# Secure Memory Management on Modern Hardware

*Reto Achermann, Nora Hossle, Lukas Humbel, Daniel Schwyn, David Cock, Timothy Roscoe*
*Systems Group, Department of Computer Science, ETH Zurich*

## Abstract

Almost all modern hardware, from phone SoCs to high-end servers with accelerators, contain memory translation and protection hardware like IOMMUs, firewalls, and lookup tables which make it impossible to reason about, and enforce protection and isolation based solely on the processor's MMUs. This has led to numerous bugs and security vulnerabilities in today's system software.

In this paper we regain the ability to reason about and enforce access control using the proven concept of a *reference monitor* mediating accesses to memory resources. We present a fine-grained, realistic memory protection model that makes this traditional concept applicable today, and bring system software in line with the complexity of modern, heterogeneous hardware.

Our design is applicable to any operating system, regardless of architecture. We show that it not only enforces the integrity properties of a system, but does so with no inherent performance overhead and it is even amenable to automation through code generation from trusted hardware specifications.

## 1 Introduction

Both new, fully-verified kernels and traditional production-quality operating systems rely on a model of memory addressing and protection so simple it is rarely remarked on: RAM and devices reside at unique addresses in a single, shared physical address space, and all cores have homogeneous memory management units (MMUs) which translate virtual addresses into this single physical address space.

The OS running on the platform then fulfills two roles: First, it manages *resource allocation*. Virtual memory makes multiplexing hardware easier by decoupling the application's view of memory from the physical resources managed by the OS, allowing *late binding* of addresses. Second it forms, alongside the MMU, a *reference monitor* [4]: All resource accesses (dereferences) are intercepted by the monitor (specifically the TLB), and checked against an *access-control policy*. This has for decades formed the basis for secure process isolation in *all* operating systems implementing virtual memory.

The reference monitor concept repeats throughout traditional OS design, with more sophisticated abstractions gradually built up, and their associated security properties enforced through a combination of hardware-provided monitors (e.g. MMUs), and software ones (e.g. traps and syscalls).

For example, consider name (or address) resolution and authorization checks in the `mmap()` syscall. A process begins with a *reference* to a file: its filename. The OS, meanwhile, enforces some access-control policy, e.g. UNIX-style permissions. The calling process *dereferences* the filename by passing it to the `open()` syscall, whereupon the OS validates the request against policy (permissions), and *resolves* the reference to another reference: the file descriptor (FD), now referring to an entry in the global open-file table. The existence of this entry, and that the process may possess a reference is justified by the top-level policy; The pattern of open files and FDs (the state) is a *projection* of something permitted by the policy.

This pattern is replicated in the VM system thanks to `mmap()`. Unix cannot directly interpose on memory reads and writes (to the buffer cache page mapped to the user), but does implement the initial `mmap()` call, and the page fault handler. The kernel builds a reference monitor by *composing* itself with that provided by the MMU. On an `mmap()` call, the kernel verifies that the FD is valid, with appropriate permissions (e.g. write), before constructing a VM region to back the mapping. The policy encoded in the region's flags is thus a (transitive) projection of the original file system permissions. On a page fault, the kernel is again invoked to lazily populate the region (from the buffer cache). Now, it can consult the mapping parameters (e.g. writable), and translate these to flags in the page-table entry.

Thus, the page-table state (e.g. permission bits), and thence the eventual TLB state, are justified by a chain of monitors all the way back up to the system policy (file system permissions). The MMU enforces this *projected* policy on the OS' behalf. Together they form, in security terms, a *compound reference*

*monitor* to enforce a policy both on real hardware resources (RAM), and abstract OS-specific objects (processes, files).

This model has worked well for decades, but has been undermined by a changing hardware contract. A modern system contains not just processors and their attached MMUs, but system MMUs or IOMMUs, memory firewalls, region lookup tables, etc. all of which mediate access to and from parts of the platform. "Smart" devices like GPGPUs, co-processors, network cards, or accelerators come with their own hardware protection and translation units [20].

In such a system, the processor's MMU alone does not form a reference monitor for memory, as it is not invoked on all accesses. Indeed, the complex address-translation topology of these systems renders even the concept of a unique physical address meaningless, raising the risk that the policy encoded into the distributed hardware reference monitor (the collections of MMUs, SMMUs, etc.) is inconsistent due to their differing views of the machine. These two problems have already led to security vulnerabilities [32, 33, 37, 41].

We identify three classes of security vulnerabilities and bugs (Table 1) that *i)* cause the execution of an operation without sufficient rights (a failure of *policy enforcement*), *ii)* allow a compromise of the reference monitor itself (e.g. writing translation tables, a failure of *partitioning*), or *iii)* use the wrong addresses in descriptors or pointers (a failure of *name resolution*). The lack of a proper reference monitor which is aware of the complex and configurable addressing network continues to result in numerous bugs and security vulnerabilities [14, 21, 42, 45, 46, 53, 61]. Confining these bugs in a kernel is hard, and they are likely to compromise the entire system [13].

In this paper we demonstrate that these whole classes of bugs can be prevented by extending the traditional OS-MMU reference monitor to cover *all* hardware translation and enforcement engines, allowing policy enforcement on all memory accesses, ensuring consistent name resolution by adopting the *decoding net* [1, 2] as a more faithful model of modern addressing hardware, and ensuring the secure partitioning of reference monitor state either through a partitioned capability system, or in a traditional kernel (such as Linux) by good software engineering practice and the application of existing memory management interfaces.

Our first contribution is to identify the undermining of the traditional OS-MMU reference monitor by a changing hardware/software contract as the root cause of several large classes of critical security bugs.

Our second contribution is to adopt a faithful model of complex addressing hardware (the decoding net), and from it derive a minimal *least-privilege* model of memory management authority on modern hardware, covering the common functionality of all virtual memory systems (§ 4.1).

Our third contribution is the specification of an OS-agnostic *reference monitor* to enforce policy expressed in the above model, prototyped as an *executable specification* in Haskell,

| **Type** | **CVE-...** |
|---|---|
| Policy enforcement | 1999-1166 2014-3601 2014-8369 2014-9888 2017-16994 2019-2250 2019-10538 2019-10539 2019-10540 |
| Partitioning | 2011-1898 2013-43292014-0972 2018-1038 2018-11994 2019-2182 2019-19579 |
| Name resolution | 2013-4329 2014-9932 2016-3960 2016-5349 2017-8061 2017-12188 2019-15099 |

Table 1: Classes of Security Vulnerabilities.

and abstracting the OS's internal policy language (e.g. capabilities or ACLs) as an *access-control matrix*.

Our fourth contribution is to demonstrate that this reference monitor design can be implemented without invasive changes on either partitioned capability systems (e.g. seL4 or Barrelfish), or on ACL-based UNIX-style kernel (such as Linux). Further our benchmarks demonstrates that there is no measurable performance cost for a secure fully-explicit least-privilege system-wide virtual memory authority implementation (§ 6)

## 2 Eliminating Classes of Bugs

The difficulty of getting complex memory addressing right in an OS is shown by the steady, ongoing stream of related bugs and vulnerabilities in operating systems, for example, policy enforcement in Linux's memory management code [25].

We identify three classes of common bugs and security vulnerabilities related specifically to the incompleteness of the current reference monitor, which would be rendered impossible under comprehensive reference monitor which faithfully reflected the hardware:

**Policy Enforcement.** These are bugs where a subject was able to change the configuration of a translation unit without having the proper rights do to so. The reference monitor fails here to enforce the system policy:

- Mappings with holes belonging to another subject [39].
- Incorrect permissions on data pages [40].
- IOMMU configured to map too large a range [47–49].

