Verified Protection Model of the seL4 Microkernel

Dhammika Elkaduwe, Gerwin Klein and Kevin Elphinstone

NICTA* 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 together with a formal proof that shows how confined subsystems can be enforced. All proofs and specifications in this paper are developed in the interactive theorem prover Isabelle/HOL.

1 Introduction

We present a machine-checked high-level security analysis of seL4 [5, 6], an evolution of the L4 kernel series [12] 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. Even relatively simple devices like mobile phones feature millions of lines of software, installed for various purposes, with varying degrees of assurance, with diverse resource requirements, and developed on a tight resource 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 the end user.

Microkernels are a promising approach with renewed industry interest to improving the security and robustness of such devices. The success of this approach depends to a large degree on the microkernel's ability to provide strong isolation guarantees between components—misbehaviour of a component should be confined to that component only.

In this paper, we analyse the seL4 kernel primitives and affirm that they are sufficient to enforce isolation. Moreover, we demonstrate through examples that the restrictions imposed are pragmatic: The mechanisms can be used in practice.

The main contributions of this work are: (a) to our knowledge, the first machine-checked specification and first machine-checked proof of a take-grant [13] (*TG*) model, (b) making the graph diagram notation that is used for TG analysis in the literature fully precise, (c) extending classical TG in the seL4 protection model by using it to control the in-kernel physical memory consumption of

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

applications, thereby making it feasible to reason about and control the physical memory consumption of applications based on the distribution of authority, and (d) applying the model directly to the seL4 kernel.

All formal definitions, theorems, and examples in this paper are machine-checked in the theorem prover Isabelle/HOL [15].

This paper is one of the steps towards our longer-term goal of full operating systems verification. We aim to formally connect the security proof presented here with the actual kernel implementation in C. All of the seL4 operations map to one or more of the operations presented in this paper. Many of them, e.g., basic thread management, translate to no-op, but the authority-relevant operations have a direct representation in the model presented here. We have so far connected an abstract operational model of seL4 with a precise executable one [3]. We are concurrently working on a formal refinement proof between the security model of this paper and this abstract model and also on the refinement proof between the executable model and the C implementation [7,8]. The security specification is ca. 300loc, the abstract one about 3kloc, the executable one 7kloc, the C code 10kloc, and the refinement proof between abstract and executable specification 100kloc. In this paper, we focus on the tip of this iceberg: the high-level aspects of seL4 security.

2 The seL4 Microkernel

The seL4 (*secure embedded L4*) kernel is an evolution of the L4 microkernel. It employes a capability [4] based protection system that is inspired by early hardware-based capability machines such as CAP where capabilities control access to physical memory, by the KeyKOS and EROS systems [9,19] with their controls on dissemination of capabilities, and by the take-grant model [13]. In this section, we provide an overview of the relevant parts of the seL4 kernel. A detailed exposition can be found elsewhere [5,6,14].

Similar to L4, the seL4 microkernel provides three basic abstractions: threads, address spaces and inter-process communication (IPC). In addition, seL4 introduces the concept of *untyped memory*, which represents a region of currently unused physical memory.

All kernel abstractions and services are provided via named, first-class kernel objects. Authority over these objects is conferred via capabilities. Any seL4 system call is a capability invocation. System call arguments can either be data or other capabilities — authorised users obtain kernel services by invoking capabilities.

A parent capability to untyped memory can be refined into child capabilities to smaller untyped memory blocks or into other kernel objects via the *retype* operation. 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.

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.

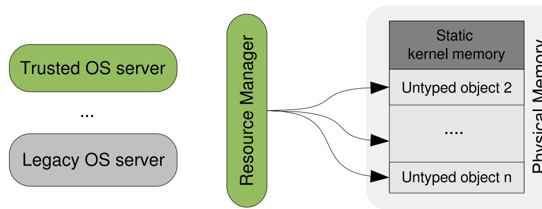

Fig. 1. Sample System Configuration

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. 1 the remainder of memory is divided into untyped memory (UM) objects. The rounded boxes in Fig. 1 stand for threads (OS servers, resource manager), the arrows stand for capabilities, and the box on the right hand side represents available physical memory. The initial user-level thread, the resource manager, has full authority over the UM objects; it is responsible for enforcing a suitable resource management policy, and for bootstrapping the rest of the system. Behaviour of any application, the legacy OS server in Fig. 1 for instance, is constrained by the capabilities the resource manager grants it. Constraining these capabilities appropriately isolates the legacy OS server from the rest of the system.

