features. As shown in Figure 1, Xenon virtual machine system configurations do not include trusted guests, e.g. a trusted device driver domain that serves guests whose information is supposed to be separated.

In the example of Figure 1, there are 3 workloads A, B, and C that are to be separated. While a conventional hypervisor can be configured to this same arrangement, it can also be configured as shown in the lower part of the figure, with a single driver domain that is trusted not to share anything contrary to the configured policy. The Xenon prototype does not support trusted guest configurations and thus its security mechanism is much simpler.

Security in Xenon is enforced entirely by a single module named MSM that replaces both the XSM security module and other code located at various places in the Xen internals. Like Xen, Xenon’s MSM enforces a policy described by a specification. The policy specification is written in XML and is compiled into a binary policy that is loaded into the VMM’s memory at boot time.

The number of rules in a specification is relatively small. For example, suppose that the workloads A, B, and C shown in the top of Figure 1 each run on a different kind of guest operating system. The workload A’s guest is configured for direct pass through of the underlying I/O hardware and pro-

vides its own device drivers. The single workload B guest has its own device driver domain that is configured to only access workload B resources, i.e. the device driver VM is untrusted. The 2 workload C guests are supported by a single untrusted driver VM. There are two additional service VMs: one for security logging and another to separate the Xenbus service [8, 33] from the domain 0 control plane. A complete policy for this configuration would require only 11 rules.

An MSM security policy contains rules describing

- domains,
- communication allowed between domains,
- rules for domain labels, and
- hypercalls allowed for each domain.

The XML for the rules includes features for aliases, profiles to collect frequently used sets of rules, domain rules, and domain sets to collect domains into a single named group. In the following, we explain each of these MSM features.

4.1 Domain Policy

A domain defines the virtual hardware environment that a guest operating system runs in, just as Xen does. Unlike Xen, Xenon always assigns the same domain id to the same domain. Security enforcement in Xen is based on an SSID that is generated from a domain’s UUID. To simplify from this design, Xenon assigns a fixed domain id that always corresponds to specific domain. MSM then uses the simple domain id as the security identifier. This design simplification is typical of our approach; since Xenon does not have to conform to the Xen code base we are free to implement direct solutions without getting sign off from the entire community.

The following policy fragment shows a single domain rule that defines a security logging domain. The profile defines the hypercalls that a security logging domain may make; the label is a mandatory access control label; and the UUID field is used to map the running VM back to its installed image.

```
<domain
  name="MsmLogger"
  id="5" profile="Logger" label="GUEST"
  uuid="1eee192b-1303-ecd4-da1d-8267fae46a1d"/>
```

To provide greater flexibility during the installation and configuration of a Xenon VM system, the domain policy allows execution of anonymous *unprotected* domains that are not defined in the policy. Unprotected domains are assigned domain ids starting at 10000. MSM will allow creation of VMs in unprotected domains, as long as no protected (i.e. defined in the policy) domains are executing their VMs. This allows an administrator to install, test, or otherwise provision VMs without security enforcement. When the VM system is otherwise ready for operational use, domains can be incrementally added to the security policy and tested as protected domains.

4.2 Communication Policy

MSM enforces communication between guests as a simple (upper triangular) *communication matrix* that defines which domains are allowed to communicate. Xenon’s communication policy is a simple all-or-nothing policy that is easy to

understand and verify: any 2 domains in a policy are either allowed to communicate and otherwise share resources, or they are not allowed to share anything. This is consistent with the separation VMM concept of using a small number of communication paths between domains. In the XML policy, communication rules are expressed as connections from domains to sets of domains. If a domain does not appear in any connection, then it can only connect to itself. This is a good example of how the separation kernel principle influences and simplifies the design of a separation VMM: we assume that most domains will not communicate with other domains and thus the default is easily described by the lack of a rule. The other type of communication pattern for a separation VMM is a *driver domain pattern* where a driver domain is allowed to connect to a set of domains that have the same workload or other security attributes.

