INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN
Lehr- und Forschungseinheit für theoretische Informatik

Bachelor-Thesis
in Computer Science

The Noninterference property proven for the access control specifation of the seL4 microkernel

Andrea Kuchar

Advisor: Dr Martin Hofmann, Ulrich Schöpp
Submission Date: 03-30-2018

Declaration of authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.

Munich, the 03-30-2018

....

Andrea Kuchar

Abstract

The thesis investigates the question if the specification of the seL4 access control system is strong enough to imply the Noninterference property. Using the verification of the Take-Grant-Protection Model [2] I deduce from it the Unwinding Theorem conditions of the nondeterministic intransitive Noninterference Model [1]. As the specifications and proofs of the take-grant model is developed in the theorem proof assistant Isabelle/HOL I use the same to verify the implication.

List of Figures

Contents

1 Introduction

SeL4 is a high-assurance, high-performance microkernel, primarily developed, maintained and formally verified by NICTA (now Trustworthy Systems Group at Data61) for secure embedded systems. In this thesis, the access control specification in terms of a classical take-grant model is proven to be sound enough to deduce from it the Noninterference property. The classical security property of noninterference assures that there is no unwanted information flow within a system. For the proof of information flow security [1] a variant of intransitive noninterference was applied. D. Elkaduwe, G. Klein and K. Elphinstone present in their paper [2] an abstract specification of the seL4 access control system in the context of a classical take-grant model and a formal proof of its decidability. With this, they showed how confined subsystems can be enforced. The presented security proofs are not yet connected with the actual kernel implementation. For the named noninterference property the authors [1] showed that it is preserved by refinement. So the goal of this thesis is the implication of the noninterference property from the take-grant specification. With this implication it is possible to create a connection with the actual kernel implementation. All proofs and specifications in this thesis are developed in the theorem proof assistant Isabelle/HOL

2 Requirements

2.1 The seL4 Microkernel

The seL4 [6] ist a small operation system kernel. It's based on the in the 1990s developed L4 microkernel and provieds a minimal number of services to applications, such as abstractions for virutal address spaces, threads, inter process comunication (IPC).
Each abstraction ist implemented by an kernel objected with methodes dependent on the abstraction it supplies. The objects can be named and accessed by capablities which are also stored in kernel objects.
Each capability contains an target object and potentially several access rights. By invoking a capability theat points to the kernel object with an corresponding method name, applications can invoke system calls.

2.1.1 System Calls

The kernel provied severel system calls:

- `send()`: The system call argument ist delivered to the target object and the application is allowed to continue. If the target is not able to receive and/or process the arguments immediately, the sending application will be blocked until the arguments can be delivered.
- `NBSend()`: Like `send()`. Exception: If the message is not deliverable it's silently droped.
- `Call()`: Like `send()` but the application is blocked until the object provieds a response, or the receiving application replies.
  If tthe argument is delivered to an application via Endpoint the receiver needs the right to respond to the sender. So in this case an additional capability is added to the arguments.
- `Wait()`: If the target object is not ready `Wait()` is used by an application to block until the object is ready.
- `Reply()`: Used to respond to a `Call()`, using the capability generated by the `Call()` operation.
- `ReplyWait()`: As a combination of `Reply()` and `Wait()` it's efficent for the common case that replying to a request and waiting for the next can be performed in a single system call.

2.1.2 Kernel Objects

The kernel objects can be invoked by applications. The following showes a brief overview of the kernel implemented objects.

- **CNodes**
  Capabilities in seL4 are stored in kernel objects called **CNodes** with a fixed number of slots that can be empty or contain a capability. They have the following operations: `Mint()`, `Copy()`, `Move()`, `Mutate()`, `Rotate()`, `Delete()`, `Revoke()`, `SaveCaller()`, `Recycle()`

- **IPC Endpoints**
  For the *interprocess communication* between threads the kernel supports synchronous (EP) and asynchronous (AsyncEP) endpoints. The capabilities to that endpoints can be limited as send-only or receive-only or be specified to pass capabilities through the endpoint.

