Verified Protection Model of the seL4 Microkernel

Dhammika Elkaduwe, Gerwin Klein and Kevin Elphinstone

NICTA[1] and University of New South Wales
Sydney, Australia
{dhammika.elkaduwe|gerwin.klein|kevin.elphinstone}@nicta.com.au

Abstract. This paper presents a machine-checked high-level security analysis of seL4 — an evolution of the L4 kernel series targeted to secure, embedded devices. We provide an abstract specification of the seL4 access control system in terms of a classical take-grant model together with a formal proof of its decidability. Using the decidability property we show how confined subsystems can be enforced. All proofs and specifications in this paper are machine-checked and developed in the interactive theorem prover Isabelle/HOL.

1. Introduction

This paper presents a machine-checked high-level security analysis of seL4 [DEK+06, EDE07]. The seL4 microkernel is an evolution of the L4 kernel series [Lie95] targeted to secure, embedded devices.

It does not need to be argued that embedded systems have become an integral part of our lives. They are increasingly deployed in safety- and mission-critical scenarios. At the same time, embedded devices are becoming multipurpose appliances, exemplified by mobile phones, PDAs, entertainment devices and set-top boxes. They feature millions of lines of software, installed for various purposes and therefore, with varying degrees of assurance and diverse resource requirements, developed on a tight budget. They feature untrusted third-party software components, applications, and even whole operating systems (such as Linux) that can be installed by the manufacturer, suppliers and even the end user.

Microkernels are a promising approach with renewed industry interest to improving the security and robustness of components on these devices. They provide strong isolation guarantees between components — misbehaviour of a component is confined within the component itself.

The success of this approach depends, to large degree on the microkernel's ability to enforce isolation between components. In this paper, we analyse the seL4 kernel primitives and affirm that they are sufficient to enforce isolation. Moreover, we provide some examples on how these mechanisms can be used in practice. If the restrictions that guarantee isolation are overly restrictive, then such a kernel will not be pragmatic. Through these examples, we show that this is not the case in seL4.

All formal definitions, lemmas, theorems, and examples we present are machined-checked using the theorem prover Isabelle/HOL [NPW02]. To our knowledge this constitutes the first machine-checked specification and the first machine-checked proof of a take-grant [LS77] model. It is also the first direct application of the take-grant model to a specific microkernel API.

Beyond this paper, it is our longer term goal to formally connect the security proof presented here with the actual kernel implementation [EKD+07, EKK06]. Our current work on this formal refinement proof has influenced some of the design choices in our specification. We briefly sketch our proof plan for refinement in Sect. 5.2.

The remainder of this paper is structured as follows. In Sect. 2 we describe the relevant parts of the seL4 microkernel, in particular its physical memory management system [EDE07] which is directly coupled with the access control model. Sect. 3 introduces the classical take-grant access control model and how seL4 deviates from the classical model. In Sect. 4 we define the notation that we use in Sect. 5 to present the abstract specification of

[1] NICTA is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.
Correspondence and offprint requests to: Dhammika Elkaduwe, Gerwin Klein and Kevin Elphinstone

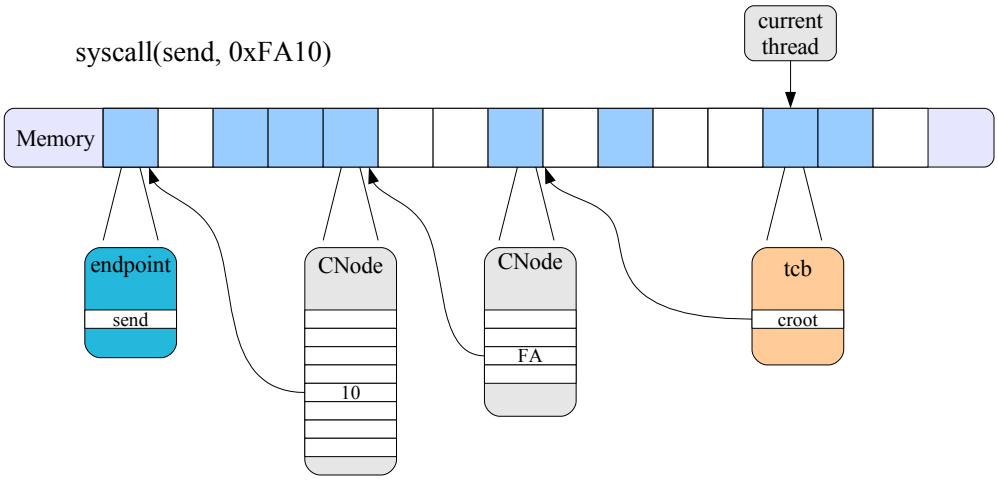


Fig. 1. Example seL4 system call

the seL4 access control model. Sect. 6 then concerns the formulation and proof of decidability [HRU76] of our protection system, and in Sect. 7 we prove mandatory isolation between subsystems and provide some example systems. Sect. 8 discusses related work, and Sect. 9 concludes.

2. The seL4 Microkernel

The seL4 (*secure embedded L4*) kernel is an evolution of the L4 [Lie96] microkernel; as the name implies it is targeted to secure embedded devices. It aims to develop an efficient and flexible kernel with assurance of its ability to enforce security policies such as isolation, spatial partitioning, and a proof of implementation correctness of the kernel against the abstract model used in the security proof.

seL4 employes a capability [DVH66] based protection system that is inspired by early hardware-based capability machines such as CAP [NW77] where capabilities control access to physical memory, the KeyKOS and EROS systems [Har85, SSF99] with their controls on dissemination of capabilities and the take-grant model [LS77]. A detailed description of the seL4 API can be found elsewhere [NIC06, DEK+06, EDE07]. In this section, we provide an overview of the relevant parts of the seL4 kernel.

2.1. Overview

Similar to the L4 microkernel, seL4 provides three basic abstractions: threads, address spaces and inter-process communication (IPC). In addition, seL4 introduces an abstraction of so-called *untyped memory*. Roughly speaking, untyped memory stands for a region of currently unused physical memory. We will describe the concept in more detail further below.

All kernel abstractions are provided via named, first-class kernel objects. Each kernel object implements a particular abstraction and supports one or more operations. Authorised users can obtain kernel services by invoking operations on kernel objects.

Authority over these objects is conferred via capabilities [DVH66], specifically, *partitioned capabilities*. Capabilities are stored inside kernel objects called *CNodes*. These can be seen as tables of capabilities, which may be inspected and modified only via invocation of the CNode object itself. Through this mechanism they are guarded against user tampering. CNodes are similar to KeyKOS *nodes* [Har85], except that they are composed similar to *guarded page tables* [Lie94] to form a local capability address space.

Capabilities in seL4 are immutable. While user-level programs may specify some of its properties at the time the capability is created, those properties may only be changed by removing the capability and replacing it with another.

All system calls are invocations of capabilities. System call arguments can either be data or other capabilities required to complete the system call. Users specify a capability as an index into a local capability address space that will translate the given index to a capability. Threads have no intrinsic authority beyond what they possess as capabilities.

Fig. 1 shows an example of a thread invoking a system call. As arguments to the system call the thread provides the operation to perform (encoded as an integer, in Fig. 1 the `send` operation), a capability index (in the example

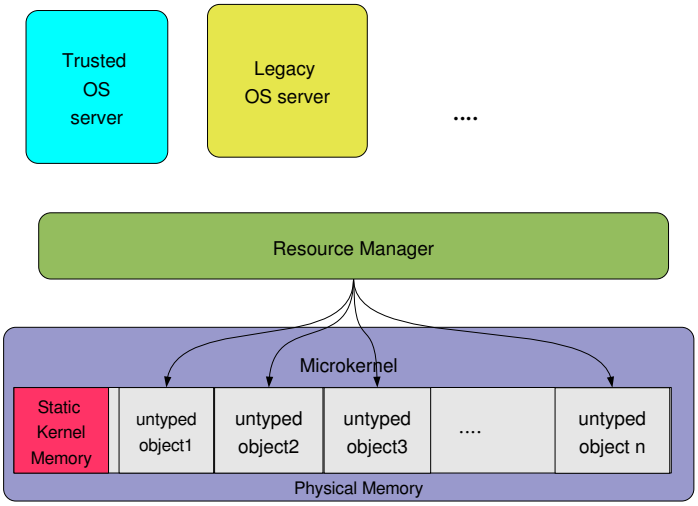


Fig. 2. Sample System Configuration

`0xFA10`) that names the capability and thereby the kernel object to invoke, and possibility further arguments of the operation consisting of either data or more capability indices, depending on the system call. Associated with each thread is a capability address space or a *CSpace*. The CSpace defines a local capability name space for each thread; it maps capability indices to capabilities. Upon receiving the request, the kernel locates the root of the CSpace associated with the currently running thread. This root, located in its thread control block (TCB), is a pointer to a CNode kernel object. Roughly speaking,[2] a CNode translates parts of of the capability index. In the example, the first CNode translates the `FA` part of the index, leading to another CNode that for `10` finally returns a pointer to the kernel object that the capability names (in the example, this is an IPC endpoint, supporting the `send` operation). The set of CNodes reachable by lookup from the root defines the CSpace. Each lookup checks whether the capability possesses sufficient authority to complete the requested operation. If so, the object named by the capability is invoked with the rest of the arguments provided in the system call. On the other hand, if the capability does not possess sufficient authority the kernel, depending on the operation, either notifies a designated thread or drops the operation silently.

The authority a thread possesses can be modified by means of system call operations on CNodes. These system calls are in turn authorised by the current distribution of capabilities, using the same mechanism explained above. The main question answered in this paper is how to model and control the distribution of authority in seL4.

The seL4 kernel uses a variant of the take-grant model to control the dissemination of authority via capabilities. Sect. 3 discuss the take-grant model and the modifications introduced by seL4. In Sect. 5 we formalise this model in Isabelle/HOL which is then used in our analysis in Sect. 6 and Sect. 7.

2.2. Memory Allocation Model

An important part of the seL4 design is that all memory — be it the memory directly used by an application (e.g. memory frames) or indirectly in the kernel (e.g. page tables), is fully accounted for by capabilities. The motivation for this allocation scheme can be found elsewhere [EDE07]. Here, we explain at an example how the system works.

At boot time, seL4 preallocates all memory required for the kernel to run — space for kernel code, data, and kernel stack. As shown in Fig. 2 the remainder of memory is divided into *untyped memory* (UM) objects. The initial thread, the *resource manager*, has full authority over these UM objects. The resource manager is responsible for enforcing a suitable resource management policy and for boot strapping the rest of the system.

A parent capability to untyped memory can be refined into child capabilities to smaller sized untyped memory blocks or into other kernel objects via the *retype* operation on UM objects. Retype has the following two restrictions:

1. the refined child capabilities must refer to non-overlapping UM objects of size less than or equal to the original, and
2. the parent capability must have not currently posses any previously refined child capabilities.

To guarantee the integrity of kernel objects, a region of memory must implement a single kernel object type at

[2] The seL4 implementation uses guarded page tables, a very similar, but slightly more complex mechanism that allows to store large CSpaces efficiently.

a time. The restrictions above ensure that there are no overlapping or previously allocated objects within the given region.

Once a kernel object is created via the retype operation, the creator has full authority over the created object. What "full authority" means depends on the object type. For example, if the created object is untyped memory, full authority means authority to retype, for a frame it is {read, write}, for an IPC endpoint it is {send, receive} and so on. The creator can then delegate all or part of the authority it possesses over the object to one or more of its clients. This is done by granting the client a capability to the object with possibly diminished access rights.