The following is an example policy based on the top of Figure 1. There are no connection rules for the workload A guest or the logger VM because they do not connect to any other domain except the control plane, and follow the default pattern. The control_plane VM rule says that it is allowed to connect to guests in the special domain set All, which allows it to connect to any domain, even those that do not appear in the policy and are thus otherwise unconnected. The 2 workload driver rules allow the appropriate driver VMs to connect to their respective domains.

```
<connection from="control_plane" to="All" />
<connection from="workload_c_driver" to="workload_c" />
<connection from="workload_b_driver" to="workload_b" />
```

4.3 Label-Based Domain Policy

Xenon's MSM provides mandatory access control (MAC) labels for enforcement of coarse-grained per-virtual-machine policies. Labels can be defined and applied to domains and also to the domain's associated disk and network interfaces. MSM labels are not limited to lattice-based information flow policies; they can be used by Xenon to enforce a variety of rules including Chinese Wall conflict sets [7, 27], type enforcement [5, 34], and time-based rules. An example of the latter would be a domain label that restricted a domain to running only during normal working hours.

MAC labels are a key component of simplified security enforcement in the Xenon prototype, but their use is well understood, so we do not discuss them at length. Further discussion of label-based mandatory access control policy in VMMs can be found in Sailer, et al. [27].

4.4 Hypercall Policy

Labels provide a MAC policy in the Xenon prototype, at the granularity of a domain. We gain extra flexibility over domain-level MAC policies through management of individual resources. Xenon and Xen can enforce policies over exactly the same set of resources, but Xenon enforces less fine grained separation of resources than Xen. Instead of the Flask-style per-individual-resource rules of Xen, Xenon enforces resource control on a per-hypercall basis. Xenon's resource policies supplement its label-based domain access control. For example, supplementary rules can be useful when MAC labels are being used for Chinese Wall conflict sets. Two domains that are allowed to execute together according to a MAC Chinese Wall policy can still be separated in other ways, e.g. USB port restrictions, by resource policy rules.

In Xenon each guest is given a profile defining which hypercalls and hypercall subcommands the guest is allowed to make. (Xen hypercalls are organized into primary hypercalls with subcommands in order to reduce the size of the hypercall page [8].) If a guest attempts to make a hypercall that is not in its profile then its request is rejected and logged (see below). Xenon's MSM can be configured with a threshold for the number of security violations allowed by a guest. If a guest attempts more unauthorized hypercalls than the threshold then the guest is terminated by the hypervisor, and the attempt is both reported to the console and permanently recorded in the MSM security log. Since all resources are accessed via hypercalls, checking on a per-hypercall basis provides essentially the same protection. The distinction is that Xen can have two individual resources for a single domain labeled with different security labels. Xenon either allows the resource class to a domain or denies it.

MSM includes hooks that check the hypercall privileges of the guest, before the VMM begins execution of the hypercall. Hypercalls in Xen are implemented as a simple call to an entry in the guest's *hypercall page*, after the guest places all parameters in general purpose registers. Each hypercall page contains a mode-specific piece of trampoline code [8] that bounces the flow of control in a way that is appropriate for the guest's mode of operation, e.g., HVM mode for guests using hardware virtualization. For example, in the case of a guest running in HVM mode on an Intel processor, the guest's hypercall page contains trampoline code that executes a `vmcall` instruction which transfers the flow of control into the VMM to perform the hypercall. In our HVM mode example, the VMM code to execute the hypercall is in C function `hvm_do_hypercall`. The MSM hook is placed in this C function, to check all Intel HVM mode guest hypercalls.

Basing resource management on the VMM "API" hypercalls rather than on individual resources enables a significant reduction in the size and complexity of the security policy specification, as depicted by the diagram in Figure 2. Because a single hypercall controls many resources, the number of rules can be reduced, in comparison to a policy that specifies individual resources directly. The reduction in security enforcement code makes this clear. The drawback to this approach is the need to construct a resource-to-hypercall map, as part of an assurance argument, to show that all resources are protected. Since an assurance argument will need list all resources anyways, the additional map is not a significant assurance burden. Describing policies using per-resource rules will eliminate this map, but cause a combinatorial explosion of rules that is less intuitive.

4.5 Protecting Multicall

The multicall feature is potentially vulnerable to a time-of-check-time-of-use attack against the separation policy. Multicall allows guests to reduce virtualization hypercall overhead by grouping multiple hypercall requests into a single multicall hypercall. A typical use would be grouping multiple requests to update a page table. Multicall requests pass in a variable-length list of guest handles that point to guest memory locations. Each list entry is a data structure corresponding to a single hypercall request. This list is not copied into the VMM's address space, to save time but also to avoid overrun attacks.