By this flexible, delegatable, but still precise accounting of all physical memory through capabilities in seL4, the question of partitioning hardware resources becomes a question of capability distribution only. The next sections show how capability distribution is modelled and controlled.

3 The seL4 Protection Model

In the remainder of this paper, we formally analyse the access control model 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. 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. 2). Each subsystem may contain one or more processes and these processes may have access to other resources or processes. In Fig. 2 and the following figures, we use shaded rounded boxes to represent processes and shaded circles to denote resources. We draw arrows between these numbered entities to denote capabilities. The larger rounded boxes mark subsystem boundaries.

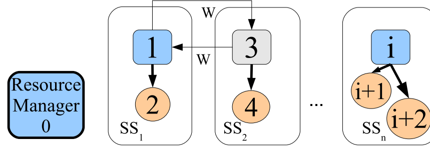

Fig. 2. Isolated Subsystems

The resource manager responsible for setting up these subsystems would like to guarantee that any given subsystem, say ss_i, cannot exceed the authority explicitly given to it, and with that, the amount of physical memory and communication channels[1]. In other words, after providing all subsystems with their initial capabilities, the resource manager would like to guarantee that no entity within ss_i can obtain capabilities to an entity in another subsystem unless these capabilities are already already present in ss_i (possibly in another entity of ss_i). Note that we are not restricting the flow of authority within a subsystem—capabilities can flow freely within a subsystem, but are not allowed to cross the subsystem boundary.

Our interest is in *mandatory* isolation: that is, in showing that a subsystem *cannot* acquire additional capabilities, rather than *it can but does not*.

The initial state in which subsystems start executing (state s_1), is created by and hence under the strict control of the resource manager. Derived states are subsequent states that are affected by the execution of subsystems. To make strong guarantees, we need to show that subsystem boundaries are not violated in any derived state. In the formal analysis below, we identify a set of invariants the resource manager can enforce on s_1, such that there is no sequence of commands that can violate any subsystem boundary.

We begin the analysis by formalising the access control model.

3.1 Formalisation

The system state consists 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 unique address which we model as natural numbers: *entity_id::nat*. An entity contains a set of capabilities which we define below, and has no additional authority beyond what it possesses as capabilities:

record *entity* = *caps :: cap set*

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

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

[1] We do not consider covert timing channels in this paper

where

datatype *rights* = *R* | *W* | *G* | *C*

The data type *rights* defines the four primitive access rights in our model. Out of these rights, *R* and *W* have the obvious meaning. They authorise reading and writing of information. The *G* right is sufficient authority to grant a capability to another entity. The *C* right models the behaviour of untyped memory objects. It confers the authority to create new entities. We use the term *allRights* to denote the set of all the access rights; formally *allRights* ≡ *{R, W, G, C}*.

The state of the whole system consists of two fields:

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

The component *heap* stores the entities of the system, it maps entity addresses (*entity_id*) to entities. The *next_id* is the next free slot for placing an entity without overlapping with any existing one.[2] This setup allows a simple test to determine the existence of an *entity_id*:

is_entity :: state ⇒ entity_id ⇒ bool
is_entity 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 s ≡ (∀*c*∈*all_caps s. is_entity s (entity c)*) ∧
(∀*e.* ¬ *is_entity s e* ⟶ *caps_of s e* = ∅)

where *caps_of s r* is the set of all capabilities contained in the entity at address *r* in state *s*, formally *caps_of s r* ≡ *caps (heap s r)*[3], and *all_caps s* is the union of the capabilities over all entities in the state *s*, formally *all_caps s* ≡ $\bigcup_e$ *caps_of s e*.

Next, we introduce the operational semantics of our model, captured in the function *step'*, shown in Fig. 3. The first argument to *step'* is the operation to perform. The second argument is the current system state and the result is the mutated state after performing the specified operation on the current state.