The example in Fig. 2 illustrates a sample system architecture, with a domain specific resource manager running at user-level, receiving authority to remaining untyped memory after boot strapping. The resource manager has the freedom to apply any policy depending on the domain, such as subdividing untyped memory for delegation to any guest OS, or withholding memory and providing an interface to applications for them to request specific OS services.

Applications are at liberty to use a suitable policy to manage the available untyped memory. This can be a simple static or a complex dynamic policy. In the above example, the legacy OS server might employ a complex and error prone policy to manage its UM objects, in contrast to a simple static policy used by the resource manager. However, since the legacy OS can not exceed the authority it possesses, any misbehaviour of the legacy OS is isolated from the rest of the system.

As a result, different allocation policies over the kernel's meta data can be enforced from user space, without modifying the kernel code. In our context, this is desirable — depending on the domain, a suitable allocation policy can be enforced without modifying the verified kernel code base.

In summary, there is no implicit memory allocation within the kernel; all allocation is explicit by user request via capability invocations. Restricting the number of UM capabilities possessed by an application guarantees the precise amount of physical memory that can be directly or indirectly consumed by a subsystem. That means, by construction, the question of partitioning hardware resources, in particular physical memory, is a question of authority distribution and with that, capability distribution, only. The next sections show how authority distribution is modelled and controlled.

3. The Take-Grant Model

Access control models (also 'protection models') provide a formalism and framework for specifying, analysing and implementing security policies. Such a model consists of: (a) finite set of access rights, and (b) a finite set of rules for modifying the distribution of these access rights. The *safety analysis* then determines, for the given set of rules in the model and an initial distribution of access rights, whether or not it is possible to reach a state in which a particular access right α is acquired by a subject that did not possess it initially. If there exists an algorithm that decides safety, then the model is said to be decidable.

The classical access model used in capability systems is the take-grant model, originally proposed by Lipton and Snyder [LS77] and later enhanced by many authors [BS79,Sny81,Bis81,Bis96,SW00]. The model is decidable in linear time [LS77]. In this section we provide an overview of the classical take-grant model and the modification introduced by the seL4.

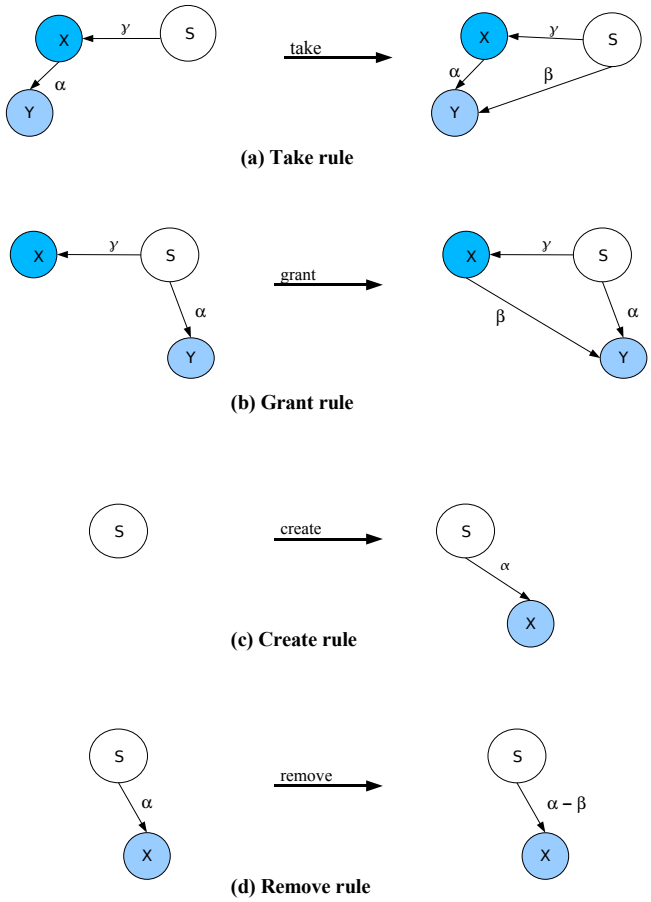
3.1. The Classical Take-Grant Model

The take-grant model represents the protection state of the system as a directed graph; where nodes represent subjects or objects in the system and the labelled, directed arcs represent authority — capabilities, possessed by a subject over the object pointed by the arc. An out going arc from X to Y with label α means X possess α authority over the object Y (see Fig. 3).

If the set of of access rights of the system is R, then any label α on an arc is a nonempty subset of R ($\alpha \subseteq R$). While R might vary, there are two rights that occur in every instance of the model: the take right (t) and the grant right (g). They play a special role in dissemination of authority.

System operations that modify the authority distribution are modelled as graph mutation rules. There are a number of such rules, but the most common ones are *take, grant, create* and *remove*. Take and grant rules propagate existing authority from one node to another. The create rule adds a node to the graph and an arc connecting the new node to an existing one. The remove rule takes away part of the access rights from an arc or removes the arc entirely. Following is a description of the graph rewriting rules:

- **take rule:** Let S, X, Y be three distinct vertices in the protection graph. Let there be an arc from X to Y labelled α, and from S to X with a label γ such that $t \in \gamma$. Then the take rule defines a new graph by adding an edge from S to Y with the label $\beta \subseteq \alpha$. Part (a) of Fig. 3 represents an application of the take rule.
- **grant rule:** Let S, X, Y be three distinct vertices in the protection graph. Let there be an arc from S to Y labelled α, and from S to X with a label γ such that $g \in \gamma$. Then the grant rule defines a new graph by adding an edge from X to Y with the label $\beta \subseteq \alpha$. Part (b) in Fig. 3 is a graphical illustration of this rule.
- **create rule:** Let S be a vertex in the protection graph. Then the create rule defines a new graph by adding an new node X and a arc from S to X with a label α. Part (c) of Fig. 3 represents an application of this rule.



(a) Take rule

(b) Grant rule

(c) Create rule

(d) Remove rule

Fig. 3. Take-Grant Rules

- **remove rule:** Let S, X be vertices in the protection graph. Let there be an arc from S to X with a label α. Then the remove rule defines a new graph by deleting β labels from α. If $\alpha - \beta = \{\}$ then the arc itself is removed. The operation is shown in part (d) of Fig. 3.

Lipton and Synder [LS77] showed that safety is decidable for this model. Bishop et. al. [BS79] later extended the analysis to cover *de facto* rights — access rights implied by a given distribution of authority. For example, assume that there is a subject S_1 with write authority over subject S_2. Moreover, S_2 has write authority to subject S_3. Then, S_1 possesses a de facto write authority to subject S_3; because data that S_1 writes to S_2 can then be transfered to S_3 via S_2. S_1 does not possess direct (de jure) write authority, nor is there any take or grant authority propagation necessary for S_1 to indirectly write to S_3. The above example is an instance of the *find* rule defined in [BS79]. Our analysis below is concerned with authority propagation and memory separation, and hence mainly with de jure authority. The analysis could itself be seen as computing an approximation of the de facto grant right distribution. The issue of other de facto rights is orthogonal and our analysis should extend easily to it.

3.2. The seL4 Take-Grant Model

The seL4 access control model is a variant of take-grant. It modifies the classical model in several aspects. While the formal access control model of seL4 is the focus of Sect. 5, in this section we provide an informal description of the modifications introduced by seL4.

The most significant of these modifications is the create rule. Adding a new node to the protection graph corresponds to allocating a new object in the concrete kernel. As we mentioned in Sect. 2.1 the authority to allocate is conferred by untyped capabilities. As such, the create rule is only applicable if there is an outgoing arc with *create* authority which is represented by label c. All other operations do not consume additional resources. All resources required for the functionality of an object are pre-allocated at the time of object creation.

The second modification is that our remove rule is simpler than that of the take-grant model. It removes the capability, that is, the whole arc, instead of removing part of its label. Our motivation for this changes is that in the

concrete kernel capabilities are immutable. The only way to diminish authority (the label of an arc) is by removing the capability and creating a new one with diminished authority.

In addition to removing a single capability, the seL4 kernel provides an operation called *revoke* which removes a set of capabilities. In practice, this mechanism is applied to re-use resources previously handed out to a subsystem. Since the kernel keeps track of what authority was derived from each untyped capability, the supervisor can remove all authority from a subsystem in one operation by revoking the untyped capability from which the subsystem was created. None of the graph rewriting rules in take-grant represent the revoke operation. However, it can be simply be thought of as a set of multiple applications of remove.

The take rule of take-grant is not employed by the seL4 kernel. All authority propagations are grant operations. The transfer of a capability is authorised by the capabilities possessed by the sender. As such, there is no explicit take (t) right in our model.

4. Notation

This section introduces the notation used in the formal parts of the paper below. We directly use the notation of the Isabelle/HOL theorem prover [NPW02] which for the most part coincides with standard mathematics. Isabelle is a generic system for implementing logical formalisms, and Isabelle/HOL is the specialisation of Isabelle for Higher Order Logic.

HOL is a typed logic whose type system is similar to that of functional programming languages like ML or Haskell. Typed variables are written `'a`, `'b`, etc. The notation `x::'a` means that HOL term `x` is of type `'a`. HOL provides a number of base types, in particular `bool`, the type of truth values and `nat`, the type of natural numbers. Type constructors are supported as well: `nat list` is a list of natural numbers and `nat set` denotes a set of natural numbers. The empty list is written `[]` and the list constructor is written with the infix `x·xs`.

In this paper we are using three ways to introduce new types. The command **datatype** defines a new algebraic data type. For example, the four primitive access rights of our kernel are defined by:

datatype `rights` = `Read` | `Write` | `Grant` | `Create`

For simple abbreviations, we write expressions like **types** `caps = capability set`. Finally, the **record** command introduces tuples with named components. In the example

```
record point =
  x :: nat
  y :: nat
```

the new type `point` is a tuple of two `nat` components. If `p` is a `point`, the term `x p` stands for the `x`-component of `p`, and `y p` for its `y`-component. If `p` has the value `(|x = 2, y = 3|)`, then the update notation `p(|y := 4|)` stands for `(|x = 2, y = 4|)`.

The space of total functions is denoted by $\Rightarrow$. Function update is written `f(x:=y)`. Sequential updates are written `f(x:=a, y:=b)`. Applying a function `f` to a set `A` is written `f ‘ A ≡ {y | ∃x∈A. y = f x}`.

We use $\Longrightarrow$ for implication when we write lemmas to separate antecedent and conclusion. `⟦ A₁; ...; Aₙ ⟧ ⟹ A` abbreviates `A₁ ⟹ (... ⟹ (Aₙ ⟹ A)...)`.

Isabelle proofs can be augmented with LaTeX text. We use this presentation mechanism to generate the text for all of the definitions and theorems in this paper, thus taking them directly from the machine-checked Isabelle sources.

5. The seL4 Protection Model

In the remainder of this paper, we formally analyse the behaviour of the seL4 kernel. Our goal is to show that it is feasible to implement isolated *subsystems* using seL4 mechanisms. An isolated subsystem can be viewed as a collection of processes or *entities* encapsulated in such a way that authority can neither get in nor out. In seL4, this also means that the subsystem cannot gain access to any additional physical memory at any time in the future and is thus strongly spatially separated from the rest of the system.