The initial trampoline-based security check can only con-

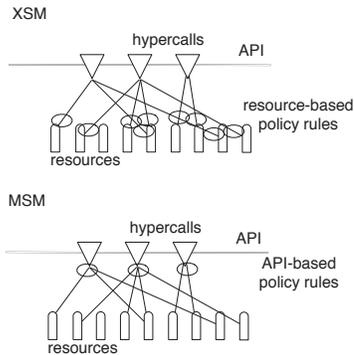


Figure 2: Comparison of resource- and API-based security policy rules.

firm a guest’s privilege to perform multicalls; it cannot check the entire list, because of the impact that would have on preserving the Xen-compatible interface of Xenon. Even if the trampoline-based check could confirm every entry on the list, a guest could still perform a time-of-check-time-of-use attack by modifying the list after the multicall was initiated. So the separation policy component of the MSM security module checks each list entry as part of the code that processes the multicall list internally. Each hypercall request is checked just before it is performed.

4.6 Logging

The MSM security logging feature allows the Xenon separation VMM to report its security events to external security management layers or intrusion detection and prevention systems. Conventional Xen’s XSM does not provide logs of security activity. This is not unusual, because persistent logging of events of any kind is problematic for VMMs. Excepting a rudimentary console for the local host hardware, a VMM does not have its own file system, high-level network interfaces (for remote persistent storage), or even disk drives per se. Instead, a VMM manages or exports devices such as block or serial devices, so any persistent storage that could be used for sensor data must be implemented by a guest. Conventional Xen’s non-security VMM logging is implemented as writes to Xen’s local console device.

MSM logging collects security events as fixed-length (currently 512 byte) records held in VMM memory. A single-purpose logging guest is given the privilege of registering with the VMM and subsequently pulling log records from VMM memory. All security logging interaction with the VMM is through a logging-specific hypercall that is controlled through the separation policy, so only authorized guests can manipulate the security logs. A typical configuration would be to have the logger directly export the log records into to a remote host, so that the log data could be incorporated into intrusion detection systems.

Xenon uses a special purpose logging guest operating system based on Xen’s mini-os. This gives the logging guest a small code base, less than 10,000 lines of source code in its present form, and a correspondingly small attack surface.

4.7 Hypercall Flooding

The Xenon prototype also includes a simple defense against *hypercall flooding attacks*. In a hypercall flooding attack, a

VMM.VM	1st Run	2nd Run	3rd Run	Change Set
Xen.alpha	167.02	167.11	167.04	0
Xen.delta	198.19	197.95	197.94	0
Xenon.alpha	167.25	167.13	167.31	294
Xenon.delta	199.00	199.16	199.21	305
Xenon.alpha	167.14	167.06	167.21	343

Table 2: Single-VM in seconds for kcbench. Change Set 0 is the Xen base. VM alpha is a PVops HVM Linux guest; VM delta is a paravirtualized Linux guest.

malicious guest generates a large number of hypercalls in an attempt to consume VMM resources. The malicious guest makes specious hypercalls but discards the results and does not wait for the VMM response. This is analogous to network attacks based on SYN flooding [3]. The Xenon VMM can be configured to measure and limit the hypercall rate for its guests. Our measurements show a typical rate of 30 hypercalls per millisecond for domain 0, running on either the Xen VMM or Xenon. The Xenon prototype can be configured to limit the hypercall rate in terms of hypercalls per millisecond. The defense is simple and adds no measurable overhead.

4.8 Minimizing Overhead

One of the goals of the Xenon project is to reduce the runtime space and time performance overhead of both the security and assurance modifications to conventional Xen. We tested each change for performance, using a variety of commodity hardware, with several benchmark tools. The benchmarks were configured to stress test the prototype VM system while measuring its performance. We have been able to keep the runtime overhead within acceptable limits by backing out any changes that perform poorly. This approach let us simplify the code by over 3000% with no impact (the raw data shows the simplified code actually ran faster, but the difference was in the noise). More information about Xenon performance measurements are presented in [21]. We present a brief table here of single VM data from a kcbench Linux compiler benchmark, as Table 2. The data is for a machine using Hardware Assisted Paging (HAP), with an Intel® S1200BTL base board, a single Intel® Xeon® (E31270) processor (3.20GHz) and 16GiB of memory. The domain 0 kernel was Fedora 3.1.9-1.fc16.x86_64; the same kernel was used for the PVops HVM guests and the paravirtualized guest was installed from the same Fedora 16 ISO image. The examples are for different Xenon change sets, to show how we have been able to maintain performance as the internals are simplified.