All these bugs are impossible once the operations are performed through a (correct) reference monitor implementing the system security property.

**Partitioning.** These bugs involve bypassing the reference monitor directly e.g. by directly modifying its internal state:

- DMA transfers into MSI-x interrupt registers [36].
- DMA transfers into IOMMU control registers [38].
- Process modifies its own page table [44].

These are prevented once the reference monitor state is identified and *partitioned* by subjecting them to system policy e.g. that no DMA engine or process may map a page table.

**Name Resolution.** This class represents inconsistent interpretations of pointers (names):

- Insufficient context to identify the correct object [42].
- Resolving addresses in the wrong context [43].

These are prevented once names are dereferenced (resolve) through a monitor with a complete, accurate model of addressing.

# 3 Background and Related Work

Before presenting our authority model and the executable specification in the next section, we will briefly cover reference monitors in a little more detail, in particular the importance of consistent naming, and how complex addressing topologies make it difficult.

We also summarize the existing decoding net model, the executable specification/refinement approach which we borrow from the seL4 system, and the related work.

## 3.1 Reference Monitors

The reference monitor is a powerful structuring concept in access control, and is implicitly used in practically every OS. A reference monitor enforces an access-control policy, allowing a separation of concerns, and thus effort: if *every* access is subject to the policy, then the overall safety of the system (w.r.t. the policy) can be guaranteed *independently* of the correctness of the components making the accesses. This is of enormous benefit to a monolithic system (e.g. Linux), where a fault in one subsystem can easily spread to others, particularly as any subsystem can, in principle modify translations. Even without enforcing a strict boundary between components (as in a microkernel), routing all updates via a single component responsible for safety ensures that *accidental* errors will no longer lead to a whole-system compromise.

The critical point for a reference monitor is that *all* accesses must pass through to it, and that it is able to accurately identify which resources are being accessed (e.g. which DRAM address will ultimately be written) when applying its policy. Both of these are undermined in the complex address-translation networks of modern systems, but not fatally so: The hardware component of the reference monitor is now *distributed* among multiple system MMUs, firewalls, etc.; addresses may be rewritten *after* policy is applied, routing them to locations that should not be accessible.

Both of these problems are solved with an accurate model of the hardware: First, to know the complete set of access-control components that must be included in the reference monitor, and second, to guarantee that any translation below the access-control level is consistent with policy.

## 3.2 The Canonical Name Problem

As established, modern platforms are composed of multiple, heterogeneous cores and devices each of which can issue accesses to addressable resources such as DRAM, non-volatile memory or device registers. Worse, there is no single "reference" physical address space [20]. Instead, a network of address spaces or buses is connected by address translation units which "route" memory accesses. As just described, in order to securely enforce access control, it is essential to know what final resource some intermediate address (or *name*) refers to.

I/O memory management units (IOMMUs, or system MMUs) translate addresses generated by accelerators and DMA-capable devices into a "canonical" system-wide physical address space. This allows user-space programs to share a virtual address space with a context on the device, but impose a further complexity burden on the underlying OS which must now ensure that IOMMUs are always correctly programmed. This code is fraught with complexity and consequent bugs and vulnerabilities, as it is also intended to provide protection from malicious memory accesses [32–35]. The problem is likely going to get worse with the proliferation of IOMMU designs built into GPUs, co-processors, and intelligent NICs.

Even memory controllers can violate the traditional model. Hillenbrand *et al.* [23] reconfigure memory controller configurations from system software to provide DRAM aliases for mitigating the performance effects of channel and bank interleaving. Proposals for "in-memory" or "near-data" processing [51,56,60] raise further questions for OS abstractions [10] and require a way to unambiguously refer to memory regardless of which module accesses it.

## 3.3 Decoding Nets

A systematic and accurate way to establish canonical names for access-controlled resources that may be referred by different *local* names in different parts of the system is provided by the established *decoding net* [1, 2] model of address translation.

Decoding nets model the addressing structure of a system as a directed graph, where nodes represent (virtual or physical) address spaces or devices (including RAM), and edges the translation of *AS-local* addresses into other address spaces or devices. The graph is a set of nodes, defined as an abstract datatype:

$$\begin{aligned} \mathit{name} &= \text{Name } \mathit{nodeid}\ \mathit{address} \\ \mathit{node} &= \text{Node } \mathbf{accept} :: \{\mathit{address}\} \\ &\qquad\qquad \mathbf{translate} :: \mathit{address} \to \{\mathit{name}\} \end{aligned}$$

The model distinguishes *local* names (*address*), relative to some address space, and *global* names (*name*), which qualify a local name with its enclosing address space. Each node may **accept** a set of (local) addresses (e.g. RAM or memory mapped device registers), and/or **translate** them to one

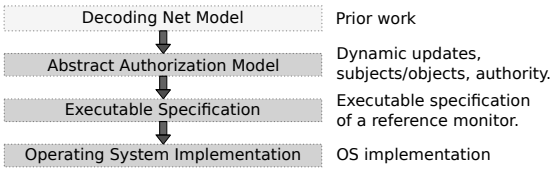



Figure 1: Methodology Overview: Refinement steps.

or more global names (addresses in other address spaces, e.g. MMU or PCI bridges).

This approach dovetails nicely with the reference monitor concept as described above. Every **translate** step corresponds to a *dereference* operation, and any **accept** can be used as a canonical name: the ID of the accepting node, plus the *local* address at which it accepts (e.g. address within a DRAM bank).

Decoding nets have been successfully used to model a wide variety of systems of exactly the sort that is of interest to us, and give a trustworthy, precise guide to where a reference monitor is required: any *configurable* translation node must be treated as part of the distributed reference monitor. It must only be configured such that its local translations are a *projection* of the higher-level security property, exactly as for a processor's MMU. *Static* configuration nodes must be configured in such a way (either by construction or static verification) that their translations are consistent with the projected policy at the point they are applied.

### 3.4 Refinements and Executable Specifications

We borrow our modeling technique, combining *refinement* with *executable specification* from the successful seL4 project. We identify all relevant *objects* (page tables, address spaces, frames, ...), the *subjects* that manipulate them (processes, devices, ...), and which *authority* each subject exercises over each object (e.g. in mapping a frame to a virtual address). These are expressed in an *access-control matrix* (following Lampson [29]) which forms our *abstract specification*, analogous to the high-level *security policy* (integrity) shown to be refined (correctly implemented) all the way down to compiled binaries for seL4 [55].

Again, as in seL4 [15], we next develop an executable specification in Haskell (see § 4.2), expressing subjects, objects, and authority as first-class objects, permitting rapid prototyping without giving up strong formal semantics. Correspondence between abstract and executable models is thus far by inspection and careful construction.

Finally, we show (again with precedent [59]) that the executable model (and hence the abstract model) permits multiple high-performance implementations (see § 5): On Barrelfish, as a representative of partitioned-capability systems including seL4 (capabilities corresponding to *rows* in the matrix), and on Linux, as a representative UNIX-style monolithic kernel (where ACLs correspond to *columns* in the matrix).

### 3.5 Related Work

The seL4 proof [28] assumed a single, fixed, physical address space and a single MMU, and provides no guarantees in the presence of other cores or DMA devices. CertiKOS [22] builds on a model of memory accesses to abstract regions of private, shared or atomic memory, but again provides no proof in the presence of other translation units or cores. Even work on verifying memory consistency in the presence of translation currently only considers the simple case of virtual-to-physical mappings [52].

Graviton [57] provides a trusted execution environment for GPUs requiring all updates to the page tables go through the command processor, acting as a reference monitor for the GPU. Komodo [19] uses ARM TrustZone [6] to implement a software enclave. Both of these works are steps in the right direction, and in this work we extend this approach to the whole system.