- **TCP**
  The *thread control block* object represents a thread of execution in seL4. It needs a CSpace (provides the capabilities required to manipulate the kernel objects) and a VSpace (provides the virtual memory environment required to contain the code and data application). The connections are illustrated in Figure 1.
  The TCB object has the following methods:
  `CopyRegisters()`, `ReadRegisters()`, `WriteRegisters()`, `SetPriority()`, `SetIPCBuffer()`, `SetSpace()`, `Configure()`, `Suspend()`, `Resume()`

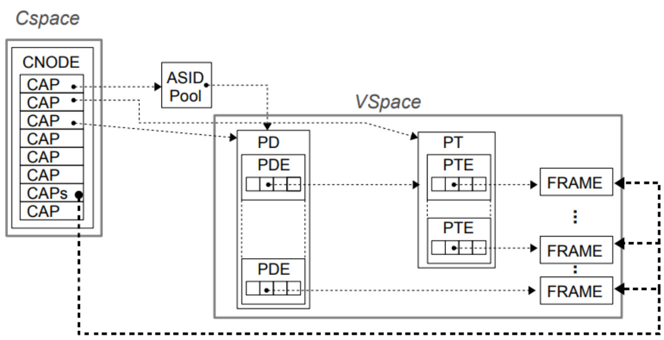

Figure 1: Internal representation of an application in seL4 [3]

- **Virtual Memory**
  A *virtual address space* (VSpace) contains objects for managing virtual memory which largely directly correspond to those of the hardware:
  Page Directory, Page Table, Page, ASID Control, ASID Pool

- **Interrupt Objects**
  For device driver applications to be able to receive and acknowledge interrupts from hardware devices.

- **Untyped Memory**
  Untyped memory objects can be devieded into a group of smaller untyped memory objects. `Retype()` ist the only method untyped memory capabilities have. It creates a number of new kernel objects and returns capabilities to the new objects if it succeeds.

2.1.3 Memory Allocation Model

Important for the seL4 is that all kernel objects must be fully contributed for by capabilities.
At boot time the kernel pre-allocates all the memory required for the kernel to run. This includes the space for kernel code, data and kernel stack. The ressource manager has full authority over the untyped memory (UM) objects, generated by deviding the remain memory into these objects.
A capability to untyped memory can be refined into child capabilities, smaller sized untyped memory blocks or other kernel objects with the retype operation on UM objects.
The creator of an kernel object has full authority over the object. This "full authority" depends on the the object type.
Figure 2 shows a sample system architecture in wich a resource manager running at user-level has the authority to the remaining untyped memory after boot strapping.

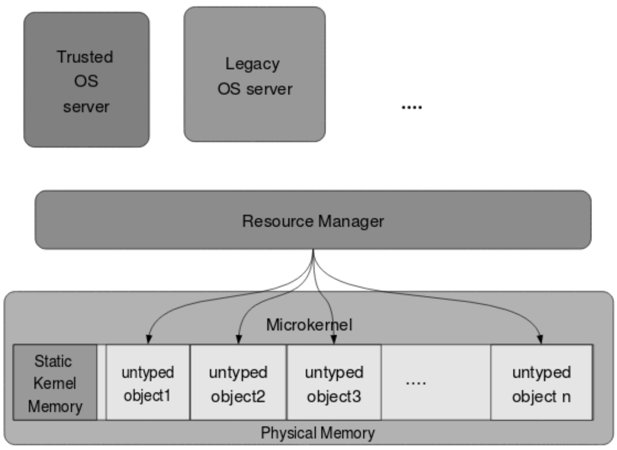

Figure 2: Sample System Configuration [2]

2.2 The Take-Grant Model

Protection or Acces control models specify, analyse and implemente secureity policies. The classical Take-Grant Model primary brought in by Lipton and Snyder, 1977 in "A Linear Time Algorithm for Deciding Subject Security".

2.2.1 The classical Model

The Take-Grant Model [2] represents the system as a directed graph where nodes represent subjects or objects in the system and arcs represent authority.
There are graph mutation rlues that represent the system operations that modify the autority distribution. The most common rules in the classical model are *take, grant, create* and *remove*.

- **take rule**: Let S,X,Y be three distinct vertices in the protection graph with an arc, labelled with α, from X to Y and one labelled with γ from S to X, such that t $\in \gamma$.