An important part of security overhead is the cost of factoring Xen’s single large domain 0 into multiple single-purpose security domains. Each one of these domains brings with it some scheduling and context switch overhead. The overhead of the MSM logging domain is so small that we were unable to measure it accurately, that is, the cost of logging is negligible. The data for change set 343 in Table 2 shows the performance change caused by running the security logger; notice that for the 2nd run, the measurement actually shows the VM system to be faster with the logger than without, i.e. the difference is not measurable in this test.

5. JUSTIFIABLE ASSURANCE

Given the separation VMM concept and a working prototype, we need a more concrete notion of highest justifiable assurance for modern commodity hardware. We need to understand how orthogonal feature creep reduces the value of individual components of an assurance argument. The key question is: does orthogonal feature creep reduce the value of a specific measure to the point where it is no longer justified?

We can use the amount of rework required by a change in hardware as the measure of value reduction. The fraction of a work product that must be changed to preserve an assurance argument, with respect to hardware variation, can be viewed as the “cost” of the hardware variation.

Using this perspective, we can identify specific assurance measures that are less sensitive to orthogonal feature creep than complete mathematical verification but still exceed commercial/open source best practice for VMM construction. For the Xenon project, we found the following 6 measures to be justifiable for modern commodity hardware. These measures also increase assurance beyond commercial/open source best practice, Common Criteria EAL4, or both. We do not consider our collection of measures to be definitive for either the concept of highest assurance justifiable for modern commodity hardware or separation VMMs. Instead, they provide a consistent package of assurance measures that can be applied to a separation VMM to give higher assurance than commercial/open source best practice etc. but not be as sensitive to orthogonal feature creep.

5.1 Reduced Features and Size

Commercial/open source best practice does not reduce features (or the size of a product) to obtain greater security. Reducing the features of a software product strongly tends to reduce both its market share and installed base. VMM software also suffers from orthogonal feature creep. So a key element of the separation VMM concept is to push back against this creep by directly reducing the features of a conventional VMM, and then reducing the size and complexity of the remaining software. The reduction is constrained by the requirement for the reduced VMM to virtualize precisely the same guests that the conventional VMM supports. Reducing the features of a product strongly tends to reduce both its attack surface and the total residual flaws present in the product. Reducing product features is easily justified as an assurance measure. It reduces the likelihood that a given hardware variation will impact the VMM and reduces the total amount of rework because the both the product and its assurance argument is smaller. Feature and size reduction is also essential to the separation VMM concept.

Space limitations prevent us from explaining all of the feature and size reductions we applied in constructing the Xenon prototype. We can give a few examples

- support for the Itanium processor: our initial decision was simply to limit the source code to a single instruction set architecture (ISA). We dropped the Itanium because there is a greater variety of commodity hardware for the x86 ISA family,
- support for the 32-bit x86 processor: we removed support for this older member of the x86 ISA family,
- transcendent memory: the design of this feature does

not properly scrub memory before re-using it and the feature is not essential,

- supervisor mode kernel: this feature is a mode of VMM execution that supports running a single VM with the same privilege level as the VMM itself. This single supervisor mode kernel VM is the only VM that can run in this mode. This VMM mode is not essential to normal virtualization.

5.2 Reduced Complexity

In this case, we mean reducing complexity to be lower than commercial/open source best practice. Developing simple but correct and efficient code is time consuming; simplifying code beyond commercial/open source best practice not only reduces the likelihood of security weaknesses but also makes it easier to review by automatic or manual means. Simplifying the VMM internals to satisfy CC requirement ADV_INT.2 also increases assurance beyond EAL4.