The first argument of each operation is the entity initiating the operation. The second argument is the capability being invoked. The third argument for *Create* points to the destination entity for the new capability, for *Grant* it is the capability that is transported and for *Remove* the capability that is removed. The fourth argument to *Grant* is a mask for the rights of the transported capability.

[2] 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 this case.

[3] *heap s* selects the field *heap* from record *s*, (|*caps* = ∅|) is the record of type *entity* containing the empty cap set, and *s*(|*heap := h'*|) is the record *s* where field *heap* is replaced by *h'*

```
step' :: sysOPs ⇒ state ⇒ state

step' (Create e c₁ c₂) s  = let newObj = (|caps = ∅|);
                                newCap = (|entity = next_id s, rights = allRights|);
                                newSrc = (|caps = {newCap} ∪ caps_of s (entity c₂)|)
                            in (|heap = (heap s)(next_id s := newObj, entity c₂ := newSrc),
                                 next_id = next_id s + 1|)
step' (Grant e c₁ c₂ r) s = s(|heap := (heap s)
                                (entity c₁ :=
                                   (|caps = {diminish c₂ r} ∪ caps_of s (entity c₁)|))|)
step' (Remove e c₁ c₂) s  = removeOperation e c₁ c₂ s
step' (Revoke e c) s      = foldr (removeCaps e) (cdt s c) s
step' _ s                 = s
```

where

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

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

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

Fig. 3. Single step execution.

There are three additional operations in the model that are matched in the last equation of *step'*: *NoOp*, *Read*, and *Write*.

As the name implies, *NoOp* does nothing, and usually is not present in traditional abstract system models. However, it is included here, because 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 *NoOp*. An example is sending a non-blocking message to a thread not willing to accept, which will result in a dropped message. In fact, neither *Read* nor *Write* 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 *Create* 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 destination entity pointed by c_2. The operation consumes resources in terms of creating the new entity in the heap. So, the subject initiating this call is required to invoke an untyped capability c_1.[4] Placing the new capability does not consume additional resources. Moreover, when performed on a *sane* state, the create operation guarantees: (a) the new entity will not overlap with any of the existing ones and, (b) no capability in the current state will be pointing to the heap location of the new entity.

The *Grant* operation adds a capability to the entity pointed to by c_1. However, unlike *Create*, the added capability is a diminished copy of the existing capability c_2. The *diminish* function reduces access rights according to the mask specified in the *Grant* operation, facilitating an entity to propagate a subset of its own authority to the receiver.

```
diminish cap r ≡ cap(|rights := rights cap ∩ r|)
```

[4] The model presented here does not take into account that memory is limited. For refinement, we introduce non-deterministic failure for operations like *Create* to mimic the real behaviour, but do not specify exactly when memory runs out.

```
legal :: sysOPs ⇒ state ⇒ bool
legal (NoOp e) s          = is_entity s e
legal (Read e c) s        = is_entity s e ∧ c ∈ caps_of s e ∧ R ∈ rights c
legal (Write e c) s       = is_entity s e ∧ c ∈ caps_of s e ∧ W ∈ rights c
legal (Create e c₁ c₂) s  = is_entity s e ∧
                            {c₁, c₂} ⊆ caps_of s e ∧ G ∈ rights c₂ ∧ C ∈ rights c₁
legal (Grant e c₁ c₂ r) s = is_entity s e ∧ {c₁, c₂} ⊆ caps_of s e ∧ G ∈ rights c₁
legal (Remove e c₁ c₂) s  = is_entity s e ∧ c₁ ∈ caps_of s e
legal (Revoke e c) s      = is_entity s e ∧ c ∈ caps_of s e
```

Fig. 4. Preconditions for executing operations.

Both the *Remove* and *Revoke* operations remove capabilities: in the former case from the entity pointed to by c_1 and in the latter case from a whole system. To facilitate *Revoke*, the seL4 kernel internally tracks in a capability derivation tree (*cdt*) [6] how capabilities are derived from one another with create and grant operations. We do not model the cdt 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 *Remove*.

Any operation is allowed only under certain preconditions, encoded by *legal*, as defined in Fig. 4. The definition of *legal* 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.