We start with an example of our requirements, which we carry forward in the discussion below. Assume there are *n* distinct subsystems in our system, namely $ss_1, ss_2 \ldots ss_n$ (see Fig. 4). Each subsystem may contain one or more entities. The resource manager responsible for setting up these subsystems would like to guarantee that any given subsystem, say ss_i, can not exceed the authority explicitly given to it, and with that the amount of physical memory and overt communication channels. In other words, the resource manager would like to guarantee that no entity within ss_i can obtain capabilities to another entity unless these capabilities are already already present in ss_i (possibly in another entity of ss_i).

Given this scenario, the question is: Under which conditions can some entity, say e_x, leak a capability to some other entity e_i at any point in the future? And, importantly, can such a future leak be foreseen and prevented?

Once the subsystems are created, they will execute system calls, and thereby modify the system state. In order

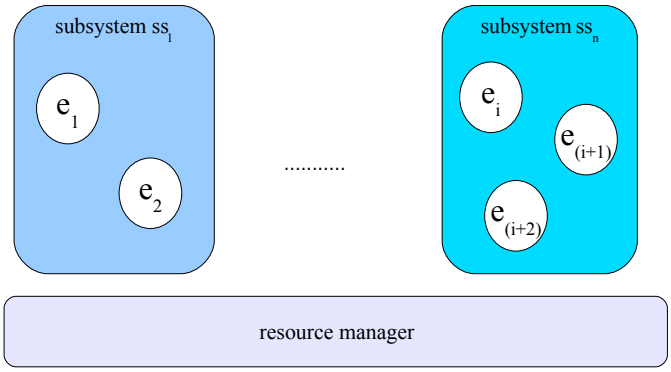


Fig. 4. Isolated Subsystems

to make strong isolation guarantees we need to show that there is no sequence of commands with which to arrive at a future system state in which e_x can leak more access to e_i.

Our interest is in *mandatory* capability confinement, that is, in showing that e_x *cannot* leak a capability to e_i, rather than that e_x *can* but does not.

The structure of our analysis is as follows: the decidability analysis in Sect. 6 shows that it is feasible to decide if such a leak can take place in any future state and identifies what restrictions should be in place to prevent such a leak. Sect. 7 shows how to implement isolated subsystems using seL4 mechanisms.

In the remainder of this section we develop an abstract model of our kernel, sketch how it refines to the concrete seL4 API, and define a number predicates that will be used below.

5.1. Formalisation

The system state consist of a collection of kernel objects. We do not make the usual distinction between active subjects and passive objects. Instead, we collectively call them *entities*. Entities are identified by their memory address which we model as natural numbers. In the usual graph model found in the literature, entities would be nodes and their addresses the names or labels of these nodes.

types `entity_id = nat`

There are four primitive access rights in our model. Each capability has associated with it a set of these rights. Thus we define

datatype `rights = Read | Write | Grant | Create`

`Read` and `Write` have the obvious meaning — they authorise reading and writing of information. Similar to the take-grant model [LS77], `Grant` is sufficient authority to propagate a capability to another entity. The `Create` right models the behaviour of untyped memory objects. It confers the authority to create new entities.

A capability is a record with two fields: a) an identifier which names an entity, and b) a set of rights which defines the operations the entity is authorised to perform on that entity.

record `cap =` `entity :: entity_id`
`rights :: rights set`

On this abstract level, a kernel object (an entity) only contains a set of capabilities. Entities have no additional authority beyond what they possess as capabilities.

record `entity =` `caps :: cap set`

The state of the system consist of two fields:

record `state =` `heap :: entity_id ⇒ entity`
`next_id :: entity_id`

The component `heap` stores the entities of the system. It can be viewed as an array that contains entities at addresses `0` up to and excluding `next_id`. The `next_id` is the next free slot for placing an entity without overlapping with any existing one.[3] This setup allows a simple test to determine the existence of an `entity_id`:

[3] An alternative to this model would be to use a partial function for the heap. We found working with a total function and an explicit, separate domain slightly more convenient in Isabelle.

```
isEntityOf :: state ⇒ entity_id ⇒ bool
isEntityOf s e ≡ e < next_id s
```

This test is not present in the kernel implementation itself. In the implementation, the existence of a capability in the system implies the existence of the entity. The same is true in our abstract model for well-formed states: the entities stored in `heap` contain capabilities, which again contain references to other entities. In any run of the system, these references should only point to existing entities. We call such system states `sane`:

```
sane :: state ⇒ bool
sane s ≡ (∀c∈all_caps s. isEntityOf s (entity c)) ∧ (∀e. ¬ isEntityOf s e ⟶ caps_of s e = ∅)

all_caps :: state ⇒ cap set
all_caps s ≡ ⋃e caps_of s e

caps_of :: state ⇒ entity_id ⇒ cap set
caps_of s sref ≡ caps (heap s sref)
```

where `caps_of s r` denotes the set of all capabilities contained in the entity with address `r` in state `s`, and `all_caps s` denotes all capabilities in the given state `s` — the union of the capabilities over all entities in the system. The create operation will guarantee that for any `sane` state `s`

- the new entity will not overlap with any of the existing ones and
- no capability in the current state will be pointing to the heap location of the new entity.

We already discussed the implications of overlapping objects in Section 2.2. As we will see later, the second property is essential for our safety analysis as well.

Next, we introduce the operations of the seL4 model, captured in data type `sysOPs`:

```
datatype sysOPs = SysNoOP entity_id
                | SysRead entity_id cap
                | SysWrite entity_id cap
                | SysCreate entity_id cap cap
                | SysGrant entity_id cap cap rights set
                | SysRemove entity_id cap cap
                | SysRevoke entity_id cap
```

A traditional abstract system model would usually lack the `NoOp` operation. We have included it in our specification, because we intend to prove a formal refinement relation between the take-grant model presented here and the interface the kernel implementation provides. Some of the operations that exist in the seL4 kernel API will not be observable on this abstract level and thus can only be mapped to `SysNoOP`. An example is sending a non-blocking message to a thread not willing to accept, which will result in a dropped message. These operations do not change the set of capabilities that is available to any entity in the system. In fact, neither `SysRead` nor `SysWrite` change the abstract system state either. We include them in this model, because they have preconditions that are observable on this level and thus might be interesting for later analysis.

The first argument of each operation indicates the entity initiating that operation. The second argument is the capability being invoked. The third argument for `SysCreate` points to the destination entity for the new capability, for `SysGrant` it is the capability that is transported and for `SysRemove` it is the capability that is removed. The fourth argument to `SysGrant` is a mask for the access rights of the transported capability. The `diminish` function reduces access rights according to such a mask.

```
diminish :: cap ⇒ rights set ⇒ cap
diminish c R ≡ c(|rights := rights c ∩ R|)
```

Through the diminish function, the entity initiating `SysGrant` operation is at liberty to transport a subset of the authority it possesses to the receiver.

Any operation is allowed only under certain preconditions, encoded by `legal`.

```
legal :: sysOPs ⇒ state ⇒ bool

legal (SysNoOP e) s          = isEntityOf s e
legal (SysRead e c) s        = isEntityOf s e ∧ c ∈ caps_of s e ∧ Read ∈ rights c
legal (SysWrite e c) s       = isEntityOf s e ∧ c ∈ caps_of s e ∧ Write ∈ rights c
legal (SysCreate e c1 c2) s  = isEntityOf s e ∧ {c1, c2} ⊆ caps_of s e ∧ Grant ∈ rights c2 ∧
                               Create ∈ rights c1
legal (SysGrant e c1 c2 r) s = isEntityOf s e ∧ {c1, c2} ⊆ caps_of s e ∧ Grant ∈ rights c1
legal (SysRemove e c1 c2) s  = isEntityOf s e ∧ c1 ∈ caps_of s e
legal (SysRevoke e c) s      = isEntityOf s e ∧ c ∈ caps_of s e
```

The definition of a legal operation firstly checks if the entity initiating the operation exists in that system state. Secondly, all the capabilities specified in the operation should be in the entity's possession at that state. Finally, the capabilities specified should have at least the appropriate permissions.

We now define how each of the operations mutates the system state, assuming it is started in a legal state. The `SysCreate` and `SysGrant` operations add capabilities to entities.

```
createOperation e c1 c2 s ≡
let nullEntity = (|caps = ∅|);
    newCap = (|entity = next_id s, rights = allRights|);
    newTarget = (|caps = {newCap} ∪ caps_of s (entity c2)|)
in (|heap = (heap s)(next_id s := nullEntity, entity c2 := newTarget),
     next_id = next_id s + 1|)
```

The *SysCreate* operation allocates a new entity in the system heap, creates a new capability to the new entity with full authority, and places this new capability in the entity pointed to by the c_2 capability. The operation consumes resources in terms of creating a new entity in the heap. So, the subject initiating this call is required to provide and invoke an untyped capability c_1. Placing the new capability does not consume additional resources.

```
grantOperation e c1 c2 R s ≡
s(|heap := (heap s)(entity c1 := (|caps = {diminish c2 R} ∪ caps_of s (entity c1)|))|)
```

The *SysGrant* operation, similar to *SysCreate*, adds a capability to the entity pointed to by c_1. However, unlike *SysCreate*, the new capability is a diminished copy of the existing capability c_2.

The *SysRemove* and *SysRevoke* operation remove capabilities from system entities.

```
removeOperation e c1 c2 s ≡
s(|heap := (heap s)(entity c1 := (|caps = caps_of s (entity c1) - {c2}|))|)
```

The *SysRemove* operation removes the specified capability c_2 from the entity denoted by c_1.

```
cdt :: state ⇒ cap ⇒ (cap × cap list) list

removeCaps e (c, cs) s ≡ foldr (removeOperation e c) cs s
revokeOperation e c s ≡ foldr (removeCaps e) (cdt s c) s

foldr f [] a = a
foldr f (x·xs) a = f x (foldr f xs a)
```

The *SysRevoke* operation is used to revoke all authority from a whole subsystem to prepare it for re-use. The seL4 kernel internally tracks in the so-called *capability derivation tree* [EDE07] how capabilities are derived from one another with create and grant operations. We do not model the capability derivation tree explicitly at this level, instead we assume the existence of a function *cdt* that returns for the current system state and the capability to be revoked, a list that describes which capabilities are to be removed from which entities. Given this list, the revoke operation is then just a repeated call of *SysRemove*.

Fig. 5 shows the definitions above as graph rewriting rules.

A single step of execution in the system is summarised by the functions *step'* and *step*:

```
step :: sysOPs ⇒ state ⇒ state
step' (SysNoOP e) s          = s
step' (SysRead e c) s        = s
step' (SysWrite e c) s       = s
step' (SysCreate e c1 c2) s  = createOperation e c1 c2 s
step' (SysGrant e c1 c2 R) s = grantOperation e c1 c2 R s
step' (SysRemove e c1 c2) s  = removeOperation e c1 c2 s
step' (SysRevoke e c) s      = revokeOperation e c s

step :: sysOPs ⇒ state ⇒ state
step cmd s ≡ if legal cmd s then step' cmd s else s
```

The state after a whole system run, i.e., executing a list of system operations, is then just repetition of *step* (note that the list of commands is read from right to left here):

```
execute :: sysOPs list ⇒ state ⇒ state
execute = foldr step
```

In our model, the kernel after bootstrapping starts with one entity, the resource manager, which posses full rights to itself. In the concrete kernel implementation, the initial state is slightly more complex, with the resource manager thread and a number of separate untyped capabilities to cover all available memory. We have summarised these here into one. We write

```
s_i ≡ (|heap = [0 ↦ {allCap 0}], next_id = 1|)
```

for the initial state s_i. The notation *[0 ↦ {allCap 0}]* stands for an empty heap where position *0* is overwritten with an object that has *{allCap 0}* as its capability set. The empty heap is defined as *emptyHeap* ≡ *λx. nullEntity* where *nullEntity* ≡ (|*caps* = ∅|) and *allCap e* ≡ (|*entity* = *e*, *rights* = *allRights*|).