In our prototype, we simplified code by the application of a small set of patterns. This was only possible because a great deal of the Xen code consistently uses other patterns and it is possible to re-factor the Xen patterns into the Xenon patterns. This refactoring still requires developers analysis; it is probably not practical to use code generation or other machine translation techniques to generate the simplifications. In some cases, e.g. in bringing the complexity of C function `x86_emulate` down from 2,450 to 27², it was necessary to design an intermediate form of the code, to support changing and testing the new code in small increments. The final stage of the re-factored code follows the Xenon simplification patterns, but the mapping is not direct from old to new.

The use of a small set of patterns to re-factor the Xen code is essential to the relatively low cost of keeping pace with the conventional Xen code base. Because of the patterns, an experienced programmer can quickly spot the points where the Xenon code should be changed, when reading a Xen patch.

5.3 Programming Language Subset

Use of a programming language subset exceeds commercial/open source best practice for VMMs. While subsets are commercial best practice for safety critical embedded systems, subsets are specialist practice for general commercial software development. Specialist use of subsets for safety critical systems makes the case that subsets provide greater assurance. Use of a programming language subset exceeds all Common Criteria assurance requirements. For a given language, the use of a carefully designed subset should increase assurance with minimal risk from hardware variation.

For the Xenon project, we chose Hatton’s EC– subset [12] of the C programming language as a basis. We had to modify this subset because the EC– subset is designed for ISO C 9899 but Xen is implemented using the GCC compiler in a way that does not conform to the ISO standard. For example, Xen’s implementation contains many uses of the `VARARGS` macros, which breaks the ISO C standard. For Xenon the modified subset uses the EC– rules that are independent of ISO C, and then conforms to the GCC features

²The careful reader will have noticed that 27 is less than the value of 70 mentioned earlier. After simplification, function `x86_emulate` is no longer the most complex function in Xenon.

Xen uses to enhance the assurance of the code, e.g. the `-Werror` flag.

5.4 Formal Security Model

Common Criteria EAL4 does not require a formal security model to guide the design and implementation of the security features of a software product. However, using a formal security model (requirement ADV_SPM.1) does increase assurance as it is used in CC EAL’s 6 and 7. Because it is very abstract, a formal security model is not likely to require significant (if any) rework because of hardware variation.

Franklin et al. [10] have already developed a formal model for the sHype security mechanism of Xen [27]. They model checked the Xen sHype Chinese Wall model using the Mur ϕ model checker. (The sHype mechanism was implemented in Xen as the Access Control Module (ACM) but ACM has been replaced by NSA’s XSM, as described above.)

McDermott and Freitas developed a formal security model specifically for Xenon [19]. This model used the Circus formalism [37] to adapt the independence policies of Roscoe, Woodcock, and Wulf [23, 24] into a state-based framework. This kind of formal security model is especially suitable for practical separation VMMs because it is a definition of security that is preserved by refinement.

5.5 Formal Interface Specification

Common Criteria EAL4 does not require a formal interface specification to guide the design and implementation of the security features of a software product. A formal interface specification (ADV_FSP.6) is used in CC EAL7, so it does increase assurance. It also leverages the assurance provided by a formal security model. An interface specification is much less abstract than a security model, but significantly more abstract than internal design or source code. A formal interface specification is likely to require rework due to hardware variation, but not enough to rule out its use. Orthogonal feature creep is also mitigated by the fact that there is no formal relationship between the interface specification and the underlying design and source code, but only a semiformal one at best. The cost of reworking a semiformal mapping is significantly less than the cost of remapping a formal refinement.

Freitas, McDermott, and Woodcock developed a partial formal interface specification [11] for Xenon. The specification modeled the event channel interface in the Circus formalism and was verified using a combination of tools including the Z / Eves theorem prover and the Community Z Tools (CZT) suite. The specification also demonstrated refinement of the formal security policy model [19] of McDermott and Freitas.

5.6 Abstract Formal Design Models

While formal verification of the complete design is not justifiable, it is justifiable to verify abstract models of parts of a design. For example, Franklin et al. have model checked the design of the Xen shadow paging mechanism [9], to show that it correctly isolates one guest’s memory from another. Formally verifying the abstract design of a VMM subsystem or component is not just an academic exercise, previous research by Franklin et al. has discovered flaws in actual VMMs [10]. Since the design models are abstract, they are less sensitive to changes in hardware.