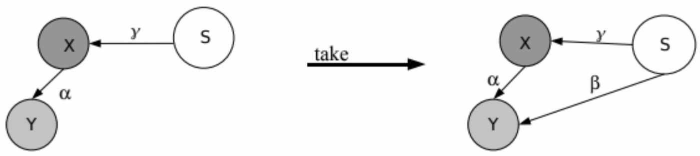

Figure 3: *Take* adds an edge from S to Y with the label $\beta \subseteq \alpha$. [2]

- **grant rule**: Let S,X,Y agein be three distinct vertices in the graph with an arc, labelled with α, from S to Y and one labelled with γ from S to X, such that g $\in \gamma$.

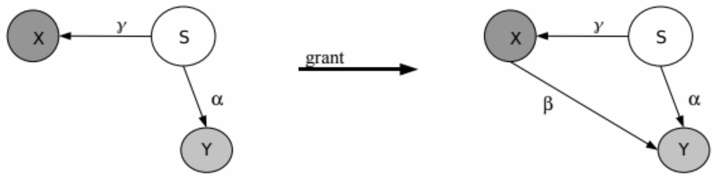

Figure 4: *Grant* adds an edge from X to Y with the label $\beta \subseteq \alpha$. [2]

- **create rule**: Let S be a vertex in the graph.

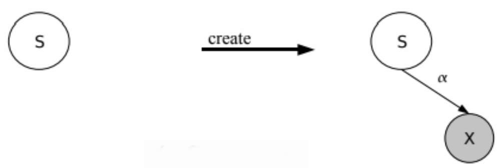

Figure 5: *Create* adds a new node X and an arc from S to X, labelled with α. [2]

- **remove rule**: Let S, X be vertices in the graph with an arc from S to X, labelled with α.

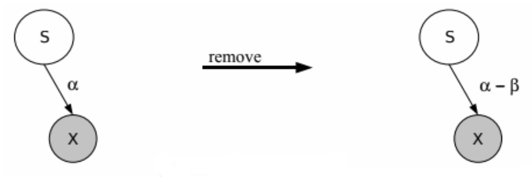

Figure 6: *Remove* deletes β labels from α or the arc itself if $\alpha - \beta = \{\}$. [2]

2.2.2 Take-Grant specified for the seL4

The Take-Grant Model specified in the paper "Noninterference for Operating System Kernels" [2] is a variant of the classical Take-Grant model.
The modification of the *create rule* is the most important one. In the kernel untyped capabilities transfer the authority that has to bei allocated and by the modification adding a new node to the protection graph corresponds to allocation a new object in the concrete kernel. So the only way to apply the create rule is if there is an outgoing arc with *create* authority. The *create* authority is represented by the label *c*.
Also the *remove rule* was modified. It doesn't remove parts of labels. Insted it removes the whole capability, which is the complete arc.
To diminish authority a capability has to be removed and newly created with diminished authority.
The kernel offers an operation called *revoke* wich removes a set of capabilities by mulitple applications of remove.
The goal of the paper "Noninterference for Operating System Kernels" was to show that it is accomblishable to implement isolated subsystems using the mechanisms of the seL4 kernel. [2]
An isolated sybsystem is an collection of connected *entities* enclosed in such a way that authority can neither get in nor out.
The exact specificaton of subsystems and entities follows in Chapter 3.

2.3 Noninterference

Noninterference is an enhancement of the information flow model, first published by Goguen and Meseguer in 1982 and updated in 1984. It ensures that objects and subject from different security levels don't interfere with those at other levels. The used noninterference formultaion for OS kernels [1] expands von Oheimb's notion of *noninfluence* [4].
The system is devided in different *domains*. An information flow *policy* $\rightsquigarrow$ specifies the allowed information flows between the domains: $u \rightsquigarrow v$ if information is allowed to flow from domain u to domain v.

For OS kernels we need an intransitive variant of noninterference, for wich $\rightsquigarrow$ can be intransitive.
The traditional Noninterference formulation was enhanced in in two ways:

1. Traditional formulations presume a static mapping `dom` from actions to domains. In an OS Kernel the mapping does not only depend on the actions but also on the current system state. So in the used formulation of Noninterference [1] `dom` also deppendes on the present state *s*.
   `dom` *a s* equates the domain associated with some action *a* that occurs from state *s*.