OpenCL's Shared Virtual Memory [27], nVidia's CUDA [50] or HSA [24] provide a unified view of memory, ensuring addresses remain valid between CPU and GPU. VAST [30] which uses compiler support to dynamically copy memory to and from the GPU and Mosaic [8], which provides support for multiple sizes of page translation in a shared virtual address space between CPU and GPU. These approaches ensure address consistency in the specific case of CPU–GPU sharing, but are again not whole-system approaches.

In DVMT [3], a customized TLB miss handler implemented as a helper thread installs entries in the TLB using specialized instructions. Similar to the MMU, the OS/hypervisor sets up data structures specifying the policy which mappings the thread is allowed to install. Again this solution focuses on the processor and its MMU.

## 4 Model

A static *decoding net* is a snapshot of the address translation configuration of a system, at a particular moment. We augment the static decoding net with a transition relation, modelling the dynamic reconfiguration of the translation hardware such as when a page table is modified. The allowable transitions express the actions (or *traces*) permitted by the model.

### 4.1 Authority and Dynamic Behavior

The system consists of a set of *address spaces* each having a current *configuration*, which corresponds to a *decoding net* node, that defines the translation of local addresses in this address-space *context*:

$$\mathit{configuration} :: \mathit{address\ space} \rightarrow \mathit{node}$$

This lets us reason about translations with the existing mechanisms available for decoding nets. Hardware constraints,

**Nodes:** $node :: Decoding\ Net\ Node$
**Objects:** $Object = \{name\}$
**Rights:** $Right = Grant \mid Map \mid Access$
**Configuration Space:**
$\texttt{ConfSpace} :: AddressSpace \to \{node\}$
**Address Space Configuration:**
$\texttt{Configuration} :: AddressSpace \to node$
**Access Control Matrix:**
$\texttt{AccessControlMatrix} :: Subject \times Object \to \{Right\}$
**Model State:**
$State = (\texttt{AccessControlMatrix}, \texttt{Configuration})$
**State Transitions:**
$\texttt{ModifyMap} :: Subject \to (name \to \{name\}) \to State \to State$

Figure 2: Model Definition

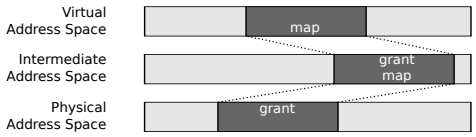



Figure 3: Mappings between address spaces showing grant and map rights of mapped segments.

e.g. an MMU that only supports the translation of naturally aligned 4 KiB blocks of addresses, are expressed as a restriction on the set of possible nodes an address space can map to. This set is the *configuration space* of an address space. Invariant I1 requires that every address space must have a well-defined configuration. The configuration space of a fixed address space is a singleton set.

**Invariant I1 (Well-defined Configuration)**
$\forall a :: AddressSpace.\ \texttt{Configuration}\ a \in \texttt{ConfSpace}\ a.$

**Configuration Authority (Mapping).** The configuration of some address spaces can be changed. The configuration space defines the set of *possible* states an address space may occupy. An *authority* is a subset of configuration transitions, representing what configuration actions a given subject is permitted to take.

Consider Figure 3, representing the general case of an update to an intermediate address space (for example the intermediate physical address, IPA, in a two-stage translation system). We identify two distinct authorities: The **MAP** authority, or the authority to change the meaning of an IPA by changing its mapping; and the **GRANT** authority, or the right to grant **ACCESS** (by mapping) to some range of physical addresses. Note that the 'virtual' and 'physical' address spaces of Figure 3 can be viewed as special cases of an intermediate address space: A top-level 'virtual' address space is simply one to which *nobody* has a **GRANT** authority, and a 'physical' address space e.g. DRAM is one to which there exists no **MAP** authority.

**Right R1 (Grant)**
The right to insert *this* memory object into *some* address space

**Right R2 (Map)**
The right to insert *some* memory object into *this* address space

**Right R3 (Access)**
The right to read or write an object.

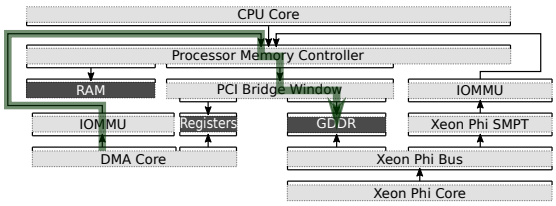



Figure 4: Address spaces in a system with two PCI devices

| *subject / object* | DMA IOMMU | buffer |
|---|---|---|
| IOMMU driver | **MAP** | |
| Xeon Phi process | | **GRANT** |

Table 2: Access control matrix of the Xeon Phi example

**Changing Mappings.** Consider Figure 4, showing the address space configuration of a system with two PCI devices: a DMA engine and an Intel Xeon Phi co-processor. Imagine that we wish to establish a shared mapping to allow a process on a Xeon Phi core to receive DMA transfers (e.g. network packets) into a buffer allocated on the GDDR (following the highlighted path from the DMA core to the GDDR).

The process 'owns' the buffer, and has the ability to call `recv()`, triggering a DMA transfer. In other words, the process has the right to grant **ACCESS** (temporarily) to the DMA core, but it clearly should not have the ability to modify the IOMMU mappings of the DMA core at will. Hence, it does not have the **MAP** authority on the relevant address space.

To change the mappings of an address space, an agent (a *subject*, in standard access-control terminology) needs both the **GRANT** authority on the buffer *object*, and the **MAP** authority on the address space *object*.

The state transition, i.e. changing the *configuration* and therefore how an address space translates addresses, is expressed by the operation `ModifyMap()`: A subject tries to change how a name is being translated by the system, and thus updates its state.

**Authority Representation.** In a monolithic kernel, both these authorities are held (implicitly) by the kernel, which exercises them on behalf of the subjects. It is up to the kernel to maintain accurate bookkeeping to determine whether any such request is safe, typically using an ACL (access-control list) i.e. the *object* lists the subjects and their authorities on it. In a partitioned-capability system such as seL4 or Barrelfish, these authorities are represented by capabilities, handed explicitly to one *subject*, to authorize the operation. In this case, subjects hold the authority on the *object*. These are equivalent from the perspective of access control, differing only in implementation: the same two basic types of authority are present.

The standard representation of authority in systems is an access control matrix [29], such as that of Table 2. This can be read in rows: The IOMMU driver has the **MAP** *capability* to the IOMMU address space, and the process the **GRANT** capa-

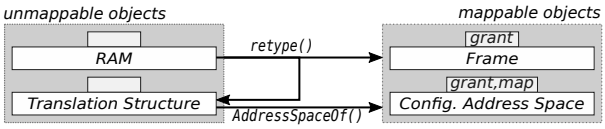



Figure 5: Object Type Hierarchy and possible rights.

bility to the buffer. Alternatively, reading down the columns gives the ACLs: the IOMMU records **MAP** *permission* for the driver, and for the buffer records a **GRANT** *permission* for the process.

**Security Property.** This access control matrix is our abstract model. A system is correct (secure) *statically*, if its current configuration is consistent with the access control matrix. It is secure *dynamically* if any possible transition, beginning in a secure state, must leave the system in a secure state. The access control matrix, together with the configuration space defines the allowable state transitions. The address space must have a valid configuration supported by hardware, and the subject modifying it must have sufficient rights to do so.

## 4.2 Executable Specification

We refine this abstract model into an executable specification of a *reference monitor* [4] for `ModifyMap()`. When composed with the reference monitor **ACCESS** i.e. the MMU, we have our desired compound reference monitor for the fully-dynamic VM system, secure for accesses beginning at any core or device.