6. RELATED WORK

There is a rapidly growing body of virtualization security research. We only discuss work that is closely related to our own goal: a separation VMM that has arguably higher security than a conventional VMM. Examples of conventional VMMs would be open source Xen distributions such as Citrix XenServer, VMWare’s ESXi, and Microsoft’s Hyper-V.

Early IBM work by Sailer et al. [27] developed a mandatory access control framework for Xen called Access Control Module (ACM). Xenon’s MSM follows some of the design principles of the IBM ACM framework.

Szefer et al. have prototyped NoHype [32]. NoHype severely reduces the size of the run-time interface between the VMM and the guest virtual machines of a platform, by locking guests to specific processor cores and reducing hypervisor interaction to a stub that handles certain privileged events such as VM exits. This makes it very unlikely that any guest can attack the VMM through its interface.

Ultimately, the NoHype approach is very similar to allocating separate hardware blade servers, from a blade enclosure, to each customer, and is not really virtualization at all, as confirmed by the title of Keller et al. [16]. The design sacrifices some of the benefits of cloud computing on a virtualized infrastructure. There is an added drawback when compared to dedicated blade servers: NoHype still has the software VMM and Linux control plane (domain 0) running on the shared hardware.

HyperSafe [35] adds software self-protection mechanisms to VMMs that run on commodity hardware, to compensate for the hardware inadequacies discussed above. The HyperSafe project prototyped 2 alternative mechanisms: *memory lockdown* based on x86 hardware support for copy-on-write and *restricted pointer indexing*. The memory lockdown feature was constructed using conventional tools, but the restricted pointer indexing is based on the specialized LLVM compiler [18]. Full results have been reported for prototyping these techniques on BitVisor [28]. BitVisor is too small and simple to support cloud computing or similar applications, but instead strongly protects a single guest. Wang and Jiang report partial results for the more complex Xen hypervisor: the memory lockdown technique has been prototyped but no code size or performance results were given [35].

NOVA [30] increases VMM security by layering it into 1) a more privileged component (micro-hypervisor) that directly controls the hardware and manages VM exits, and 2) a user space per-guest VMM that emulates sensitive instructions, handles VM exits, and implements virtual devices. Communication between the privileged layer and the per-guest VMMs is by messages sent from the privileged layer to portals in the per-guest VMMs. For example, if a per-guest VMM executes a sensitive instruction, the per-guest VMM performs a VM exit and the privileged VMM transmits the correct state to a handler at the appropriate port in the per-guest VMM. The extra transitions between layers do impose an overhead compared to conventional virtualization technology; the precise amount varies depending on the hardware/software configuration used. The NOVA prototype currently runs unmodified Linux guests but is actively under development to reduce overhead and support Windows guests.

Zhang et al. [38] have prototyped CloudVisor to use nested virtualization and encryption to protect the guest

VMs, the higher level VMM (a modified Xen VMM), and the control plane VM. The underlying CloudVisor VMM is tiny, about 5K SLOC. CloudVisor does have some performance impact, e.g. worst case I/O intensive applications over 50%. The CloudVisor approach can provide strong security but the threat model precludes cloud providers from inspecting the computing practices of tenant guest machines, to ensure that tenants abide by their good user agreements. For clouds where the provider does not need to enforce any kind of restrictions on tenant behavior, the CloudVisor approach is a good fit.

The Xenon project constructed a Xen based prototype [20] that focused on transforming open source Xen 3 into a Common Criteria defined higher assurance form. Unlike the prototype reported here, the earlier Xenon 3 prototype was a partial prototype that did not change Xen security subsystems. The Xenon prototype has re-factored the entire Xen 4 code base and made extensive changes to the Xen security subsystems.

Current examples of full mathematical verification applied to VMM-like systems included Klein et al. [17] and Heitmeyer et al. [13]. The *L4.verified* project of Klein et al. is targeted at cell phones and similar hardware. The formal verification applied to C source code (i.e. a practical verification) but placed impractical restrictions on the use of pointers (for system software) and did not verify memory management. The definition of security verified was not a information flow security definition, but the take-grant model, which is much simpler than a true information flow model that would be needed by a separation kernel. The embedded device verified by Heitmeyer et al. has much higher assurance and is verified to enforce a true information flow model. It is accurately described as a strict separation kernel but it does not run on commodity hardware.