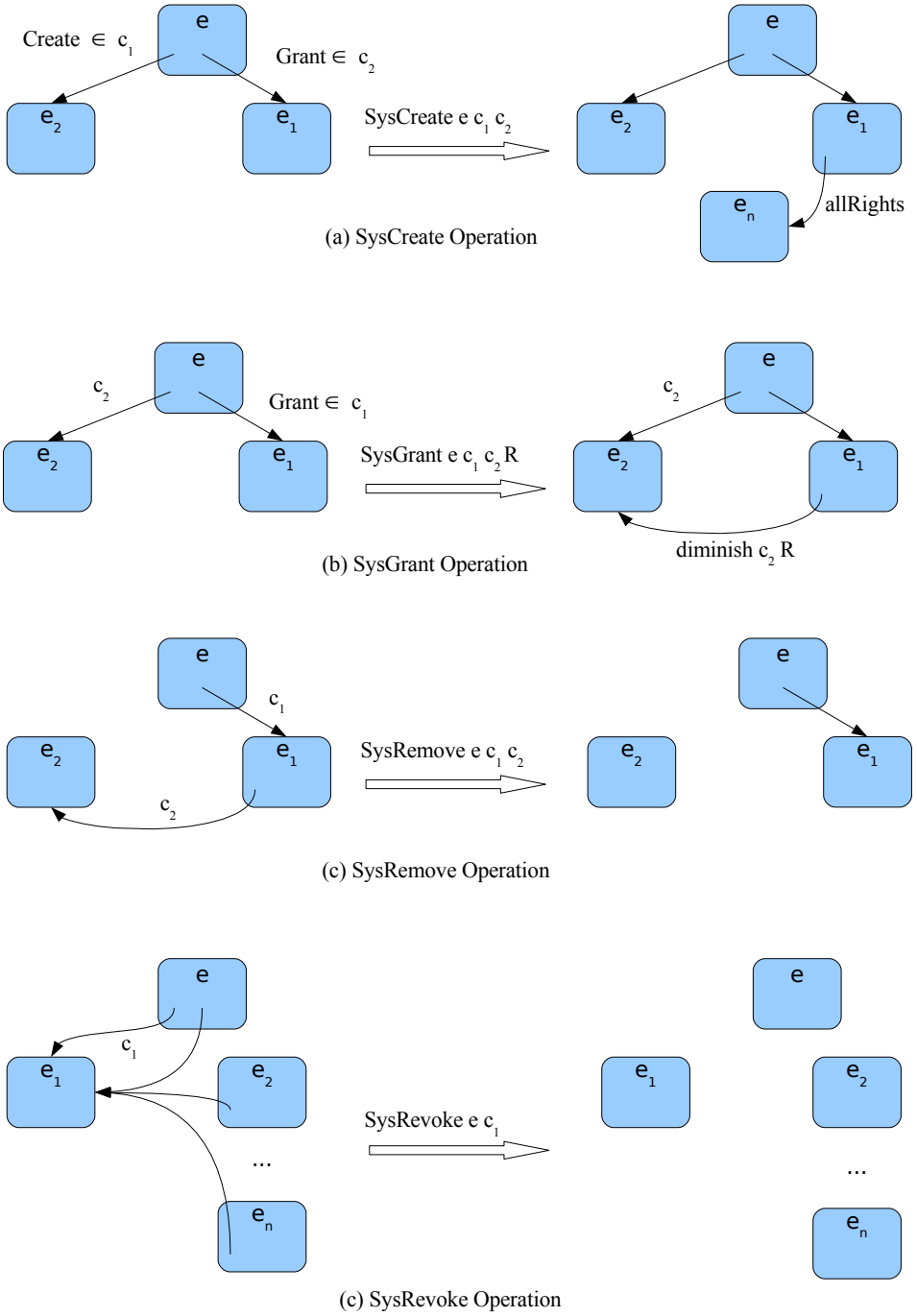


Fig. 5. The seL4 authority distribution rules

5.2. Refining to the seL4 API

The abstract formalism we presented in the last section provides a framework for reasoning about the ability to enforce security policies. One important question, however, is: What does such a policy mean for the concrete kernel? While a full treatment of his topic is beyond the scope of this paper, it is our longer term goal to formally connect the security proofs presented here with the concrete kernel by means of formal refinement between the abstract model we presented above and the deployable kernel implementation. In other work [EKD+07, DEK+06, TKN07] we have developed a precise, abstract formalisation of the seL4 API and have shown how we plan to prove refinement between this API and the implementation.

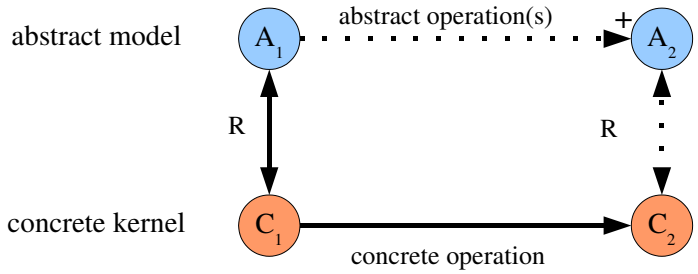


Fig. 6. Refinement

In this section we briefly describe how to connect the abstract take-grant model of Sect. 5.1 to the API formalisation. We have not mechanised this proof in Isabelle yet, but we can give a sketch of the proof outline below.

We show formal refinement [dRE98] by forward simulation. For this, we need to map each concrete kernel API call to a sequence of one or more take-grant model operations, and show that they preserve a suitable refinement relation between abstract and concrete states.

Fig. 6 shows an example. Given a concrete kernel state (represented by C_1 in Fig. 6), and the corresponding abstract state (represented by A_1), we need to find a refinement relation R between them such that it is preserved by execution in the following way. Whenever we perform an operation on the concrete kernel, and thereby modify its state to C_2, we need to show that

- there exist a sequence of operations in the abstract model corresponding to the concrete operation, resulting in state A_2, and
- the relation R holds for the new states C_2 and A_2.

Once this is shown for all concrete operations, we can conclude that all state invariants proven on the abstract model hold for the concrete API (under the image of the refinement relation R). The security properties shown in the sections below are such state invariants.

As we mentioned earlier, the design of the abstract model was influenced by our plan to prove formal refinement. We have already discussed the motivation behind keeping *SysNoOP*. Concrete operations such as *IPCSend* (that is a *send* operation invoked on an communication endpoint), or *IPCReceive* (a *receive* operation on an endpoint) map to *SysWrite* and *SysRead* in the abstract model respectively. Not all concrete operations can be modelled by a single abstract operation. For example, the concrete kernel provides a system call to move capabilities from one CSpace to another. In the abstract model, this corresponds to two operations: a *SysGrant* followed by a *SysRemove*.

The refinement relation R can be sketched as follows. Each concrete kernel object apart for CNode objects maps to an entity in the abstract model. CNode objects store capabilities and are related to thread objects in the concrete kernel by the lookup operation explained in Sect. 2.1. They map to the capability sets of the entities corresponding to these threads. All other entities have empty capability sets. The refinement proof then consists of observing for each concrete API operation that its effect is correctly reduced to the effect on the capability distribution in the abstract model.

5.3. Predicates

Before we proceed to our analysis, a precise statement of what we mean by leak is warranted. Earlier in this section we provided an informal description of leak: if we have two entities, e_x and e_i, we want to prevent e_x from giving (or leaking) a capability to e_i in the current state as well as in any future state that can be reached by executing any sequence of commands.

In our model, there are two operations that add a capability to a system entity — `SysCreate` and `SysGrant`. Furthermore, they are legal only if the entity initiating the operation has a capability that points to the entity under consideration and has `Grant` authority (see definition in Sect. 5.1). Such a capability we call a `grantCap`:

```
grantCap :: entity_id ⇒ cap
grantCap e ≡ (|entity = e, rights = {Grant}|)
```

Note that `grantCap e` is the least authorised capability that allows an addition of a capability to entity `e`. We use the infix operator `:<` to indicate that a set of capabilities has at least as much authority as a given capability:

```
c :< C ≡ ∃c'∈C. entity c = entity c' ∧ rights c ⊆ rights c'
```

If there is a capability in the given set such that it points to the same object and has equal or more authority, then $:<$ returns *true* and *false* otherwise.

Now we can define the predicate *leak*. We write $s \vdash e_x \rightarrow e_i$ to indicate that in state s, entity e_x has the ability to give a capability to entity e_i. Its definition is as follows:

$$leak :: state \Rightarrow entity\_id \Rightarrow entity\_id \Rightarrow bool$$
$$s \vdash e_x \rightarrow e_i \equiv grantCap\ e_i :< caps\_of\ s\ e_x$$

Preventing a leak in the initial state (s_0) is trivial — the resource manager creates this state, and therefore the capabilities possessed by any entity are directly under its control. More interesting are leaks that might occur in some later state:

$$leakInFuture :: state \Rightarrow entity\_id \Rightarrow entity\_id \Rightarrow bool$$
$$\Diamond\ s \vdash e_x \rightarrow e_i \equiv \exists cmds.\ execute\ cmds\ s \vdash e_x \rightarrow e_i$$

That means, there is a sequence of commands, that if executed will result in a state in which entity e_x can leak to e_i. The purpose of the analysis in the next section is twofold. Firstly, we show that given s, it is feasible to decide whether $\Diamond\ s_0 \vdash e_x \rightarrow e_i$ is true. Secondly, we identify a restriction R that the resource manager can enforce on s such that $R\ s \Longrightarrow \neg\ \Diamond\ s \vdash e_x \rightarrow e_i$.

6. Access Control in seL4 is Decidable

In this section, we show that the seL4 access control model is decidable. To be precise, we show that given a state s, and two entities e_x and e_y in s, we can give a tight and safe approximation of the value of the predicate $\Diamond\ s \vdash e_x \rightarrow e_i$. We will show formally that the approximation is safe and argue informally that it is tight. In fact, the literature usually does not call this predicate an approximation, but just makes stronger assumptions on the system such that the approximation is precise and the problem can be called decidable outright. We prefer to leave the model itself more general, and recognise that the decision procedure is in fact a (very good) approximation only.

Using this approximation, if the resource manager adheres to certain restrictions on the initial dissemination of capabilities, a leak can be prevented in any state derived from s.

Note that we are not excluding the ability to create entities. Contrary to the pen-and-paper proofs in the literature, we directly show the property for any sequence of commands, including ones that add new entities to the state. The only restriction is that the entities e_x and e_y already exist in s, otherwise the statement does not make sense in state s. We will discuss later why this does not constitute any loss of generality.

The proof is about 2000 lines of Isabelle scripts. We show here the essential lemmas and, in brief, various engineering techniques we used in the proof. The structure of the proof is as follows. We identify a property, related to *leak*, that is preserved by *step*. We then show that this property naturally lends itself to identifying R — the restriction that will prevent a leak in future.