This specification serves as an intermediate step between (Figure 1) the abstract model and the concrete OS implementation of the next section, and also an OS-agnostic prototype for implementation in other systems. This approach is inspired by *seL4* [17], which also employed an intermediate Haskell specification to facilitate prototyping.

**Explicit Translation Structures.** We now explicitly represent address translation structures (e.g. page tables, or memory-mapped device registers) as memory objects, without imposing any particular layout on them. This allows us to reason about the manner in with address translation depends on the contents of a memory object (e.g. page tables in RAM, or the contents of device registers).

Once the translation structures are explicit, and noting that these are exactly the reference monitor state we must securely partition, we can state the partitioning invariant (Invariant I2) in terms of implementation-visible objects.

**Invariant I2 (Partitioning)**
No subject has **ACCESS** to a translation object

We model address translation structures as an opaque data type (`TStructure`). This allows us to maintain generality by assuming nothing about their actual inner structure:

```
data Object = RAM {base::Name, size::Natural}
            | Frame {base::Name, size::Natural}
            | TStructure {base::Name, size::Natural}
```

Memory objects form a hierarchy (Figure 5 shows an excerpt) which defines how the different types of objects can be *derived* from each other. For example, in-memory translation structures (**TSTRUCTURE**) are created by retyping **RAM** objects. **RAM** is the base type for *untyped* memory. Retyping **RAM** to a **FRAME** makes it possible to map it into an address space i.e. to **GRANT** access to it. Note, that neither **RAM** nor **TSTRUCTURE** have the **GRANT** right, and therefore these may never become accessible (partitioning).

An address space is derived from (and defined by) a translation structure, and is an explicit object granting the right to map this space into higher-level address spaces (e.g. a second-stage page table defining an IPA space, assigned to the guest-physical address space of a virtualized OS): Figure 3.

```
AddressSpaceOf :: Object -> AddressSpace
```

**Authority and State.** The system is a set of agents, a *mapping database* (MDB) recording the derivation relation between objects, and a set of active address spaces:

```
data KState = KState (Set Agent) MDB (Set AddrSpace)
```

Authority is either directly to an object, or a meta-authority, the right to grant an authority to another. In turn set of such authorities, coupled with an identifier, define an agent.

```
data Authority = Access Object | Map Object
               | Grant Authority
```

**Reference monitor.** The model exposes a set of operations that either change a configuration or access a memory address. The set of permitted operations defines the behavior of the reference monitor. We express this in Haskell as a custom state monad:

```
data Operation a = Operation (State -> (a, State))
  instance Monad (Operation) where ...
```

The reference monitor intercepts operations and verifies that the agent performing the operation has sufficient rights to execute it. We express the changes to the system's state as sequence of operations on the reference monitor, e.g. `retype()` or `map()`, forming a trace of operations:

```
mappingTrace = do
    ...
    -- retype a RAM object to a Frame
    res <- Model.retype RAM Agent Frame Agent
    -- retype another RAM object to a TStructure
    res <- Model.retype RAM2 Agent TStructure Agent
    -- map the frame into the translation structure
    mapping1 <- Model.map TStructure Frame Agent
    ...
```

Model traces are sequences of monitor states, (**KSTATE**), each corresponding to a static decoding net model. Operations include:

- `retype()` converts an existing object into an object of a permissible sub type.
- `map()` installs a mapping in a translation structure.
- `copy()` copies an authority from one subject to another.

**Valid Traces.** Contained within the set $T$ of all possible traces, there is a set of traces $T_V \in T$ that conform to all con-

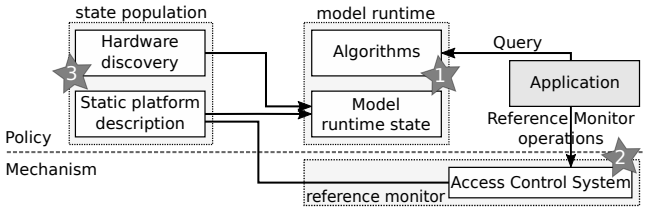



Figure 6: Implementation Overview

straints enforced by the executable specification. We express these traces in the model as sequences of `KState`s. All other traces ($T - T_V$) indicate ending in a failure state (e.g. that execution ended in a state not satisfying the access-control policy).

**Summary.** The executable specification allows us to both simulate and specify sequences of operations such as memory accesses or translation configurations as they would be performed by a concrete OS, implementing the new abstract model.

## 5 Implementation in a real OS

In this section, we describe the implementation of the reference monitor and runtime support libraries and services in two classes of operating systems: a complete implementation in *Barrelfish/MAS*[1] as a representative of a partitoned capability system, derived from the open-source Barrelfish OS [12], and side-by-side a sketch of an implementation within Linux, as a representative of a traditional UNIX-style kernel.

**Architecture Overview.** Figure 6 shows an overview of the resulting architecture. We separate policy and mechanism: (1) at the center is the runtime representation of the model (§ 5.1) which stores the memory topology and provides queries and algorithms for memory allocation policies, (2) the reference monitor which enforces access control and provides the mechanisms for resource management and configuration, and (3) static platform descriptions and dynamic discovery mechanisms (§ 5.3) provide input for the policy and mechanism implementations.

### 5.1 Runtime Support

We implement the runtime representation of the address space model (Figure 6, (1)) in a policy engine. On Barrelfish, this is merged into the Prolog-based system knowledge-base (SKB) [54], which already stores both static and dynamic facts about the system. On Linux, we could use a standalone Prolog instance and run it as a service, or implement the model directly along with other memory allocation policies inside the kernel. We now describe the model representation, its algorithms and potential optimizations.

**Model representation.** We implement the model representation by asserting facts for the accept, translate and overlay

[1] MAS stands for **m**ultiple **a**ddress **s**paces.

```
assert(translate(RegionFrom,RegionTo)).
assert(overlay(NodeFrom,NodeTo)).
assert(accept(Region)).

dn_get_allocation_range(NodeSrc, NodeDst).
dn_get_config_nodes(NodeSrc, NodeDst).
dn_resolve_range(Node, Addr, Size).
dn_resolve_range(NodeSrc, Addr, Size, DstSrc).
```

Listing 1: Prolog Model Representation

constructs of the model (see syntax in [2]). Listing 1 shows the corresponding Prolog rules. This encodes the decoding net, and adds the information to the database.

**Algorithms.** On top of the model encoding, we implement several algorithms, useful for making allocation and configuration policy decisions. For instance, to set up a device, the driver uses the `dn_get_allocation_range()` query to find a suitable address space for memory allocation, then runs `dn_get_config_nodes()` to get the list of address spaces which need to be configured to make the memory resource accessible, and lastly execute `dn_resolve_range()` to obtain the address at which the device sees the memory resource.

The result of the queries is then converted into a sequence of capability operations to allocate memory, setup translation structures and perform the relevant mappings. Note, the model queries only provide a roadmap, the actual reconfiguration steps are invocations of the reference monitor which enforces the authority and integrity of the system following the definition of the executable specification (§ 4.2).

**Optimization.** Running the Prolog queries on the full graph is costly. We provide a library that caches the (flattened) graph representation consisting only of cores/devices, configurable address spaces and memory nodes in the Prolog engine *and* directly in C using adjacency lists. We can then run a shortest-path algorithm to perform the queries, which minimizes the number of address spaces to configure.

### 5.2 Reference Monitor

We now describe the implementation of the reference monitor defined by the executable specification in Linux and *Barrelfish/MAS*.