7. CONCLUSIONS

Our results confirm IAD's position on modern commodity hardware. In parts of Xenon where the code is largely independent of the underlying hardware, a useful separation kernel style policy can be enforced with less than 500 lines of code. In the parts of the VMM that must deal with orthogonal feature creep and legacy hardware features, e.g., the paging code, the reduction cannot be as much. We tried simplifying the paging code but the performance penalty for the simpler code was in excess of 5%, i.e. too much for a single security feature.

We have been successful in reducing the size of a conventional VMM to roughly 3/4 of its original size without losing the ability to virtualize the same operating systems as Xen. Some of this success is due to orthogonal feature creep in the conventional VMM software but some is also due to being focused on a different goal. The Xen community is concerned with security but makes fewer concessions to it when design tradeoffs are considered. The Xenon VMM puts security on par with function and performance.

Our ability to simplify the code exceeded our expectations. At the beginning of the project we did not think it would be possible to actually simplify C function `x86_emulate` by such a large amount and we further did not expect it to be as fast as the original. Our results indicate that using an open source VMM as the basis for a higher assurance separation VMM is practical.

Our performance results for the security logging domain

are encouraging. They suggest that it should be possible to refactor Xen's domain 0 into a number of simpler management domains, without harming performance.

Application of formal methods to abstract representations can be useful. Formal models of not only the hypercall interface but also abstractions of key subsystem design (e.g. the shadow paging) and the fundamental security policy can increase the assurance of a VMM, without excessive sensitivity to orthogonal feature creep.

Our future work with Xenon will focus on further size reduction, reducing the size of the domain 0 control plane, and construction of evidence for less-than-full-mathematical-verification. None of this work would have been possible without the high-quality Xen code base.

8. REFERENCES

- [1] Green Hills Software INTEGRITY-178B Separation Kernel, comprising: INTEGRITY-178B Real Time Operating System (RTOS), version IN-ICR750-0101-GH01_REL running on Compact PCI card, version CPN 944-2021-021 with PowerPC, version 750cxe. Science International Applications Corporation (SAIC), September 2008.
- [2] J. Alves-Foss, W. S. Harrison, P. Oman, and C. Taylor. The MILS architecture for high assurance embedded systems. *International Journal of Embedded Systems*, 2((3/4)), 2006.
- [3] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*, 2nd ed. Wiley Publishing, Inc., 2008.
- [4] A. Bensoussan, C. Clingen, and R. Daley. The Multics virtual memory: concepts and design. In *Proc. Symposium on Operating Systems Principles (SOSP)*, 1969.
- [5] W. E. Bobert and R. Y. Kain. A practical alternative to heirarchical integrity policies. In *Proc. 8th National Computer Security Conference*, Gaithersburg, Maryland, US, 1985.
- [6] C. Boettcher, R. DeLong, J. Rushby, and W. Sifre. The MILS component integration approach to secure information sharing. In *27th IEEE/AIAA Digital Avionics Systems Conference*, 2008.
- [7] D. Brewer and M. Nash. The Chinese wall security policy. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 206–214, Oakland, California, US, May 1989.
- [8] D. Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice-Hall, 2008.
- [9] J. Franklin, S. Chaki, A. Datta, J. McCune, and A. Vasudevan. Parametric verification of address space separation. In *Proc. 1st Conference on Principles of Security and Trust (POST)*, Tallin, EE, March 2012.
- [10] J. Franklin, S. Chaki, A. Datta, and A. Seshadri. Scalable parametric verification of secure systems: How to verify reference monitors without worrying about data structure size. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, US, May 2010.
- [11] L. Freitas, J. McDermott, and J. Woodcock. Formal methods for security in the Xenon hypervisor. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(5):463–489, 2011.