In the proof we only consider *sane* states: those where no dangling capability pointers exist and where newly created entities are guaranteed not to overlap with existing ones (see Sect. 5.1). This is trivially true for the initial state of the system, and is preserved over execution.

Lemma 1. Single execution steps preserve sanity: $sane\ s \Longrightarrow sane\ (step\ a\ s)$

Proof. By case distinction on the command to be executed, unfolding definitions, and observing that no operation creates overlapping objects or references to non-existing objects. □

Lemma 2. Execution preserves sanity: $sane\ s \Longrightarrow sane\ (execute\ cmds\ s)$

Proof. By induction over the command sequence and Lemma 1. □

The main invariant property of the system relating to authority is the symmetric, reflexive, transitive closure over the *leak* relation — the grant arcs in the capability graph. Occasionally, the symmetric closure alone is useful. We call it *connected* and write $s \vdash e_x \leftrightarrow e_y$:

$$s \vdash e_x \leftrightarrow e_y = s \vdash e_x \rightarrow e_y \vee s \vdash e_y \rightarrow e_x$$

The intuition behind this invariant is the following. We are looking at grant capabilities only, because these are the only ones that can disseminate authority. We need the transitive closure, because we are looking at an arbitrary number of execution steps. We need the symmetric closure, because as soon as there is one entity in the transitive closure that has a grant capability to itself, it can use this capability to invert all grant arcs. The reflexive part of the closure is introduced by the create operation. Given the transitive and symmetric part, each entity with a suitable create right can create a transitive grant path to itself via the entity it has created.

There are two assumptions in this closure that make it an approximation: symmetry and reflexivity. Without these, the property is not invariant over the grant and create operations. With them, we might claim that a given system can gain more authority in the future than what it in fact can: a) if there is no transitive self-referential grant capability in the system or b) if there is no create capability in the system. Although it is possible to build such

systems in seL4, and for small, static systems this might even occur in practise, these are very simple to analyse and it is unlikely that the approximation will lead to undue false alarms. For the vast majority — those with the ability to create and to grant to themselves — the invariant and therefore the prediction is precise.

Recall the definition of `leak` (see Sect. 5.3): it is computed based on the authority one entity possesses over the other. Given our states are `sane`, we see that only two existing entities can be `connected`.

Lemma 3. Connected implies existence: $[\![\texttt{sane s; } s \vdash e_x \leftrightarrow e_y]\!] \Longrightarrow \texttt{isEntityOf s } e_x \wedge \texttt{isEntityOf s } e_y$

Proof. By unfolding the definitions, and observing from `leak` that one entity must possess a capability that points to the other. Thus, given the state is `sane`, both must be existing entities in that state. □

The next part of the proof analyses how each of the operations affects the `connected` relation. In particular, we are interested in properties of the form 'if two entities are connected after execution of an operation, they must have already been connected before'.

The `SysNoOP` operation, as the name implies does nothing. Similarly, `SysRead` and `SysWrite` mutate data and have no effect on the capability distribution. Therefore, it is trivial to conclude that these three operations will not effect the `connected` relation:

Lemma 4. Connected is invariant over NoOP: $\texttt{step (SysNoOP e) } s \vdash e_x \leftrightarrow e_y \Longrightarrow s \vdash e_x \leftrightarrow e_y$

Lemma 5. Connected is invariant over Read: $\texttt{step (SysRead e c) } s \vdash e_x \leftrightarrow e_y \Longrightarrow s \vdash e_x \leftrightarrow e_y$

Lemma 6. Connected is invariant over Write: $\texttt{step (SysWrite e c) } s \vdash e_x \leftrightarrow e_y \Longrightarrow s \vdash e_x \leftrightarrow e_y$

Proof. By unfolding the definition of `step`. □

The remaining four operations — `SysCreate`, `SysGrant`, `SysRemove` and `SysRevoke`, modify the capability distribution, and therefore have the potential to modify the predicate. We now examine their behaviour in detail.

Out of these operations, `SysRemove` and `SysRevoke`, remove capabilities: from a single entity in the case of `SysRemove` or from a set of entities in the case of `SysRevoke`. Moreover recall that `leak` and therefore `connected` checks for the existence of a particular capability. As such, neither operation has the potential to connect two entities that are disconnected. This leads to the following two lemmas.

Lemma 7. Connected is invariant over Remove: $\texttt{step (SysRemove e } c_1 \texttt{ } c_2\texttt{) } s \vdash e_x \leftrightarrow e_y \Longrightarrow s \vdash e_x \leftrightarrow e_y$

Proof. By unfolding the semantics of remove and observing that they can not add capabilities to the state. □

Lemma 8. Connected is invariant over Revoke: $\texttt{step (SysRevoke e c) } s \vdash e_x \leftrightarrow e_y \Longrightarrow s \vdash e_x \leftrightarrow e_y$

Proof. By induction over the list of capabilities to remove and Lemma 7. □

The `SysGrant` operation on the other hand, does have the ability to connect two entities that previously had not been connected. However, only under restricted conditions: the grant operation can connect two entities only if they were transitively connected in the state before.

Lemma 9. Grant preserves the transitive, reflexive closure of connections:

$\texttt{step (SysGrant e } c_1 \texttt{ } c_2 \texttt{ R) } s \vdash e_x \leftrightarrow e_y \Longrightarrow s \vdash e_x \leftrightarrow^* e_y$

Proof. Suppose $s \vdash e_x \leftrightarrow e_y$, then by definition of the transitive closure, the lemma is true. Thus, the case we need to consider is when $\neg\ s_i \vdash e_x \leftrightarrow e_y$, but $\texttt{step (SysGrant e } c_1 \texttt{ } c_2 \texttt{ R) } s \vdash e_x \leftrightarrow e_y$. For this to happen, either e_x or e_y, in the derived state, must possess a `grantCap` to the other. Let this entity be x and the other y. From the definition of `SysGrant`, we see that $\texttt{entity } c_1 = y$, $\texttt{entity } c_2 = x$ and $\texttt{Grant} \in \texttt{rights } c_2$. Moreover, from `legal` we see $\{c_1, c_2\} \subseteq \texttt{caps\_of s e}$ and $\texttt{Grant} \in \texttt{rights } c_1$. Thus, by definition and symmetry of `connected`, we have $s \vdash x \leftrightarrow e$ and $s \vdash e \leftrightarrow y$. Given these facts, from the definition of transitive and reflexive closure, we can conclude the lemma. □

Unlike the operations we considered thus far, `SysCreate` introduces a complication in that it introduces new entities in to the system. However, if we consider existing entities and make use of the `sane` state property, we see that:

Lemma 10. Create preserves connected on existing entities:

$[\![\texttt{isEntityOf s } e_x\texttt{; isEntityOf s } e_y\texttt{; step (SysCreate e } c_1 \texttt{ } c_2\texttt{) } s \vdash e_x \leftrightarrow e_y]\!]$
$\Longrightarrow s \vdash e_x \leftrightarrow e_y$

Proof. By unfolding the definition of create operations and making use of the fact that there can be no dangling references that might point to the new entity (given the state is `sane`), and that the new entity does not overlap with any of the existing ones. □

From these lemmas we see that if two existing entities become connected after executing a single command, then they should either be connected in the previous state, in the case of `SysNoOP, SysRead, SysWrite, SysRemove, SysRevoke` and `SysCreate`, or connected transitively, in the case of `SysGrant`. By combining the previous lemmas we have:

Lemma 11. Connected after a single command:

$[\![isEntityOf\ s\ e_x;\ isEntityOf\ s\ e_y;\ step\ cmd\ s \vdash e_x \leftrightarrow e_y]\!] \Longrightarrow s \vdash e_x \leftrightarrow^* e_y$

Proof. By case distinction on the command, and using the appropriate lemma from Lemma 4 to Lemma 10. □

The plan for the rest of the proof is as follows: We first lift Lemma 11 to the transitive closure, such that $step\ cmd\ s \vdash e_x \leftrightarrow^* e_y \Longrightarrow s \vdash e_x \leftrightarrow^* e_y$, for any two existing entities e_x and e_y. Then, by induction we get $execute\ cmds\ s \vdash e_x \leftrightarrow^* e_i \Longrightarrow s \vdash e_x \leftrightarrow^* e_i$. Moreover, if e_x can leak to e_i in the derived state $execute\ cmds\ s$, then by definition $execute\ cmds\ s \vdash x \leftrightarrow y$, and hence $execute\ cmds\ s \vdash x \leftrightarrow^* y$. From this we can conclude that if e_x can leak to e_y in some future state derived from s then $s \vdash e_x \leftrightarrow^* e_y$.

It turns out, that the first step, lifting Lemma 11 to the reflexive, transitive closure, is the most interesting one. For this proof we use induction over the reflexive transitive closure. Although, we are considering the $connected$ relationship between existing entities, the proof obligation in the induction step is more general in that it requires us to consider entities that might have being introduced by the current command. Recall that there is only one command that introduces a new entity — $SysCreate$. It turns out that Lemma 10 is not strong enough to get through the induction step, because it requires both entities to exist in the pre-state.

Hence, we break the proof into two parts: we treat $SysCreate$ separately from the other commands. We call the commands that do not introduce new entities *transporters* and start by looking at these in more detail. We proved:

Lemma 12. Transport commands preserve $connected^*$ in sane states:

$[\![step\ cmd\ s \vdash e_x \leftrightarrow^* e_y;\ sane\ s;\ \forall e\ c_1\ c_2.\ cmd \neq SysCreate\ e\ c_1\ c_2]\!]$
$\Longrightarrow s \vdash e_x \leftrightarrow^* e_y$

Proof. Firstly we see that the derived state is $sane$, this follows from Lemma 1. Next, we induct over the transitive and reflexive closure. If two entities, x and y, are $connected$ in the derived state, from Lemma 3 it follows both x and y exists in that state. Given that transporters do not add new entities, x and y are entities in s_i, hence from Lemma 11 we can conclude the lemma. □

The complication that arises with $SysCreate$ is that it introduces new entities. Thus, in the induction over the transitive and reflexive closure, the entity considered in the induction step can be the newly created entity. To get through the induction, we need to strengthen the lemma to answer the following question: Can two entities, say x and y, become transitively connected through the newly introduced entity, and if so what is the relationship between x and y in the previous state?

Lemma 13. Given two entity e_x and e_z in the state after $SysCreate\ e\ c_1\ c_2$, given that e_x exists in the pre-state s, and given that $sane\ s$, we know that $s \vdash e_x \leftrightarrow^* e$ if e_z is the entity just created, or $s \vdash e_x \leftrightarrow^* e_z$ otherwise. In Isabelle:

$[\![step\ (SysCreate\ e\ c_1\ c_2)\ s \vdash e_x \leftrightarrow^* e_z;\ isEntityOf\ s\ e_x;\ sane\ s]\!]$
$\Longrightarrow if\ e_z = next\_id\ s\ then\ s \vdash e_x \leftrightarrow^* e\ else\ s \vdash e_x \leftrightarrow^* e_z$

Proof. We note that the derived state is $sane$, which follows from Lemma 1 proceed by induction over the transitive and reflexive closure. The base case is trivial. The induction step has the following form: given $isEntityOf\ s\ e_x$, $sane\ s$, $step\ (SysCreate\ e\ c_1\ c_2)\ s \vdash e_x \leftrightarrow^* e_y$, $step\ (SysCreate\ e\ c_1\ c_2)\ s \vdash e_y \leftrightarrow e_z$, and the induction hypothesis $if\ e_y = next\_id\ s\ then\ s \vdash e_x \leftrightarrow^* e\ else\ s \vdash e_x \leftrightarrow^* e_y$, we show $e_z = next\_id\ s \longrightarrow s \vdash e_x \leftrightarrow^* e$ and $e_z \neq next\_id\ s \longrightarrow s \vdash e_x \leftrightarrow^* e_z$. We make a case distinction on whether e_y and e_z are existing or newly introduced entities.