**Resource Management.** Both, Linux and Barrelfish already have thorough resource management mechanisms, albeit diferent: Barrelfish manages physical resources using a distributed, partitioned capability system for naming, access control, and accounting of objects. As in seL4 [18], capabilities are *typed* to indicate what can be done with the memory they refer to; rules dictate valid *retype* operations (e.g RAM to a Frame). Linux maintains a data structure, the page struct, for every 4 KiB page of memory. In both systems, only the kernel has direct access to those data structures, and can maintain the partitioning invariant.

**Reference Monitor.** As with all microkernels, Barrelfish's kernel is essentially nothing but a reference monitor. It uses the capability system to express the objects in memory and

the authority a process (subject) has over them. Any changes to the translation units (e.g. mapping a memory frame into the IOMMU) correspond to capability operations. The reference monitor checks type, address spaces and rights of the capabilites.

On Linux, we can use the para-virtualization interface (PV-Ops) to implement a reference monitor inside the kernel itself. We can then extend the PV-Ops interface to include all address translation units in the system. This effectively implements a well-defined hypercall interface to request changes to the translation tables from the hypervisor acting as the reference monitor. Similarly, the nested kernel [16] integrates a privileged kernel inside the monolithic kernel which interposes all updates to translation tables. Extending this interface to include all other translation hardware as well, would present a good way to implement a reference monitor inside the Linux kernel.

**Naming of Resources.** Barrelfish's capabilities contain physical addresses to identify the objects they are referring to. To be able to still identify the objects uniquely in the presence of multiple address spaces we change the capability system in *Barrelfish/MAS* to use canonical base names, consisting of an address space identifier and an address within that address space. We adapt the kernel to consider the ASID when performing capability operations. An operation may now fail in new ways, due to incompatible address spaces of the capabilities (e.g. one cannot directly map host physical frame to a guest virtual address).

Linux uses the physical frame number (PFN) uniquely identify every 4 KiB page of memory. Using the sparse memory model [58] or heterogeneous memory [31], we can implement memory nodes (address spaces) a dynamic mapping of the PFN to the underlying page struct. In this manner, we can use the PFN as the memory resource's canonical name.

On both operating systems, we need a function to dereference the canonical name of a resource into a locally valid address. We can *generate* such a translation function based on the platform description or the model state.

**Object Types.** In addition, *Barrelfish/MAS* introduces new capability types for all hardware translation units (not just page tables), ASID allocation, and entire physical, intermediate or virtual address spaces. Like Barrelfish, we allow a capability to refer to a memory region of arbitrary size, but require that it must not span multiple address spaces.

On Linux, we do not need to use typed objects as such as the kernel does not expose handles to physical resources to user space. Internally, Linux already uses different accounting types for memory allocations.

**Page Tables and Address Spaces.** *Barrelfish/MAS* introduces distinct capability types for all hardware-defined translation structures (register sets or page table levels). Each of these capability types are translation structures in the sense of the executable spec. Since a page table defines an address space, we can *derive* an *address space* capability from it, and use it to install mappings in other address spaces. Deleting the page table capability triggers a recursive deletion of its spanned address spaces and all possible mappings. We integrated this process into the capability system. This is effectively equivalent to *revoking* all descendants of the address space capability and then deleting it. This ensures, that there are no mappings referring to an invalid address space.

With the implementation of para-virtualization and KVM-based virtualization, Linux has support to represent the guest address space inside the kernel. This would be one possiblity to get support for different address spaces in the kernel. Alternatively, we can use the sparse memory model or HMM to create "virtual" memory nodes that correspond to an intermediate address space.

**Tracking Mappings.** *Barrelfish/MAS* uses designated mapping capabilities to track mappings. For every mapped object, there is a corresponding mapping capability, which is a descendant thereof. Therefore, the capability system is able to locate and invalidate all mappings when access to an object is revoked. Note, translation structures effectively define an address space, and hence there is no difference between mappings of multi-level page tables, or actual frames.

Similar to the mapping capabilities, Linux uses the `rmap` data structure to store where a page of memory is mapped. This is already maintained for the page cache, as well as guest memory pages. We can use this mechanisms to track all mappings of a page in Linux.

## 5.3 Model Population

The last part of the implementation describes how the model state is populated (③ in Figure 6). There are two major sources of memory topology information building up the runtime representation: *i)* static description of platforms (or parts there of), and *ii)* discovery mechanisms such as PCI or ACPI, which may instantiate predefined descriptions.

**Static Platform Descriptions.** The memory topology of parts of the system – or in the case of SoC the entire system – is fixed and known in advance: for instance, the Xeon Phi co-processor has a defined number of cores and memory. We can therefore write down a description of the memory subsystem. For this, we use a domain specific language (DSL), which follows closely the syntax of the formal model, allows writing down the memory topology of the entire system, or its sub-components. The DSL compiler then produces a set of Prolog rules, which populate the model at runtime, either fully or in response to hardware discovery events. On Linux, we can use `procfs` and `sysfs`, as well as device trees to obtain system topology descriptions.

**Using Static Descriptions: Code Generation.** From the static descriptions, we can pre-compute and enumerate the address spaces of the hardware component, or in the case of SoC platforms, the entire memory topology. The DSL compiler generates a set of data structures and code used by the

reference monitor to instantiate the initial set of capabilities, verify address space compatibility in capability operations, translation tables, or functions to convert the canonical names into valid, local physical or virtual addresses. We evaluate this scenario in § 6.4.

**Using Static Descriptions: Hardware Discovery.** In general, the configuration of a platform is known after device discovery mechanisms such as ACPI or PCI (if percent). During this process, the model is dynamically populated with the partial descriptions of its components: e.g. the ACPI table indicates the presence and version of an IOMMU, and in response the partial description of the IOMMU is instantiated and added to the model at runtime. A driver may update the model with more precise information, e.g. only the Xeon Phi driver knows the precise number of cores and memory size of the PCI Express attached co-processor.

# 6 Evaluation

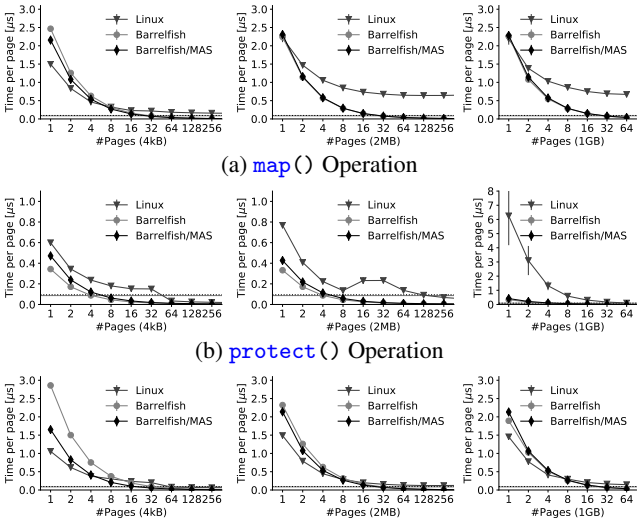
In this section, we present a quantitative and qualitative performance evaluation of the address space and least-privilege authority model in *Barrelfish/MAS*. The goal of this section is to establish the following:

1. The mechanism implementation results in a performant memory system (§ 6.1, § 6.2).
2. The policy implementation produces usable results within reasonable overheads (§ 6.3).
3. Qualitatively demonstrate, that the resulting system is able to handle complex memory topologies (§ 6.4).