The single step execution function *step* first checks whether the operation is legal in the current state and if so, calls *step'*.

```
step :: sysOPs ⇒ state ⇒ state
step cmd s = (if legal cmd s then step' cmd s else s)
```

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
```

After the kernel bootstraps itself, it creates state s_0 with one entity; the resource manager, which possesses full rights to itself. In the concrete kernel, the initial state is slightly more complex, containing the resource manager thread and a number of separate untyped capabilities to cover all available memory. We have folded these here into $s_0 \equiv$ (|*heap = [0 ↦ {allCap 0}], next_id = 1*|). 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, where *allCap e* ≡ (|*entity = e, rights = allRights*|). Note that this initial state is sane, and that execution preserves sanity. Formally:

$$\textit{sane s} \Longrightarrow \textit{sane (execute cmds s)}$$

Thus, all states considered in the analysis below are sane.

4 Mandatory Isolation of Components

In this section, we show that it is feasible to implement *isolated subsystems* in seL4. A subsystem is merely a set of entities related to one another in a certain way, which we define later. By isolated we mean that none of the entities in the subsystem ss_1 will gain access to a capability to an entity of another subsystem ss_2 if that authority is not already present in ss_1. If the authority is already present, then we show that it cannot be increased. Subsystems can grow over time and the statement also includes entities that currently do not exist yet.

Our focus is on authority propagation or capability *leakages* from one subsystem to another. We start by considering two entities: e_x and e_y. We write $s \vdash e_x \rightarrow e_y$ to denote that entity e_x has the ability to leak authority to entity e_y in state s.

$leak :: state \Rightarrow entity_id \Rightarrow entity_id \Rightarrow bool$
$s \vdash e_x \rightarrow e_y \equiv gCap\ e_y\ :<\ caps_of\ s\ e_x$
where
$gCap\ e \equiv (\!|entity = e,\ rights = \{G\}|\!)$, and
$c\ :<\ S \equiv \exists c' \in S.\ entity\ c = entity\ c' \wedge rights\ c \subseteq rights\ c'$

where the notation $c\ :<\ S$ reads *the capability set* S *provides as least as much authority as the capability* c. Based on the operational semantics of our model, there are two operations that may create such a leak—*Create* and *Grant*, and these are legal only if the entity initiating the operation has a capability to the entity under consideration with at least grant (G) authority. We therefore define a subsystem as a set of entities connected by grant capabilities. From the analysis in Sect. 4.1 we identify a property that is preserved by *step* and hence by *execute*, which can be used to decide whether an entity will be able to leak to another in the future. Sect. 4.2 then extends this result to show how isolated subsystems can be implemented using seL4. Sect. 4.3 illustrates how isolated subsystems are implement in our abstract model, together with a discussion on how we realised this in the concrete system. The proof is 1200 lines of Isabelle script. We show here only the essential lemmas and their intuition.

4.1 Conditions for Authority Propagation

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

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 grant arcs. Given the transitive and symmetric part, the reflexiveness follows.

Next, we analyse the effect of each operation on the *connected* relation. Obviously, *NoOp*, *Read* and *Write* do not modify the capability distribution and therefore cannot affect *connected*. Similarly, *Remove* and *Revoke* remove capabilities and thus cannot connect two disconnected entities. Moreover, if performed on a sane state, *Create* cannot connect two existing entities: it introduces a new non-overlapping entity and connects that to an existing one. The *Grant* operation, on the other hand has the potential to connect two existing entities, but only under restricted conditions: the grant operation can connect two entities only if they were transitively connected in the state before. Thus, for two existing entities we proved:

Lemma 1. *If two entities of state* s *are connected after an execution step, they must have been transitively connected before. Formally:*
$[\![\mathit{is_entity}\ s\ e_x;\ \mathit{is_entity}\ s\ e_y;\ \mathit{step}\ \mathit{cmd}\ s \vdash e_x \leftrightarrow e_y]\!]$
$\Longrightarrow s \vdash e_x \leftrightarrow^* e_y$

Our plan is to first lift Lemma 1 to the transitive and reflexive closure, such that $\mathit{step}\ \mathit{cmd}\ s \vdash e_x \leftrightarrow^* e_y \Longrightarrow s \vdash e_x \leftrightarrow^* e_y$, for any two existing entities e_x and e_y, by 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 been introduced by the current command. It turns out that Lemma 1 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 *Create* separately from all other commands which we call *transporters*. For transporters, we proved:

Lemma 2. *Transporters preserve* connected* *in sane states:*
$[\![\mathit{step}\ \mathit{cmd}\ s \vdash e_x \leftrightarrow^* e_y;\ \mathit{sane}\ s;\ \forall e\ c_1\ c_2.\ \mathit{cmd} \neq \mathit{Create}\ e\ c_1\ c_2]\!]$
$\Longrightarrow s \vdash e_x \leftrightarrow^* e_y$

For *Create* we proved:

Lemma 3. *Given entities* e_x *and* e_z *in the state after* Create $e\ c_1\ c_2$*, given that* e_x *exists in the pre-state* s*, and given that* sane s*, we know* $s \vdash e_x \leftrightarrow^* e$ *if* e_z *is the entity just created, or* $s \vdash e_x \leftrightarrow^* e_z$ *otherwise. Formally:*
$[\![\mathit{step}\ (\mathit{Create}\ e\ c_1\ c_2)\ s \vdash e_x \leftrightarrow^* e_z;\ \mathit{is_entity}\ s\ e_x;\ \mathit{sane}\ s]\!]$
$\Longrightarrow \mathit{if}\ e_z = \mathit{next_id}\ s\ \mathit{then}\ s \vdash e_x \leftrightarrow^* e\ \mathit{else}\ s \vdash e_x \leftrightarrow^* e_z$

Then, by induction over the command sequence, together with Lemma 2 and Lemma 3 we conclude:

Theorem 1. *If two entities in a sane state* s *are transitively connected after execution, they already have been transitively connected in* s*:*
$[\![\mathit{sane}\ s;\ \mathit{is_entity}\ s\ e_x;\ \mathit{is_entity}\ s\ e_y;\ \mathit{execute}\ \mathit{cmds}\ s \vdash e_x \leftrightarrow^* e_y]\!]$
$\Longrightarrow s \vdash e_x \leftrightarrow^* e_y$

By computing the symmetric, reflexive and transitive closure over *leak* on state s_1—for which there are a number of well known efficient algorithms (for example [16])—we can predict capability leakages that might happen in future states. However, this closure is an approximation because of the symmetry assumption. Without it, the property is not invariant over the grant and create operations. With this assumption, we might claim that a given subsystem can gain more authority in the future than what it in fact can if there is no transitive self-referential grant capability in the system or 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 false alarms. For the majority of systems the invariant and therefore the prediction is precise.

4.2 Subsystems

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 *leak*:

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

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, in our example the entities of subsystem ss_1 in s_1 are $\mathit{subSys}\ s_1\ 1 = \{1, 2\}$ (see Fig. 2).

We aim to show that some subsystem, say ss_1, cannot increase its authority over entity e_i. To formally phrase this statement, we introduce two more concepts: the *subSysCaps* function and the *dominates* (:>) operator.

$$\mathit{subSysCaps}\ s\ x \equiv \bigcup \mathit{caps_of}\ s\ \text{`}\ \mathit{subSys}\ s\ x$$
$$c :> S \equiv \forall c' \in S.\ \mathit{entity}\ c' = \mathit{entity}\ c \longrightarrow \mathit{rights}\ c' \subseteq \mathit{rights}\ c$$

The *subSysCaps* function initially finds the set of entities in the subsystem, and then returns the union of all capabilities possessed by the entities in that subsystem. A capability c dominates a capability set S ($c :> S$) if S provides at most as much authority as capability c over the entity c points to.
For isolation to hold we would like to show:

$$\forall \mathit{cmds}.\ c :> \mathit{subSysCaps}\ (\mathit{execute}\ \mathit{cmds}\ s_1)\ e_1$$

where c is the authority currently possessed by the subsystem over some entity. By using :>, we express that authority currently possessed can never grow, as opposed to giving a particular fixed value or restricting ourselves to a particular access right. For isolation to hold we need to show that the above property is true for all states derived from s_1. For a single step of execution we proved:

Lemma 4. *Single execution steps do not increase subsystem authority:*
$[\![\mathit{sane}\ s;\ \mathit{is_entity}\ s\ e_1;\ \mathit{is_entity}\ s\ (\mathit{entity}\ c);\ c :> \mathit{subSysCaps}\ s\ e_1]\!]$
$\Longrightarrow c :> \mathit{subSysCaps}\ (\mathit{step}\ \mathit{cmd}\ s)\ e_1$

Proof. We begin by noting that no entity that is transitively connected to e_1 possesses a capability with more authority than c, formally c :> subSysCaps s e_s = $\forall e_x.\ s \vdash e_x \leftrightarrow^* e_s \longrightarrow c$:> caps_of s e_x. Moreover, given that the entity pointed to by c already exists and the state is *sane*, the only possibility of obtaining a more authorised capability is by one of the entities within the subsystem receiving a capability via a grant operation. However, for such a grant operation to be *legal* we see that there should be an entity that can *leak* to the subsystem, and hence is *connected* to the entity that received the capability and possesses a more authorised capability than c. This is a contradiction to the assumptions.

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; is_entity s e_s; is_entity s (entity c); c :> subSysCaps s e_s⟧
$\Longrightarrow$ c :> subSysCaps (execute cmds s) e_s

This concludes our isolation proof. The authority that a subsystem collectively has over another entity cannot grow beyond what is conferred by the resource manager initially. Going back to our initial example in Fig. 2, this means that no entity in subsystem ss_1 will ever gain more authority than {W} over entity 3. Moreover, none of these entities will ever gain any authority over entity i.

The statement of Theorem 2 has assumptions about entities existing in state s before execution. Since the authority bound is over whole subsystems and not particular entities, the theorem also covers entities that are created during execution and do not exist in s yet, as long as they belong to a subsystem that is spanned by an existing entity. Since the *Create* operation always connects the new entity to an existing one, we are covered.

What about new subsystems as opposed to new entities, though? The only way to create new subsystems is to remove grant arcs between entities such that two parts of an existing subsystem become disconnected. In the state s' before that removal, Theorem 2 ensures that none of the two candidates have violated their authority bounds. In the state directly after removal, authority has only been removed from the system, so neither subsystem has gained authority. Now we can apply Theorem 2 again. The problem is merely naming the intermediate new subsystems and their parentage.

The main conclusion including new subsystems is: no subsystem can gain more authority than what it already possesses to your resources unless you created it yourself, in which case it cannot exceed what you gave it.

4.3 Implementing Subsystems

We now show how the isolated subsystems described above are bootstrapped and implemented in seL4. Recall that after system startup, the initial state s_0

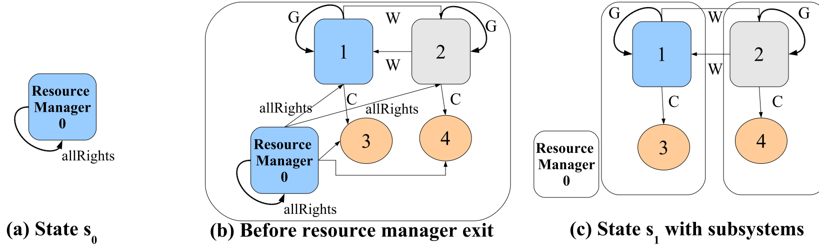

Fig. 5. Subsystem Configuration

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.

For each of the subsystems the resource manager creates a subsystem resource manager who is responsible for bootstrapping the rest of that particular subsystem. This delegation 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 that is used by the developer to specify which subsystems should be created together with what authority they should possess and how much physical memory should be committed to each subsystem. Given below is such a specification:

```
"ss1" { text {1024 to 4096}; data {4096 to 5120 };
        resource {64};       comm {this → "ss2"}; };
"ss2" { text {5120 to 6144}; data {6144 to 10240 };
        resource {64};       comm {this → "ss1"}; };
```

This system would constitute two subsystems: *ss1* and *ss2*, each with authority to send information to the other—specified by *comm {this → ssX}*, and access to 64KB physical, untyped memory (*resource {64}*). The keywords *text* and *data* specify where to find the text and the data segments of each subsystem resource manager respectively.

For *ss1* and *ss2* to be authority isolated, the resource manager should guarantee $\neg\ s \vdash ss1 \leftrightarrow^* ss2$. This is achieved by construction—there is no language construct to specify grant authority between subsystems. Note that the subsystem managers are still free to provide grant authority within the subsystems. We merely exclude the possibility for authority to leak between partitions. A small compiler then translates this specification to a sequence of kernel operations for execution by the resource manager at boot time.

Part (a) of Fig. 5 shows the initial state of our resource manager. The configuration after creating and populating each entity in accordance with the above specification is given in part (b) of Fig. 5. However, in this state, both entity 1 and 2 are still connected through the resource manager, hence formally inhabit the same single subsystem. The final task of the resource manager therefore is to break these grant arcs—once bootstrapped, the resource manager removes its own capabilities to entities and exits, thereby arriving at part (c) of Fig. 5. We call this state s_1, the initial state of the untrusted user mode system.