If both are existing entities we conclude using Lemma 11. If both are newly added, we have $s \vdash e_x \leftrightarrow^* e$ by assumption.

If e_y is newly created, and e_z is existing, we know from the induction hypothesis that $s \vdash e_x \leftrightarrow^* e$ and need to show $s \vdash e_x \leftrightarrow^* e_z$. This is true, since from the assumption $step\ (SysCreate\ e\ c_1\ c_2)\ s \vdash e_y \leftrightarrow e_z$, the definition of $legal$, and $sane\ s$, we have that $s \vdash e \leftrightarrow e_z$ and therefore with transitivity what we had to show.

The remaining case is the dual. We know that e_y is an existing entity, and hence $s \vdash e_x \leftrightarrow^* e_y$. We also know that e_z is new and therefore need to show $s \vdash e_x \leftrightarrow^* e$. This reduces to showing $s \vdash e_y \leftrightarrow e$ which again follows from the assumption $step\ (SysCreate\ e\ c_1\ c_2)\ s \vdash e_y \leftrightarrow e_z$, e_z being new, the definition of $legal$, and $sane\ s$. This concludes the proof of Lemma 13. □

We combine Lemma 12 and Lemma 13 to prove the following.

Lemma 14. Single execution steps preserve $connected^*$ for existing entities in sane states:

$[\![step\ cmd\ s \vdash e_x \leftrightarrow^* e_z;\ isEntityOf\ s\ e_x;\ isEntityOf\ s\ e_z;\ sane\ s]\!]$
$\Longrightarrow s \vdash e_x \leftrightarrow^* e_z$

Proof. By case distinction on transporter commands and create, and using Lemma 12 and Lemma 13 to prove each case respectively. □

The rest of the proof is easy. By induction, we can immediately conclude:

Lemma 15. Execution preserves $connected^*$ for all entities existing in any sane initial state:

$$[\![sane\ s;\ isEntityOf\ s\ e_x;\ isEntityOf\ s\ e_y;\ execute\ cmds\ s \vdash e_x \leftrightarrow^* e_y]\!]$$
$$\Longrightarrow s \vdash e_x \leftrightarrow^* e_y$$

Proof. By induction on the list of commands and Lemma 14 □

Together with $s_i \vdash e_x \rightarrow e_i \Longrightarrow s_i \vdash e_x \leftrightarrow^* e_i$, we conclude our theorem on the decidability of the seL4 model:

Lemma 16. In a sane state, if one existing entity can leak authority to another entity at any time in the future, then they are connected now:

$$[\![sane\ s;\ isEntityOf\ s\ e_x;\ isEntityOf\ s\ e_y;\ \Diamond\ s \vdash e_x \rightarrow e_y]\!] \Longrightarrow s \vdash e_x \leftrightarrow^* e_y$$

Proof. By Lemma 15 and the definition of $\Diamond\ s \vdash e_x \rightarrow e_y$. □

The decidability (or approximation thereof) of this model is more naturally phrased like in the literature as the contrapositive of Lemma 16. It clearly identifies the restriction that will prevent a leak from e_x to e_y in any future state:

Theorem 1. In any sane state, if two existing entity are not connected, they will never be able to leak authority to each other.

$$[\![sane\ s;\ isEntityOf\ s\ e_x;\ isEntityOf\ s\ e_y;\ \neg\ s \vdash e_x \leftrightarrow^* e_y]\!]$$
$$\Longrightarrow \neg\ \Diamond\ s \vdash e_x \rightarrow e_y$$

Proof. Contrapositive of Lemma 16. □

The $connected^*$ relation can easily and efficiently be computed by the resource manager in the initial state who has full control and knowledge over the initial distribution of grant rights.

The assumptions of Theorem 1 are that the state s is sane and that both entities exist in s. Sanity is not a problem. We have already shown that it is preserved by execution and since the initial state of the system is sane, it is a global system invariant. The restriction to existing entities might be a reason for concern, though. Formally, we can make no useful statement in s over entities that do no exist in s yet. However, intuitively, we would like the non-leakage property to be true as well for all entities that do not exist yet. The theorem implies that e_x can not leak authority to e_y via any of these new entities (create operations were not excluded in the proof), but what about a new entity e, created later, leaking new authority to e_y? The theorem does not make a statement about this (the precondition $isEntityOf\ s\ e$ is false), but we can run the system up until e is freshly created. In this state e exists, has no authority yet, the state is sane, and e_y still exists. The theorem is then applicable and says that e will not be able to leak to e_y if e_y is not transitively connected to e at that point.

This situation is not entirely satisfying. The classic non-leakage property does not fully express our intuition about the system and for new entities might require the resource manager to keep track of the grant capability distribution in the system.

The next section generalises the above theorem to make a more intuitive and direct statement about authority distribution.

7. Mandatory Isolation of Components

In the last section we proved a standard take-grant non-leakage property for authority distribution in seL4, but found that the statement is not fully satisfying with respect to entities that do not yet exist in the state that is analysed.

In this section, we revisit what the intuition of authority confinement should be and show that it is feasible to implement *isolated subsystems* in seL4. A subsystem is merely a set of connected entities and by isolated we mean that none of the entities in the subsystem will gain access to a capability over an entity of another subsystem if that authority is not already present in the subsystem. If the authority is already present, then we show that it cannot be increased. Subsystems can grow over time and therefore the statement also includes entities that currently do not exist yet.

We start with an illustrative example. Given in Fig. 7 is the configuration of a system with two subsystems. We call them subsystem ss_1 and ss_2. The state shown here is the initial configuration s_0 after bootstrapping. In s_0, neither of the two entities e_1 and e_2 has a capability to e_5. This object might be any system resource, such as a communication endpoint or an untyped object that encapsulates a region of memory.

For isolation to hold we need to show that in no future system state any of the entities in ss_1 — which might grow/shrink depending on create/remove operations — will have a capability to e_5.

Formally, we identify a subsystem by any of its entities e_s and define it as the set of entities in the symmetric, reflexive, transitive closure of grant arcs to e_s, or short $connected^*$:

$$subSys :: state \Rightarrow entity\_id \Rightarrow entity\_id\ set$$
$$subSys\ s\ e_s \equiv \{e_i \mid s \vdash e_i \leftrightarrow^* e_s\}$$

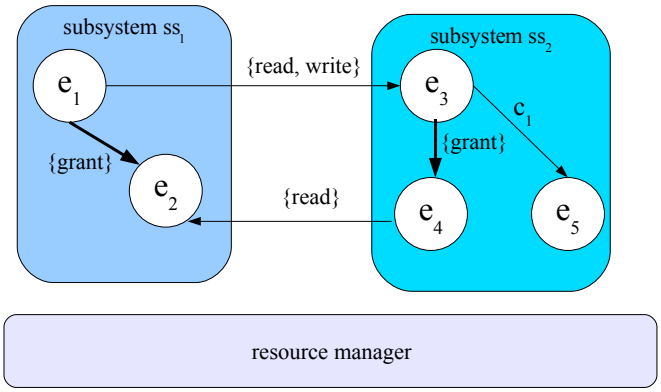


Fig. 7. Example System Configuration

We obtain the entities in a subsystem, using the *subSys* function, specifying the current system state and one of the entities in that subsystem. For instance, the entities of subsystem ss_1 in s_0 are `subSys s0 e1 = {e1, e2}`.

For this proof, we have three main assumptions. Firstly, as we have done before, we assume s is *sane*. Secondly, we assume that the entity we are interested in gaining authority to (or not) exists in s. In the case of our example this is entity e_5. Thirdly, we assume that at least one entity in the subsystem under consideration exists now, otherwise the subsystem would be empty. In the example we have for instance $e_s = e_1$.

Given these, we aim to show that the subsystem of e_s cannot increase authority over e. To formally phrase this statement, we introduce two more concepts — the *subSysCaps* function and the *:>* operator.

```
subSysCaps :: state ⇒ entity_id ⇒ cap set
subSysCaps s x ≡ ⋃caps_of s ‘ subSys s x
```

The function initially finds the set of entities in the subsystem by using *subSys*, and then returns the union of all capabilities possessed by the entities in that subsystem.

The infix operator `c :> C` expresses that the capability set C provides at most as much authority as capability c. That means, if c only posses a grant right to some entity e, then no capability in C will provide more than a grant right to e. It is the dual to *:<*. Formally:

```
c :> C ≡ ∀c’∈C. entity c’ = entity c ⟶ rights c’ ⊆ rights c
```

In our example, we write `noCap e5 :> subSysCaps s0 e1` with `noCap e ≡ (|entity = e, rights = ∅|)` to indicate that there is no e_5 capability anywhere in subsystem ss_1. Similarly, for e_3 we write `rwCap e3 :> subSysCaps s0 e3`, where `rwCap e ≡ (|entity = e, rights = {Read, Write}|)`. That is, the maximum authority within subsystem ss_1 over the entity e_3 is *{Read, Write}*. For isolation to hold we would like to show that:

```
∀cmds. c :> subSysCaps (execute cmds s0) e1
```

where c is some capability to an entity in s. For example c can be `noCap e5`, if we are considering the flow of authority over e_5, or `rwCap e3` if the interest is on e_3.

By using *:>* we can express that authority currently possessed can never grow, as opposed to giving a particular fixed value or restricting ourselves to particular access rights like grant only in Sect. 6. Below, we show such isolation is feasible. Following that, in Sect. 7.2, we provide an example of isolated subsystems together with a brief description of how the resource manager bootstraps them.

7.1. Isolation of Authority

To show isolation, we build upon the results from Sect. 6. It turns out that the main classical take-grant theorem of Sect. 6 is not of much direct use in this proof. However, the central Lemma 14 of Sect. 6 can be used to good advantage. Recall, that for two existing entities in any *sane* state s, they can not become transitively connected after executing a single command, unless they were already transitively connected before:

```
⟦step cmd s ⊢ ex ↔* ey; isEntityOf s ex; isEntityOf s ey; sane s⟧
⟹ s ⊢ ex ↔* ey
```

The main lemma we would like to show in this section is that single step execution does not increase the authority of a subsystem, that is, something of the form:

```
c :> subSysCaps s es ⟹ c :> subSysCaps (step cmd s) es
```

This can then be lifted again to command sequences by induction.

The term `c :> subSysCaps s e_s` can be expressed directly by referring to the entities of subsystem e_s. By definition, we get for any entity that is transitively connected to e_s in s, that there is no capability that contains more authority than c. Formally:

Lemma 17. Unfolding `c :> subSysCaps s e_s`:

```
(c :> subSysCaps s e_s) = (∀ e_x. s ⊢ e_x ↔* e_s ⟶ c :> caps_of s e_x)
```

Proof. By unfolding the definitions of `:>`, `subSysCaps` and `subSys`. □

For isolation to hold we need to show that the above property is true for all states derived from s.