**Evaluation Platform.** All performance measurements are performed on a dual-socket server consisting of two Intel Xeon E5-2670 v2 processors (*Ivy-Bridge* micro-architecture) with 10 cores each. The machine has 256 GiB of main memory split equally into two NUMA nodes. The machine runs in "*performance mode*", with *disabled* simultaneous multi-threading (SMT), Intel TurboBoost technology, and Intel Speed Stepping, to ensure consistent measurements. The machine further contains two Intel Xeon Phi co-processor 31S1 attached as a PCI Express 3.0 device. The co-processors have 57 cores with four hardware threads per core, and 8 GiB GDDR memory. The Intel VT-d [26] (IOMMU) is enabled. We use a vanilla Ubuntu 18.04 LTS with Linux kernel 4.15. For a fair comparison we disable specter/meltdown mitigation as they slow down memory operations significantly and Barrelfish doesn't implement them. Barrelfish and *Barrelfish/MAS* are compiled in release mode.

## 6.1 VM Ops - Map/Protect/Unmap

In this part of the evaluation, we quantitatively evaluate the performance of *Barrelfish/MAS*'s virtual memory operations in comparison to vanilla Barrelfish and Linux.



(a) `map()` Operation

(b) `protect()` Operation

(c) `unmap()` Operation

Figure 7: Measured Latency per Page for the VM Operations on Linux, Barrelfish and *Barrelfish/MAS*.

| | `map()` | `protect()` | `unmap()` |
|---|---|---|---|
| 4 KiB page | Linux-shmfd | Linux-shmfd | Linux-shmat |
| 2 MiB large page | Linux-shmat | Linux-mmap | Linux-shmat |
| 1 GiB huge page | Linux-shmat | Linux-shmat | Linux-shmat |

Table 3: The Best Configuration of the Linux VM Operations.

**Benchmark Methodology.** We compare the performance of the virtual memory operations `map()`, `protect()` and `unmap()` for buffer sizes from 4 KiB to 64 GiB using one of the three native supported page sizes (4 KiB, 2 MiB and 1 GiB). On *Barrelfish/MAS* and Barrelfish, we use the default user-level virtual memory management library, and on Linux we take the fastest of the measured different techniques to map memory using anonymous memory (`mmap()`), shared memory objects (`shmfd()`) or shared memory segment (`shmat()`). We exclude the allocation and clearing of backing memory in this benchmark as it affects all systems the same and would dominate the execution times.

**Results.** Figure 7 contains the results of this evaluation for the three operations and page sizes. The graphs show the median latency (lower is better) and standard error per modified page table entry. We scale the number of changed page table entries. For Linux, we select the *best* configuration as indicated in Table 3. We make the following observations:

• *Amortization:* The general pattern is similar: the cost per page decreases with increasing numbers of affected pages. The cost of the virtual region management, syscall overhead, locating the page table entry is amortized among multiple pages, whose mappings are likely to be in consecutive page table entries.

• `map()`. Both, Barrelfish and *Barrelfish/MAS* have matching performance patterns, independent of the used page size. Linux is faster for mapping up to two 4 KiB pages. For larger pages Barrelfish (as well as *Barrelfish/MAS*) outper-

forms Linux. This is not an effect of our implementation but due to Linux allocating lower-level page tables, in case the super-page mapping needs to be broken up. Therefore, Linux allocates and clears memory to hold the page table. Zeroing a page can add up to 0.71$\mu s$ which is the difference we see in the graph. Both, Barrelfish and *Barrelfish/MAS* only have to create a new mapping capability and insert it into the MDB.

• `protect()`. We observe very predictable patterns for Barrelfish and *Barrelfish/MAS*, where vanilla Barrelfish is slightly faster due to storing an explicit pointer to the page table directly in the mapping capability, whereas *Barrelfish/MAS* stores the canonical name which requires an address translation causing more work. In both cases, the mapping capability contains all information to perform the operation. Linux needs to walk the page table to locate the page table entry to be protected. This is again not an effect of the MAS extension but a difference between Linux and vanilla Barrelfish.

• `unmap()`. Up to eight affected pages, Linux is faster than Barrelfish and *Barrelfish/MAS*, which both need to remove and delete the mapping capability from the MDB, which results in another syscall on Barrelfish (*Barrelfish/MAS* removes this when clearing the page table entry). Removing the mapping capability gets amortized when more pages are affected.

**Discussion.** In direct comparison with Barrelfish, we observe that *Barrelfish/MAS* is able to match the performance in all cases. Moreover, the comparison with Linux shows, that *Barrelfish/MAS* has comparable performance to a mainstream OS. We conclude that our least-privilege access control model with support for multiple address spaces can be implemented with fine granularity while maintaining competitive memory management performance.

## 6.2 VM Ops - Appel-Li Benchmark

The Appel-Li benchmark [5] exercises the virtual memory subsystem with operations, which are relevant to tasks such as garbage collection or tracking page modifications.

**Benchmark Methodology.** The benchmark consists of the following three experiments:

1. *prot1-trap-unprot.* Randomly pick a page of memory, write-protect the page, write to it, take a trap, unprotect the page, continue with next page.
2. *protN-trap-unprot.* Write-protect 512 pages of memory at once, write to each page of memory in turn, taking a trap and unprotecting the page.
3. *trap only.* Pick a protected page, write to it and take the trap continue with next page without changing any permissions.

We run this benchmark on Barrelfish and *Barrelfish/MAS*. In addition, we compare to Linux as a frame of reference. On Barrelfish and *Barrelfish/MAS* the numbers include the cost of virtual address space accounting in userspace.

**Results.** We show the benchmark results in Figure 8. Each

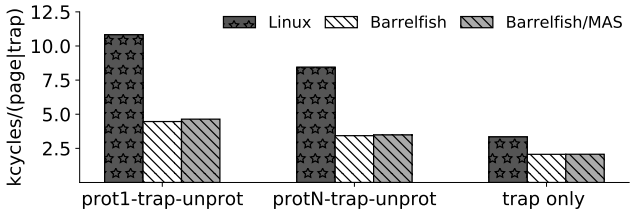



Figure 8: Appel-Li Benchmark on *Barrelfish/MAS* and Linux.

bar corresponds to a different OS and represents the time taken per page. The three bar groups represent the three benchmark experiments. The standard error is less than 0.5%. We make the following observations:

• *Barrelfish vs. Barrelfish/MAS.* Direct comparison shows a slowdown of less than 5% for *Barrelfish/MAS* vs. Barrelfish. The trap performance of both systems is the same.

• *Linux vs Barrelfish.* Barrelfish outperforms Linux in all experiments. Barrelfish can use its capability system to efficiently find the page table that has to be modified while Linux needs to walk the page table tree. Furthermore Barrelfish reflects the trap directly to user-space without checking whether the faulting address has been previously allocated [9]. This applies to *Barrelfish/MAS* as well as vanilla Barrelfish and is independent of our extension.

• *Batching.* The protection of 512 pages in one syscall (*protN-trap-unprot*) amortizes the total syscall overheads, which reduce the time per page on all systems by 600-2000 cycles.

**Discussion.** In this evaluation, we show that *Barrelfish/MAS* is able to match the performance of Barrelfish with a maximum overhead of less than 5%, despite support for explicit address spaces. The comparison to Linux again shows that *Barrelfish/MAS*'s memory operation performance is competitive to that of a mainstream OS.

## 6.3 Dynamic Updates of Translation Tables

In this evaluation, we investigate the overheads of the model runtime representation and the translation unit reconfiguration following the principle of least-privilege.

**Benchmark Methodology.** This benchmark models an offload-scenario, where an application workload wants to make use of a co-processor attached to PCI Express. We use the Xeon Phi co-processor for this purpose. We are interested in the sequence of initialization steps to establish a shared buffer between the CPU cores and the co-processor:

1. *Model Query.* Evaluate the runtime representation to find a suitable memory region and needed re-configuration steps.
2. *Allocate and Map.* Request memory from the allocator and map it into the application's virtual address space.
3. *Program Translation Units.* Re-configure the translation units indicated in the model query response. Here, this includes *i)* the IOMMU, and *ii)* the SMPT of the co-processor.