- [12] L. Hatton. EC— a measurement based safer subset of ISO C suitable for embedded systems development. *Information and Software Technology*, 47(3):181–187, 2005.
- [13] C. Heitmeyer, M. Archer, E. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proc. 13 ACM Conf. on Computer and Communications Security*, Alexandria, Virginia, US, 2006.
- [14] T. Jaeger and J. Tidswell. Practical safety in flexible access control models. *ACM Trans. on Information and System Security*, 4(2):158–190, May 2001.
- [15] P. Karger and R. Schell. Thirty years later: Lessons from the Multics security evaluation. In *In Proc. Annual Computer Security Applications Conference*, 2002.
- [16] E. Keller, J. Szefer, J. Rexford, and R. Lee. Virtualized cloud infrastructure without the virtualization. In *International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society Press, June 2010.
- [17] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cook, P. Derrin, D. Elkaduwe, K. Englehardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating System Principles*, Big Sky, MT, US, October 2009.
- [18] C. Lattner. LLVM: An infrastructure for multi-stage optimization. Master’s thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.
- [19] J. McDermott and L. Freitas. A formal security policy model for Xenon. In *Proc. Formal Methods in Security Engineering (FMSE ’08)*, October 2008.
- [20] J. McDermott, J. Kirby, B. Montrose, T. Johnson, and M. Kang. Re-engineering Xen internals for higher-assurance security. *Information Security Technical Report*, 13(1):17–24, 2008.
- [21] J. McDermott, B. Montrose, M. Li, J. Kirby, and M. Kang. The Xenon separation VMM: Secure virtualization infrastructure for military clouds. In *Military Communications Conference - MILCOM 2012*, Orlando, FL, US, October 2012.
- [22] B. Randell and J. Rushby. Distributed secure systems: Then and now. In *23rd Annual Computer Security Applications Conference (ACSAC)*, Miami, FL, US, December 2007.
- [23] A. Roscoe. CSP and determinism in security modelling. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, US, May 1995.
- [24] A. Roscoe, J. Woodcock, and L. Wulf. Non-interference through nondeterminism. In *Proc. ESORICS*, Brighton, UK, November 1994.
- [25] S. Rueda, H. Vijayakumar, and T. Jaeger. Analysis of virtual machine system policies. In *Proc. ACM Symposium on Access Control Models and Technologies (SACMAT)*, Stresa, Italy, June 2009.
- [26] J. Rushby. Design and verification of secure systems. *Proc. ACM Symposium on Operating System Principles*, 15:12–21, 1981.
- [27] R. Sailer, T. Jaeger, E. Valdez, R. Cáceres, R. Perez, S. Berger, J. Griffin, and L. van Doorn. Building a MAC-Based security architecture for the Xen open-source hypervisor. In *Proc. 21st Annual Computer Security Applications Conference*, Tucson, Arizona, US, December 2005.
- [28] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. BitVisor: a thin hypervisor for enforcing I/O device security. In *Proc. 2009 ACM SIGPLAN/SIGOPS Int. Conf. on Virtual Execution Environments*, pages 121–130, Washington, DC, US, 2009.
- [29] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: system support for diverse security policies. In *Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*, Washington, DC, US, 1999.
- [30] U. Steinberg and B. Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proc. 5th European conference on Computer Systems*, pages 209–222, Paris, FR, 2010.
- [31] Systems and N. A. Center. *Separation Kernels on Commodity Workstations*. Information Assurance Directorate, NSA, March 2010.
- [32] J. Szefer, E. Keller, R. Lee, and J. Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proc. Computer and Communications Security*, Chicago, IL, US, October 2011. ACM.
- [33] C. Takemura and L. Crawford. *The Book of Xen*. No Starch Press, 2010.
- [34] K. Walker, D. Sterne, M. L. Badger, M. Petkac, D. Shermann, and K. Oostendorp. Confining root programs with domain and type enforcement (DTE). In *Proc. 6th USENIX UNIX Security Symposium*, San Jose, California, US, July 1996.
- [35] Z. Wang and X. Jiang. HyperSafe: a lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proc. 31st IEEE Symposium on Security & Privacy*, Oakland, California, US, May 2010.
- [36] A. Watson and T. McCabe. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*, NIST Special Publication 500-235. National Institute of Standards and Technology, 1996.
- [37] J. Woodcock, A. Cavalcanti, M.-C. Godel, and L. Freitas. Operational semantics of Circus. *Formal aspects of computing*, 2008. in press.
- [38] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proc. 23rd ACM Symp. on Operating Systems Principles (SOSP)*, pages 203–216, Cascais, Portugal, 2011.