Lemma 18. Single execution steps do not increase subsystem authority:

```
⟦sane s; isEntityOf s e_s; isEntityOf s e; entity c = e; c :> subSysCaps s e_s⟧
⟹ c :> subSysCaps (step cmd s) e_s
```

Proof. We may assume a sane state s, and two entities e_s and e such that e is the entity the capability c points to. Let us consider the situation after executing a single command on s. Let the new state be s'. After unfolding the goal as above, we may additionally assume $s' \vdash e_x \leftrightarrow^* e_s$ for an arbitrary, but fixed e_x and now have to show `c :> caps_of s' e_x`. We proceed by case distinction on whether `c :> caps_of s e_x`, that is if c already dominated all authority of e_x before the command was executed.

- We start with the case `¬ c :> caps_of s e_x`. That means, in s the entity e_x already had a capability with more authority than c.
  We know by assumption that `isEntityOf s e_s`. Moreover, given we are considering `sane` states, e_x also must already exist in s — otherwise it could not have any capability, in particular not one stronger than c. Given both of these are entities in s and we know by assumption that $s' \vdash e_x \leftrightarrow^* e_s$, we get via Lemma 14 that $s \vdash e_x \leftrightarrow^* e_s$. But if that is the case, then e_x was already part of the e_s subsystem in s, and thus the subsystem e_s in s already had more authority than c, which is a contradiction.
- In the second case, we assume `c :> caps_of s e_x` and we still need to show that the execution step did not add authority stronger than c. Here, we proceed by case distinction over the command that was executed. The only interesting cases are `SysGrant` and `SysCreate`.
  If the operation was some `SysGrant e_g c_1 c_2 R`, the capability `diminish c_2 R` is being added to the entity of c_1. If that entity is not e_x, then `caps_of s' e_x = caps_of s e_x` and we are done. If the entity of c_1 happens to be e_x we need to check that `diminish c_2 R` has less authority than c. Via `sane` we know that e_x exists in s (c_1 points to it) and from $s' \vdash e_x \leftrightarrow^* e_s$ we get again via Lemma 14 that $s \vdash e_x \leftrightarrow^* e_s$. From `legal` we know that `{c_1, c_2} ⊆ caps_of s e_g` and `Grant ∈ rights c_1` and therefore $s \vdash e_x \leftrightarrow e_g$. By transitivity, e_g is in the subsystem of e_s, and by assumption `c :> subSysCaps s e_s`. The capability c_2 is part of e_g, hence part of `subSysCaps s e_s` and therefore has less authority than c. The diminished version of c_2 has even less authority and in particular less than c. This concludes the grant case.
  If the operation was some `SysCreate e_c c_1 c_2`, then there are three possibilities: e_x is the entity that was created, it was the target of c_2 that gets the new capability to the entity that was created, or it is neither of these. In the latter case, `caps_of s' e_x = caps_of s e_x` and we are done. In the first case, `caps_of s' e_x = ∅` and since trivially `c :> ∅`, we are done as well. In the remaining case, we add the capability `newCap s` to `caps_of s e_x`. We know that `newCap s` points to `next_id s` which was not an entity in s. On the other hand, we know by assumption that the target of c is an entity in s. Thus, the addition of `newCap s` to `caps_of s e_x` does not increase authority over the target of c. This concludes the create case and the proof.

□

The above lemma is essentially the induction step of the final isolation theorem. In addition, there is one more small lemma that we need. We observe that if e is an entity in s, then it will be an entity in any subsequent state:

Lemma 19. Entities are preserved by execution: `isEntityOf s x ⟹ isEntityOf (execute cmds s) x`

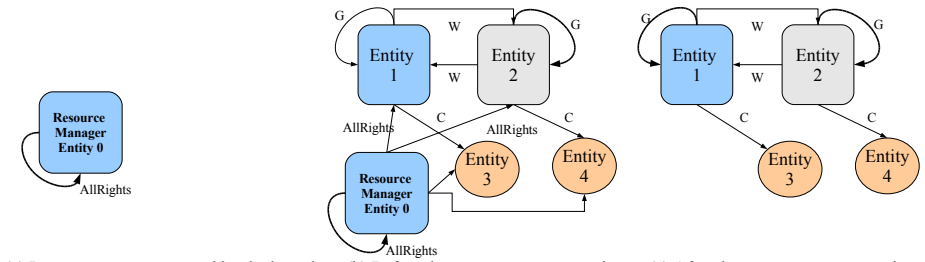
Proof. By induction on `cmds`, then unfolding the definitions and observing that `next_id` never decreases. □

This leads us to the final isolation theorem.

Theorem 2 (Isolation of authority). Given a sane state s, a non-empty subsystem e_s in s, and a capability c with a target identity e in s, if the authority of the subsystem does not exceed c in s, then it will not exceed c in any future state of the system.

```
⟦sane s; isEntityOf s e_s; isEntityOf s e; entity c = e; c :> subSysCaps s e_s⟧
⟹ c :> subSysCaps (execute cmds s) e_s
```

Proof. By induction over the command sequence, and using the lemmas Lemma 18, Lemma 19, and Lemma 2 to prove the induction step. □



(a) Resource manger created by the kernel (b) Before the resource manager exit (c) After the resource manager exits

Fig. 8. Subsystem Configuration

This concludes our isolation proof. The authority that any subsystem collectively has over another entity can not grow beyond what is conferred by the resource manager initially.

Going back to our example in Fig. 7, this means that no entity in subsystem ss_1 will ever have more authority than *{Read, Write}* over entity e_3. Moreover, none of these entities will ever gain any authority at all over entity e_5.

7.2. Implementing Subsystems

In this section we provide a description of how the resource manager bootstraps subsystems, together with an example of a system with two isolated subsystems. Note that there are a number of methods the resource manager can use: given here is one possible method.

Recall that after system startup, the initial state s_i contains only the resource manager with full access rights to itself and with the authority over all the physical memory that is not used by the kernel: $s_i \equiv (\!|heap = [0 \mapsto \{allCap\ 0\}],\ next\_id = 1|\!)$ (see also Sect. 5.1).

It is the responsibility of the resource manager to create and set up the rest of the system.

The current mechanisms used by our resource manager for simple static systems is as follows. For each of the subsystems it creates a *subsystem resource manager* who is responsible for bootstrapping the rest of that particular subsystem. This scheme stems from a major application domain of seL4: running para-virtualised operating systems in each subsystem.

Coupled with the resource manager is a specification language. This language allows the developer to specify the subsystems that should be created together with the authority they should possess over one another and the amount of physical memory that should be committed each manager. Given below is an example written in our specification language:

```
"ss0" {
          text { 1024 to 4096 };
          data { 4096 to 5120 };
          resource { 4 };
          comm { this → ss2 };
};

"ss1" {
          text { 5120 to 6144 };
          resource { 4 };
          comm { this → ss1 };
};
```

To explain the above specification, the system has two subsystems *ss0* and *ss1*. Given this specification the resource manager creates two subsystems by creating two subsystem resource managers. The keywords *text* and *data* specify where to find the text and data segment of the program. The *resource* specifies the amount of physical memory — untyped capabilities, each should have access to. Keyword *comm* specifies the required communication channels. We write *comm {this → ss1}* to say that the current entity should be able to send information to *ss1*.

For *ss0* and *ss1* to be authority confined subsystems, what should be guaranteed is that neither *ss0* nor *ss1* has a capability with *Grant* authority that points to the other. In other words, $\neg\ s \vdash ss0 \leftrightarrow ss1$. This property is guaranteed by our specification language — there is no language construct to specify such a connection. Note that this does not mean that there will be no grant operations in the system at all. The subsystem resource managers are free to provide grant authority within each of the subsystems. We merely exclude the possibility for authority to leak between subsystem partitions. A simple compiler then compiles this specification to a sequence of kernel operations, which is used by the resource manager in the bootstrapping process. While it is feasible to use the specification language directly in the resource manager, we use the compiler to reduce runtime complexity.

Initially, the resource manager creates a pool of entities. Then it populates each entity in accordance with the description produced by the compiler. In addition to what is in the description, each entity gets a capability to itself with *Grant* authority to enable authority distribution within the subsystem itself.

The system state created by the kernel is shown in part (a) of Fig. 8. As mentioned earlier, this state consists of a single entity: the resource manager (entity 0). In the diagram, `allRights` refers to full authority, and `C, G, W` is used to denote `Create`, `Grant` and `Write` rights respectively. At the start, the resource manager possesses a capability with full authority to itself (see part (a) of Fig. 8). The configuration soon after creating and populating each entity with the required authority, according to the above specification is given in part (b) of Fig. 8. We see that in part (b) of the diagram the two entities (entity 1 and 2) are connected through the resource manager which formally means that both entities still inhabit the same single subsystem. The final task of the resource manager is to break these connections. Once all the required capabilities are in place, the resource manager removes its own capabilities to the bootstrapped entities and exits. Thereby, it breaks the connection and makes them isolated subsystems, as shown in part (c) of Fig. 8. The state just after resource manger has exited is the state (s_0).

One possible sequence of commands the resource manager (entity 0) can execute to produce s_0 is given below:

```
cmdSeq ≡
[SysCreate 0 (grantCap 0) (allCap 0), SysCreate 0 (grantCap 0) (allCap 0),
 SysCreate 0 (grantCap 0) (allCap 0), SysCreate 0 (grantCap 0) (allCap 0),
 SysGrant 0 (allCap 1) (allCap 1) {Grant},
 SysGrant 0 (allCap 1) (allCap 2) {Write},
 SysGrant 0 (allCap 1) (allCap 3) {Create},
 SysGrant 0 (allCap 2) (allCap 2) {Grant},
 SysGrant 0 (allCap 2) (allCap 1) {Write},
 SysGrant 0 (allCap 2) (allCap 4) {Create},
 SysRemove 0 (allCap 0) (allCap 1), SysRemove 0 (allCap 0) (allCap 2),
 SysRemove 0 (allCap 0) (allCap 3), SysRemove 0 (allCap 0) (allCap 4)]
```

where `allCap e` ≡ ⦇`entity = e, rights = allRights`⦈ and `grantCap` e_x ≡ ⦇`entity =` e_x`, rights = {Grant}`⦈.

The direct, formal description of the final state created by the resource manager (i.e. the state shown in part (c) of Fig. 8) is given below:

s_0 = ⦇`heap = [1 ↦ {grantCap 1, writeCap 2, utCap 3}, 2 ↦ {grantCap 2, writeCap 1, utCap 4}], next_id = 5`⦈

where `writeCap e` ≡ ⦇`entity = e, rights = {Write}`⦈ and `utCap e` ≡ ⦇`entity = e, rights = {Create}`⦈.

Note that in s_0 all subsystems, including `1` and `2`, have only one entity. There are 5 subsystems in s_0, only the entities `1` and `2` have a `grantCap` to themselves. Each of `1` and `2` possesses the authority to send information to the other, conferred by the corresponding `writeCap` and has access to an untyped capability. They can use this untyped capability to create other entities and bootstrap the remainder of the subsystem.

The two main subsystems thus created cannot increase the authority they have over each other. For example, we can show

Lemma 20. For no sequence of commands can the subsystem `1` gain authority over entity `4` which stands for the physical memory resources of subsystem `2`.