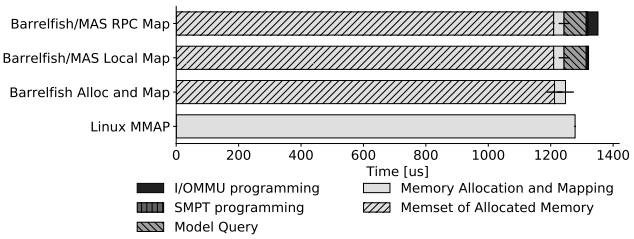



Figure 9: Breakdown of the Offloading Scenario.

We profile the execution of these steps and measure the time it takes to perform each step individually. We evaluate two mechanisms to program the IOMMU, *i)* to use capability invocations directly, and *ii)* use an RPC to the IOMMU service acting as a reference monitor. The buffer size used is 8 MiB. As a frame of reference, measure the time it takes to just allocate and map memory on both Linux (using `mmap()`) and vanilla Barrelfish.

**Results.** The breakdown of the operation into the steps is shown in Figure 9. We show both the numbers for both mechanisms to program the IOMMU, and for comparison, we include the time it takes to just allocate and map the memory on vanilla Barrelfish and Linux. The x-axis represents the measured times in $\mu s$. We make the following observations:

• *Memory Allocation and Mapping.* All three OSes use about the same time to allocate and map the required memory region, which accounts for the majority of the profiled time. It is dominated by zeroing the newly allocated memory.

• *Model Query.* Evaluating the model at runtime accounts for less than 5% of the total runtime.

• *SMPT Configuration.* Programming the SMPT of the co-processor uses less than 0.3% of the runtime.

• *IOMMU Programming.* The configuration of the IOMMU using direct capability invocations is fast (0.2% of the runtime). When using the RPC to the IOMMU reference monitor, this requires capability transfers which corresponds to about 3% of the execution time.

Overall, the resulting overhead for the model query and the address space configuration accounts for 5.7%. There is no significant difference in the memory allocation and mapping times of *Barrelfish/MAS* compared to Barrelfish and Linux.

**Discussion.** In this evaluation, we have shown that it is possible to efficiently implement a representation of our executable model in an operating system and reconfigure address spaces following the principle of least-privilege. Moreover, subsequent allocations may use the cached results of the model query, reducing the overhead even further. Note, that the query merely indicate the operations to be carried out, but the capability system enforces the integrity thereof.

## 6.4 Correctness on Simulated Platforms

In this evaluation, we qualitatively show the application and integration of the address space model into the OS toolchain

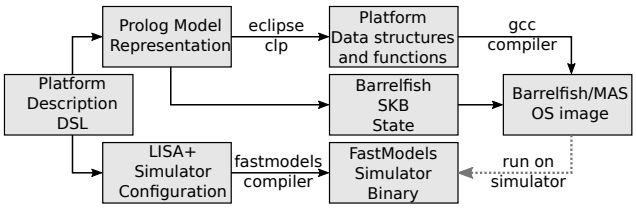



Figure 10: Running *Barrelfish/MAS* on an ARM FastModels [7] Platform Based on a Hardware Description.


Figure 11: FastModels Simulator Configuration

to *generate* low-level, platform-specific OS code and data structures. By doing that we show, that our implementation is functional even when run on simulated platforms with unusual address space topologies not supported by other systems. While these simulated platforms are extreme, they include other real systems such as those with secure co-processors.

**Evaluation Methodology.** We design and build the toolchain illustrated in Figure 10 and write a series of different platform descriptions using a DSL. These platform descriptions then specify the memory topology of the simulated platforms. The DSL compiler then generates:

1. *Executable Model.* A runtime representation of the memory topology model, and
2. *Simulator Configuration.* The LISA+ hardware description that configures the ARM FastModels simulator [7].

The generated runtime representation of the topology model then acts as the initial state for the Barrelfish SKB, and is used to generate low-level OS code and data structures, which are compiled and linked into a platform-specific *Barrelfish/MAS* OS image.

We mention four example configurations we tested for this evaluation. Figure 11 shows an illustration of the simulated platform, which consists of two ARM Cortex A57 processors, each having a configurable local memory map which defines at which addresses they see the DRAM regions (and the rest of the system in general) in their local address space. We evaluated the following configurations:

1. *Uniform* Both cores have an identical memory map.
2. *Swapped* DRAM is split in two halves, where each core sees the two halves at swapped address ranges.
3. *Private* One shared memory region, and each core further has a private memory region, inaccessible by the other.
4. *Private Swapped* Combines the *swapped* and *private* setups: shared memory with swapped views, and private memory per core.

**Results.** During out experiments, we managed to compile *Barrelfish/MAS* and run it successfully on all tested platform

configurations. This includes various memory management tasks and shared-memory message passing between the cores. There was no programmer effort required, besides writing the platform description.

**Discussion.** We know of no other current OS designs which can manage memory globally in all these cases. Popcorn Linux [11] and Barrelfish have limited support for case 3; while regular Linux and seL4 only support case 1. In contrast, *Barrelfish/MAS* supports all four cases.

*Barrelfish/MAS* is able to boot and manage memory on all platforms without modifications, regardless of the topology.

### 6.5 Evaluation Summary

In this evaluation, we have shown that it is possible to efficiently implement the address space model and least-privilege memory management in an OS. We have quantitatively evaluated *Barrelfish/MAS*'s virtual memory system, the reconfiguration operations, and analyzed the space and runtime complexity of maintaining kernel state.

Moreover, we have seen that *Barrelfish/MAS* is able to handle complex and non-standard memory topologies by strictly using the memory object's canonical name in the capability system, and generated translation functions which further convert this canonical name to a valid local address

## 7 Conclusion

In this paper, we made the case to bring back the concept of a reference monitor to mediate access to memory resource on modern, heterogeneous platforms. We presented a fine-grained, realistic memory protection model based on which we can extend the reference monitor to include all memory translation and protection hardware present in the system. This allows systems software to adapt their access control model and catch up with the complexity of modern hardware.

We have shown that our design is applicable to any OS, regardless of its architecture. We have developed an executable specification of a reference monitor including the state, operations and authority, on which we have based our prototype implementation in *Barrelfish/MAS*. Not only can this memory protection model eliminate three different classes of bugs and vulnerabilities, but there is also no inherent performance overhead in implementing it in an operating system. Moreover, based on trusted hardware specifications we can increase the level of automation and generate low-level operating systems code. We believe that our approach can lay the foundation for both fully verified systems and more reliable memory management in existing systems.

We plan to open-source the reference monitor and *Barrelfish/MAS* implementations.

## References


[1] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. Formalizing Memory Accesses and Interrupts. In *Proceedings of the 2nd Workshop on Models for Formal Analysis of Real Systems*, MARS 2017, pages 66–116, 2017.

[2] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. Physical Addressing on Real Hardware in Isabelle/HOL. In *Proceedings of the 9th International Conference on Interactive Theorem Proving*, ITP'18, pages 1–19, Oxford, United Kingdom, 2018. Springer International Publishing.

[3] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 457–468, New York, NY, USA, 2017. ACM.

[4] James P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Vol. I, AD-758 206, Electronic Systems Division, Deputy for Command and Management Systems HQ Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, Massachusetts 01730, USA, 10 1972.

[5] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 96–107, New York, NY, USA, 1991. ACM.