```
[Create 0 (allCap 0) (allCap 0), Create 0 (allCap 0) (allCap 0),
 Create 0 (allCap 0) (allCap 0), Create 0 (allCap 0) (allCap 0),
 Grant 0 (allCap 1) (allCap 1) {G}, Grant 0 (allCap 1) (allCap 2) {W},
 Grant 0 (allCap 1) (allCap 3) {C}, Grant 0 (allCap 2) (allCap 2) {G},
 Grant 0 (allCap 2) (allCap 1) {W}, Grant 0 (allCap 2) (allCap 4) {C},
 Remove 0 (allCap 0) (allCap 1), Remove 0 (allCap 0) (allCap 2),
 Remove 0 (allCap 0) (allCap 3), Remove 0 (allCap 0) (allCap 4)]
```

Fig. 6. Sequence of kernel operations for bootstrapping

One possible sequence of commands the resource manager (entity 0) can execute to produce s_1 is given in Fig. 6 (for convenience shown left to right). The closed term for the initial state is s_1 = (|*heap* = [1 ↦ {*gCap* 1, *wCap* 2, *cCap* 3}, 2 ↦ {*gCap* 2, *wCap* 1, *cCap* 4}], *next_id* = 5|), where *gCap e*, *wCap e*, and *cCap e* stand for a capability to entity *e* with *G*, *W*, and *C* rights respectively. The term *noCap e* in the lemma below stands for a capability to entity *e* without any access rights.

Strictly speaking, there are 5 subsystems in s_1, each with one entity, but only the entities 1 and 2 contain capabilities. They constitute the main two subsystems. We can now, for example show the following.

Lemma 5. *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_1*) 1*

5 Related Work

Harrison et al. [10] first formulated access control analysis in a model known as *HRU* and focused on the ability of a subject to obtain a particular authority over another in some future state. Our analysis differs from HRU in that we focus on the collective authority possessed by a set of entities.

As mentioned earlier, the take-grant (*TG*) model [13] is closely related to the seL4 protection model. The original analysis of de jure access rights on the TG model [2, 13] already uses the same approximation to model the exposure of access rights: the transitive, symmetric closure on the given initial graph. Our model is different to the classic TG model in that it is aimed at reasoning over the distribution and control of physical resources like memory. We do not use the take-rule in seL4. This has the advantage of giving each subject control over the distribution of its authority at the source. Additionally, to better model the accounting for physical memory, we have added a more complex create rule. Our proof shows that the desirable properties of TG still hold and can be generalised to a statement on full subsystems.

Snyder [20] and later Bishop [1] 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 subgraph which is similar to our concept of subsystem. The analysis we presented can be extended easily to de facto rights.

Shapiro [18] applied the *diminish-take* model—another variant of TG to capture the operational semantics of the *EROS* system.

All formalisations and proofs mentioned in the works above are pen-and-paper only, mostly using graph diagram notation. Our model and proof are formalised and machine-checked in Isabelle/HOL, making the argument fully precise. While we can affirm the previous general results for our model, we did find that graph diagrams can often be deceptively simple, glossing over subtle side conditions that the theorem prover enables us to track precisely. Examples of such side conditions are: new nodes added to the graph cannot overlap with the existing ones (a condition explicitly tested in the kernel), and the graph cannot have dangling arcs. The precise statement of our theorem also makes clear that at each time it covers only existing entities. Although it can be applied inductively for these situations, the issue does not become apparent in earlier formalisations.

Rushby [17] goes further in providing a formulation of isolation that is called non-interference. In this paper we only consider access control and propagation of authority. The non-interference property is stronger: It covers full information flow and timing channels. While such an analysis on the seL4 design would certainly be worthwhile, we do not believe that a strong information flow property can be established by the implementation on current mainstream hardware. Our access control model on the other hand can be.

6 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 and with a more complex, realistic, create rule for achieving precise control of memory allocation. Our formalisation makes the graph diagram notation that is used for this type of analysis in the literature fully precise.

We have shown, in Isabelle/HOL, that seL4 mechanisms are sufficient to enforce mandatory isolation between subsystems and that collective authority of subsystems does not increase. Through an example we have shown that the restrictions required for isolation are pragmatic, and we have successfully implemented a resource manager capable of bootstrapping a paravirtualised Linux kernel [11] on seL4.

Since all memory is controlled directly by capabilities in seL4, isolated subsystems are fully spatially separated from one another. The model is general enough to also allow for explicit information flow across subsystem boundaries via 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.

Future work includes the de facto rights analysis which, as mentioned above, should be easy to add. More importantly future work also includes a formal refinement between the model presented here and our work on the binary compatible seL4 API model [5]. The aim is to make our security analysis apply directly to

the full C and assembler implementation of seL4 on the ARM11 platform. The model presented here is a slightly simplified version of the one that we intend to use for refinement. The main difference in the refinement model is the use of non-determinism to indicate potential failures like running out of memory on object creation.

References

1. M. Bishop. Conspiracy and information flow in the take-grant protection model. *Journal of Computer Security*, 4(4):331–360, 1996.
2. M. Bishop and L. Snyder. The transfer of information and authority in a protection system. In *7th SOSP*, pages 45–54, New York, NY, USA, 1979. ACM Press.
3. D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In C. Munoz and O. Ait, editors, *TPHOLs'08*, LNCS, 2008. To appear.
4. J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9:143–155, 1966.
5. P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *ACM SIGPLAN Haskell WS*, Portland, OR, USA, Sep 2006.
6. D. Elkaduwe, P. Derrin, and K. Elphinstone. A memory allocation model for an embedded microkernel. In *Proc. 1st MicroKernels for Embedded Systems*, pages 28–34, Sydney, Australia, Jan 2007. NICTA.
7. K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *11th HotOS*, San Diego, CA, USA, May 2007.
8. K. Elphinstone, G. Klein, and R. Kolanski. Formalising a high-performance microkernel. In R. 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.
9. N. Hardy. KeyKOS architecture. *ACM Operat. Syst. Rev.*, 19(4):8–25, Oct 1985.
10. M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *CACM*, pages 561–471, 1976.
11. B. Leslie, C. van Schaik, and G. Heiser. Wombat: A portable user-mode Linux for embedded systems. In *6th Linux.Conf.Au*, Canberra, Apr 2005.
12. J. Liedtke. On μ-kernel construction. In *15th SOSP*, pages 237–250, Copper Mountain, CO, USA, Dec 1995.
13. R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *J. ACM*, 24(3):455–464, 1977.
14. National ICT Australia. *seL4 Reference Manual*, 2006. `http://www.ertos.nicta.com.au/research/sel4/sel4-refman.pdf`.
15. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
16. E. Nuutila. *Efficient transitive closure computation in large digraphs*. PhD thesis, Helsinki University of Technology, Jun 1995.
17. J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, Dec 1992. `http://www.csl.sri.com/papers/csl-92-2/`.
18. J. S. Shapiro. The practical application of a decidable access model. Technical Report SRL-2003-04, SRL, Baltimore, MD 21218, Nov 2003.
19. J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *17th SOSP*, pages 170–185, Charleston, SC, USA, Dec 1999.
20. L. Snyder. Theft and conspiracy in the Take-Grant protection model. *Journal of Computer and System Sciences*, 23(3):333–347, Dec 1981.