∀ `cmds. noCap 4 :> subSysCaps (execute cmds` s_0`) 1`

Proof. Firstly, we note `sane` s_0. Moreover, by examining s_0 we see that `isEntityOf` s_0 `1` and `isEntityOf` s_0 `4`. Then we observe that `noCap 4 :> subSysCaps` s_0 `1`. Given these facts, we can directly apply Theorem 2 and conclude. □

8. Related Work

The use of abstract formulations of the protection system for safety analysis, i.e. the process of determining whether a particular access can take place, has a long history. Harrison et al. [HRU76], in a model known as *HRU*, showed safety is undecidable in the general case. Since then a number of protection models for which safety is decidable has been proposed — the take-grant model [LS77] and its variations [LM82, Min84], the schematic protection model [San88] and the typed access matrix model [San92a].

In particular the take-grant model [LS77] (TG), as we mentioned before is closely related to our work. The model is generally described in terms of graph theory — each node in the directed graph represents a subject and outgoing, labelled arcs represent the authority possessed by the subject. System operations are modelled as graph rewriting rules. The key similarity between the TG and our model is the distinction between the authority to manipulate data and capabilities. However, the semantics of the create rule in our model differs from that of TG in that the ability to create in our model is explicitly conferred by a capability. This is important to enforce resource usage restrictions in seL4. Furthermore, there is no take rule in our model.

The analysis of de jure (explicitly authorised) access rights in [LS77, BS79] on the TG model states that by constructing a transitive closure on the given initial graph the potential exposure of access rights can be revealed. Our machine checked Theorem 2 affirms this result. In contrast to the pen-and-paper proof that mainly uses graph diagrams for reasoning, we fully formalise the system and make our wellformedness conditions explicit: new nodes added to the graph should not overlap with the existing ones, the graph should not have dangling arcs, and the entity under consideration must exist at the time of analysis. All these restrictions relate to introducing new nodes to the protection graph. If the new node overlaps with an existing one then such a graph mutation will not preserve the initial transitive closure. The same is true if there are dangling references that happen to point to the new node.

Snyder [Sny81] and later Bishop [Bis96] enhanced the TG model by introducing de facto rules — rules that derive feasible information flow paths given the capability distribution. They used the term *island* to denote a maximum take-grant connected subject only subgraph which is similar to our definition of a subsystem. Their analysis identifies conditions under which information can flow from one island to another. The analysis we presented can be extended easily to de facto rights.

Minsky [Min84], in analysing the *send/receive* transport mechanism — a restrictive form of TG, showed that creation of new subjects does not increase the possibility of a leak between two existing subjects as long as all subjects possess the ability to send and receive authority from themselves; such a state he calls *uniform*. Thus, he was able to ignore the create operation from the analysis, assuming the initial state is uniform. The motivation behind removing create operation is to fix the number of subjects in the system. However, our work shows that an indefinite number of subjects does not complicate the analysis — the create operation can be incorporated into the proof with relative ease and therefore the restriction to a uniform initial state is not necessary. It is enough to make the closure over the grant-graph reflexive in the analysis.

The authors in [LM82] showed that by removing the grant rule from TG the flow of authority can be made unidirectional. All capability transfers are authorised by the authority possessed by the receiver rather than the sender, thereby providing endogenous control — the ability to receive a capability is determined solely by the authority within the entity. In a similar manner, the *diminish-take* [Sha99] model proposes filters on the take operation to enforce a transitive read-only path by the authority on the receiver-side. While desirable, endogenous control is not central to our system — the resource manager who enforces the policy has a global view at its disposal. Moreover, there is a natural fit between the grant rule and the operations of the concrete seL4 implementation.

The *schematic protection model* [San88], or *SPM* is closer to HRU [HRU76]. As such, it is more expressive than the TG model and TG can be viewed as an instance of SPM. Moreover, an extension by Ammann and Sandhu [AS90, AS91]; called the *extended* SPM or *ESPM*, yields a model that is formally equivalent to the *monotonic* HRU [HR78] model. Subjects in SPM are associated with a static security type. Each type is allowed to create other types as defined by *can-create* relationship. The model is decidable for *acyclic* creates [San92b]. That is, if subjects of type *a* are allowed to directly or indirectly create subjects of type *b*, then it should not possible for subjects of type *b* to directly or indirectly create subjects of type *a*. The can-create relation is static, in that the types of subjects that can be created by another type, do not change as the system evolves. In seL4 however, the ability to create is dynamic. The distribution of untyped capabilities changes, depending on grant and create operations. Moreover, the static nature of the can-create relation is exploited in the analysis of SPM. All create operations are assumed to occur first. Each subject creates subjects of all possible type, and so do the newly created subjects. Once this state is computed, any subsequent create is redundant, thus the analysis focuses on *copy* (or grant) operations. Note that in [San88] there is another operation called *demand*, which was later shown to be redundant [San89]. Our analysis is much more direct in that we do not need to make any assumptions about the command sequences, and in particular do not need to move all create operations to the beginning, which for seL4 would constitute a loss of generality.

The typed access matrix model [San92a] (TAM) introduces the notion of strong typing to HRU. The monotonic TAM (MTAM) is decidable, but NP-hard. A simplified version of MTAM called, *ternary* MTAM is decidable in polynomial complexity. The techniques used in the analysis of this model [San92a] is similar to that of the SPM.

9. Conclusions

In this paper, we have presented a machine-checked, high-level security analysis of the seL4 microkernel.

We have formalised an access control model of seL4 in the interactive theorem prover Isabelle/HOL. The formalisation is inspired by the classical take-grant model, but without the take rule which does not exist in the seL4 kernel and with a more realistic create rule that is explicitly authorised by a capability. Our formalisation makes the intuitive graph diagram notation that is commonly used for this type of analysis fully precise.

We have shown, in Isabelle/HOL, that our access control model is decidable, or more precisely, that a close approximation thereof is.

We have also shown, in Isabelle/HOL, that the kernel provided mechanisms are sufficient to enforce mandatory isolation between subsystems and that collective authority of subsystems does not increase.

Since all memory is controlled directly by capabilities in seL4, this implies that it is possible to build fully spatially separated systems on top of seL4. The model is general enough to also allow for explicit information flow between partition boundaries via explicit read/write operations. Our main theorem shows that subsystems can neither exceed their authority over physical memory nor their authority over communication channels to other subsystems.

Through an example we have shown that the restrictions required for isolation are pragmatic, and have implemented a resource manager for static systems on seL4 that can bootstrap subsystems from a simple and clear specification language.

In our analysis, we do not look at covert information channels, but at explicit authority distribution only. This might include de facto rights later, but not covert timing channels or physical attacks on the hardware. We are able to guarantee full spatial memory separation and thereby fault isolation, but we believe that with current stock hardware preventing all covert timing channels is not possible.

Future work includes the de facto rights analysis mentioned above, which should be easy to do in our model, and, more importantly, a formal refinement between the model presented here and our work on the abstract seL4 API we have presented elsewhere [DEK+06]. The aim is to make our security analysis apply directly to the full C/assembler implementation of seL4 on the ARM11 platform.

References

[AS90] P. E. Ammann and R. S. Sandhu. Extending the creation operation in the Schematic Protection Model. In *proc. 6th Anual Computer Security Applications*, pages 340–348, Tucson, AZ, USA, Mar 1990. IEEE Comp. Soc.

[AS91] P. E. Ammann and R. S. Sandhu. Safety analysis for the extended schematic protection model. In *proc. IEEE Symp. Research in Security and Privacy*, pages 87–97. IEEE Comp. Soc., May 1991.

[Bis81] M. Bishop. Hierarchical take-grant protection systems. *8th SOSP*, 15(5):109–122, Dec 1981.

[Bis96] Matt Bishop. Conspiracy and information flow in the take-grant protection model. *Journal of Computer Security*, 4(4):331–360, 1996.

[BS79] Matt Bishop and Lawrence Snyder. The transfer of information and authority in a protection system. In *7th SOSP*, pages 45–54, New York, NY, USA, 1979. ACM Press.

[DEK+06] Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *ACM SIGPLAN Haskell WS*, Portland, OR, USA, Sep 2006.

[dRE98] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47. 1998.

[DVH66] J.B. Dennis and E.C. Van Horn. Programming semantics for multiprogrammed computers. *CACM*, 9:143–55, 1966.

[EDE07] Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. A memory allocation model for an embedded microkernel. In *proc. 1st MicroKernels for Embedded Systems*, pages 28–34, Sydney, Australia, Jan 2007. NICTA.

[EKD+07] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *11th HotOS*, San Diego, CA, USA, May 2007.

[EKK06] Kevin Elphinstone, Gerwin Klein, and Rafal Kolanski. Formalising a high-performance microkernel. In Rustan Leino, editor, *Workshop on Verified Software: Theories, Tools, and Experiments (VSTTE 06)*, Microsoft Research Technical Report MSR-TR-2006-117, pages 1–7, Seattle, USA, Aug 2006.

[Har85] Norman Hardy. KeyKOS architecture. *Operat. Syst. Rev.*, 19(4):8–25, Oct 1985.

[HR78] M. Harrison and W. Ruzzo. Monotonic protection systems. In R. DeMillo, D. Dobkin, A. Jones, and R. Lipton, editors, *Foundations of Secure Computation*, pages 337–365. Academic Press, New York, 1978.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *CACM*, pages 561–471, 1976.

[Lie94] Jochen Liedtke. Page table structures for fine-grain virtual memory. *IEEE Technical Committee on Computer Architecture Newsletter*, 1994.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *15th SOSP*, pages 237–250, Copper Mountain, CO, USA, Dec 1995.

[Lie96] Jochen Liedtke. Towards real microkernels. *CACM*, 39(9):70–77, Sep 1996.

[LM82] Abe Lockman and Naftaly H. Minsky. Unidirectional transport of rights and take-grant control. *IEEE Trans. Softw. Engin*, 8(6):597–604, Nov 1982.

[LS77] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *J. ACM*, 24(3):455–464, 1977.

[Min84] Naftaly H. Minsky. Selective and locally controlled transport of privileges. *ACM Trans. Program. Lang. Syst.*, 6(4):573–602, 1984.

[NIC06] National ICT Australia. *seL4 Reference Manual*, 2006. http://www.ertos.nicta.com.au/research/sel4/sel4-refman.pdf.

[NPW02] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NW77] R.M. Needham and R.D.H. Walker. The Cambridge CAP computer and its protection system. In *6th SOSP*, pages 1–10. ACM, Nov 1977.

[San88] Ravinderpal Singh Sandhu. The Schematic Protection Model: Its Definition and Analysis for Acyclic Attenuating Schemes. *Journal of the ACM*, 35(2):404–432, Apr 1988.

[San89] Ravinderpal Singh Sandhu. The demand operation in the schematic protection model. *Information Processing Letters*, 32(4):213–219, Sep 1989.

[San92a] R. S. Sandhu. The typed access matrix model. In *proc. of the IEEE Symp. Security and Privacy*, pages 122–136. IEEE, 1992.

[San92b] Ravi S. Sandhu. Undecidability of safety for the schematic protection model with cyclic creates. *J. Comput. Syst. Sci*, 44(1):141–159, 1992.

[Sha99] Jonathan S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999.

[Sny81] Lawrence Snyder. Theft and conspiracy in the Take-Grant protection model. *Journal of Computer and System Sciences*, 23(3):333–347, Dec 1981.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *17th SOSP*, pages 170–185, Charleston, SC, USA, Dec 1999.

[SW00] Jonathan S. Shapiro and Samuel Weber. Verifying the EROS confinement mechanism. In *IEEE Symp. Security & Privacy*, 2000.

[TKN07] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *34th POPL*, pages 97–108, Nice, France, Jan 2007.