[6] ARM Ltd. *ARM Security Technology - Building a Secure System using TrustZone Technology*, prd29-genc-009492c edition, 4 2009.

[7] ARM Ltd. Development Tools and Software: Fast Models. https://www.arm.com/products/development-tools/simulation/fast-models, 8 2019.

[8] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. Mosaic: A GPU Memory Manager with Application-transparent Support for Multiple Page Sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 136–150, New York, NY, USA, 2017. ACM.

[9] Moshe Bar. The Linux Signals Handling Model. *Linux Journal*, 5 2000. https://www.linuxjournal.com/article/3985.

[10] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. It's Time to Think About an Operating System for Near Data Processing Architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 56–61, New York, NY, USA, 2017. ACM.

[11] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 29:1–29:16, New York, NY, USA, 2015. ACM.

[12] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[13] Simon Biggs, Damon Lee, and Gernot Heiser. The jury is in: Monolithic os design is flawed: Microkernel-based designs improve security. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Adam Chester. Exploiting CVE-2018-1038 - Total Meltdown. Online. https://blog.xpnsec.com/total-meltdown-cve-2018-1038/, 4 2018.

[15] David Cock, Gerwin Klein, and Thomas Sewell. Secure Microkernels, State Monads and Scalable Refinement. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 167–182, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 191–206, New York, NY, USA, 2015. ACM.

[17] Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. Running the Manual: An Approach to High-assurance Microkernel Development. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 60–71, New York, NY, USA, 2006. ACM.

[18] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the sel4 microkernel. In *Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE '08, pages 99–114, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using Verification to Disentangle Secure-enclave Hardware from Software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 287–305, New York, NY, USA, 2017. ACM.

[20] Simon Gerber, Gerd Zellweger, Reto Achermann, Kornilios Kourtis, Timothy Roscoe, and Dejan Milojicic. Not Your Parents' Physical Address Space. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 16–16, Berkeley, CA, USA, 2015. USENIX Association.

[21] Xiling Gong. Exploiting Qualcomm WLAN and Modem Over the Air. In *Proceedings of the BlackHat USA 2019*, 2019.

[22] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 653–669, Berkeley, CA, USA, 2016. USENIX Association.

[23] Marius Hillenbrand, Mathias Gottschlag, Jens Kehne, and Frank Bellosa. Multiple Physical Mappings: Dynamic DRAM Channel Sharing and Partitioning. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 21:1–21:9, Mumbai, India, 2017.

[24] HSA Foundation. *HSA Runtime Programmer's Reference Manual*, version: 1.1.4 edition, 10 2016.

[25] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. An Evolutionary Study of Linux Memory Management for Fun and Profit. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 465–478, Berkeley, CA, USA, 2016. USENIX Association.

[26] Intel Corporation. *Intel Virtualization Technology for Directed I/O - Architecture Specification*, d51397-011, revision 3.1 edition, 6 2019.

[27] Khronos OpenCL Working Group. *The OpenCL Specification*, version: 2.1, document revision: 24 edition, 2 2018.

[28] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[29] Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[30] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. VAST: The Illusion of a Large Memory Space for GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 443–454, New York, NY, USA, 2014. ACM.

[31] Linux Kernel Documentation. *Heterogeneous Memory Management (HMM)*, version 5.0 edition, 4 2019.

[32] A Theodore Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon W Moore, and Robert NM Watson. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals. In *NDSS*, 2019.

[33] Alex Markuze, Adam Morrison, and Dan Tsafrir. True IOMMU Protection from DMA Attacks: When Copy is Faster Than Zero Copy. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 249–262, New York, NY, USA, 2016. ACM.

[34] Benot Morgan, Eric Alata, Vincent Nicomette, and Mohamed Kaaniche. Bypassing IOMMU Protection against I/O Attacks. In *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, pages 145–150, 10 2016.

[35] Benot Morgan, Eric Alata, Vincent Nicomette, and Mohamed Kaaniche. IOMMU Protection Against I/O Attacks: A Vulnerability and a Proof of Concept. *Journal of the Brazilian Computer Society*, 24(1):2, 1 2018.

[36] NATIONAL VULNERABILITY DATABASE NVD. CVE-2011-1898. Online, 8 2011.

[37] NATIONAL VULNERABILITY DATABASE NVD. CVE-2013-4329. Online, 9 2013.

[38] NATIONAL VULNERABILITY DATABASE NVD. CVE-2014-0972. Online, 8 2014.

[39] NATIONAL VULNERABILITY DATABASE NVD. CVE-2014-3601. Online, 8 2014.

[40] NATIONAL VULNERABILITY DATABASE NVD. CVE-2014-9888. Online, 8 2014.

[41] NATIONAL VULNERABILITY DATABASE NVD. CVE-2015-6994. Online, 1 2017.

[42] NATIONAL VULNERABILITY DATABASE NVD. CVE-2016-5349. Online, 4 2017.

[43] NATIONAL VULNERABILITY DATABASE NVD. CVE-2017-12188. Online, 10 2017.

[44] NATIONAL VULNERABILITY DATABASE NVD. CVE-2018-1038. Online, 8 2018.

[45] NATIONAL VULNERABILITY DATABASE NVD. CVE-2015-4421. Online, 5 2019.

[46] NATIONAL VULNERABILITY DATABASE NVD. CVE-2015-4422. Online, 5 2019.

[47] NATIONAL VULNERABILITY DATABASE NVD. CVE-2019-10538 - Modem into Linux Kernel issue. Online, 8 2019.

[48] NATIONAL VULNERABILITY DATABASE NVD. CVE-2019-10539 - Compromise WLAN Issue. Online, 8 2019.

[49] NATIONAL VULNERABILITY DATABASE NVD. CVE-2019-10540 - WLAN into Modem issue. Online, 8 2019.

[50] NVIDIA Corporation. *Unified Memory in CUDA 6*, 11 2013.

[51] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent RAM. *IEEE Micro*, 17(2):34–44, 3 1997.

[52] Bogdan F. Romanescu, Alvin R. Lebeck, and Daniel J. Sorin. Specifying and Dynamically Verifying Address Translation-aware Memory Consistency. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 323–334, New York, NY, USA, 2010. ACM.

[53] Pierre Schnarz, Joachim Wietzke, and Ingo Stengel. Towards attacks on restricted memory areas through co-processors in embedded multi-os environments via malicious firmware injection. In *Proceedings of the First Workshop on Cryptography and Security in Computing Systems*, CS2 '14, pages 25–30, New York, NY, USA, 2014. ACM.

[54] Adrian Schüpbach, Andrew Baumann, Timothy Roscoe, and Simon Peter. A Declarative Language Approach to Device Configuration. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 119–132, New York, NY, USA, 2011. ACM.

[55] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 Enforces Integrity. In Markovan Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving*, pages 325–340, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[56] Erik Vermij, Leandro Fiorin, Rik Jongerius, Christoph Hagleitner, Jan Van Lunteren, and Koen Bertels. An architecture for integrated near-data processors. *ACM Trans. Archit. Code Optim.*, 14(3):30:1–30:25, September 2017.

[57] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on gpus. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 681–696, USA, 2018. USENIX Association.

[58] Andy Whitcroft. Sparsemem Memory Model. https://lwn.net/Articles/134804/, 8 2019.

[59] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. Mind the Gap. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 500–515, Berlin, Heidelberg, 2009. Springer-Verlag.

[60] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 85–98, New York, NY, USA, 2014. ACM.

[61] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. Understanding the security of discrete gpus. In *Proceedings of the General Purpose GPUs*, GPGPU-10, pages 1–11, New York, NY, USA, 2017. ACM.