2. Due to the fact that the noninterference formulation in "Noninterference for Operating System Kernels" [1] was preserved by refinement, it is necessary to avert all *domain-visible* nondeterminisms.
   Domain-visible nondeterminism is nondeterminism that can be observed by any domain.
   From every confidential source of information which is present in the refinement, such nondeterminisms can be abstracted. From this would result the existence of insecure refinements.
   Lemma 2 [1] determine the restriction of no domain-visible nondeterminisms formally and will be clarified later.

3 Formalisation of the Take-Grant Model

3.1 Capabilities

In the Take-Grant model for seL4 [2] the authors waived the usual differentation betwenn subjects and objects and called all kernel objects *entities*.

The entities memory address identifies them and is modeled as a natural number.

```
type_synonym entity_id = nat
```

With each capability a set of rights is associated. There ar four access rights in the system model:

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

- *Read* authorises the reading of information from another entity.
- *Write* authorises the writing of information to another entity.
- *Grant* authorises the passing of a capability to another entity.
- *Create* authorises the creation of new entities, which models the behavior of untyped memory objects.

A capability has two fields:

1. An identifier which names an target-entity
2. A set of rights which defines which system-operations the source-entity is authorisied to perform on the target-entity.

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

An entity has a set of capabilities:

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

The systems state includes two flields:

1. The `heap`, which stores the entities of the system like an arry form address `0` up to and excluding `next_id`.
2. `next_id` contains slot for next entity without overlapping with an existing one.

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

3.2 System Operations

The system operations of the seL4 are determined in the 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
```

The entity_id in each operation is the entity initiating the operation. The first named capability is the one that is being invoked. The second capability for `SysCreate` points to the target entity for the new capability. For `SysGrant` it's the passed capability and for `SysRemove` it's the one that has to be removed. The rights set in `SysGrant` necessary for the initiating entity to have the option only to transport a subset of the authority it offers to the receiver.
The `diminish` function applies this mask on the given acces rights:

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

`legal` defines on what terms any system operation is allowed.

```
legal ::  "sysOPs ⇒ state ⇒ bool" where

    "legal  (SysNoOp e) s          =  isEntityOf s e"
  | "legal  (SysCreate e c1 c2) s  =  (isEntityOf s e ∧ c1, c2 ⊆ caps_of s e ∧
                                      Grant ∈ rights c2 ∧ Create ∈ rights c2)"
  | "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  (SysGrant e c1 c2 r) s =  (isEntityOf s e ∧ isEntityOf s (entity c1)
                                      ∧ 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"
```

`isEntityOf` tests the existence of an `entity_id`, `caps_of` issues the set of all capabilities contained in the entity with the address `r` in state `s`.

The original executions of `SysRead` and `SysWrite` don't have an underlying function. For implying the noninterference property I have to include what happens if an entity reads or writes a value from another entity. For this purpose I defined a `readOperation` and a `writeOperation`.
The `step'` and `step` functions define the execution of a single system operation:

```
step' ::  "sysOPs ⇒ state ⇒ state" where
     "step'  (SysNoOp e) s           =  s"
  |  "step'  (SysRead e c) s         =  readOperation e c s"
  |  "step'  (SysWrite e c) s        =  writeOperation e c s"
  |  "step'  (SysCreat 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" where
step cmd s ≡ if legal cmd s then step' cmd s else s
```

The new defined functions `readOperation` and `writeOperation`:

```
readOperation ::  "entity_id ⇒ cap ⇒ modify_state" where
"readOperation e c s ≡ s(| heap := (heap s)(e := (|caps = caps_of s e, eValue = value_of
s (entity c)|))|)"

writeOperation ::  "entity_id ⇒ cap ⇒ modify_state" where
"writeOperation e c s ≡ s(| heap := (heap s)(entity c := (|caps = caps_of s (entity c),
eValue = value_of s e|))|)"
```

The rest of the system operation stay as they are:

```
createOperation ::  "entity_id ⇒ cap ⇒ cap ⇒ modify_state" where
createOperation e c1 c2 s ≡
 let   nullEntity = (|cap = , eValue = NULL|) ;
       newCap = (|entity = next_id s, rights = all_rights|);
       newTarget = (|caps = newCap  caps_of s (entity c2), eValue = NULL|)
 in    s(|heap := (heap s)(entity c2 := newTarget, next_id s := nullEntity), next_id := next_id s+1|)"

grantOperation ::  "entity_id ⇒ cap ⇒ cap ⇒ rights set ⇒ modify_state" where
"grantOperation e c1 c2 R s ≡
s(|heap := (heap s)(entity c1 := (|caps = diminish c2 R ∪ caps_of s (entity c1), eValue
= value_of s (entity c1)|))|)"


removeOperation ::  "entity_id ⇒ cap ⇒ cap ⇒ modify_state" where
"removeOperation c1 c2 s ≡ s(|heap := (heap s)(entity c1 := (|caps = caps_of s (entity c1)
- c2, eValue = value_of s (entity c1)|))|)"
```

4 Validation of Confidentiality

First I tried to validate confidentiality for the different system operations as they are defined in the take-grant-model. With this model it's impossible to decide whether a change of value has been recognized by another domain.
In the paper an entity only include a set of capabilites. For my purpose I need the option to access the content of the entities. This ist because the rules for noninterference state that no information is allowed to flow from one domain to another. This includes the information stored in the kernel objects. Therefore I extendet the original record `entity` by adding a *value* modelled by a natural number.

My entity type:

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

After this change it was feasible to deside confidentiality for this model in the following way.

I took one Low-level-Subsystem and one High-level-Subsystem with entities in them and tested for different right-sets and different operations if the confidentiality-property holds. The following shows an example of this approach:

- e_1 $\in$ H, $e_2 \in$ L, $c_1 \in$ s, $c_2 \in$ t
- H equates a High level domain that implements the subsystem 'H'
- L equates a Low level domain that implements the subsystem 'L'

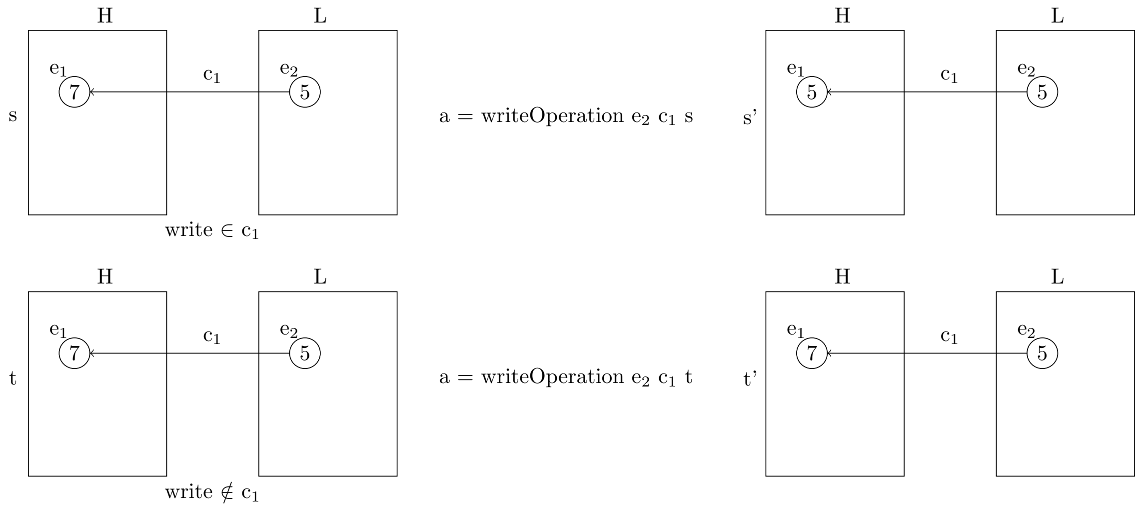

Figure 7: Confidentiality of Write 1

* s $\overset{L}{\sim}$ t $\Rightarrow$ aquiv_nonin s t L
** writeOperation e_2 c_2 t changes e_1 $\in$ H not e $\in$ L
*** writeOperation e_2 c_1 s = s' $\overset{****}{=}$ s
**** legal(SysRead e_2 c_1) s = false

$\forall$ e$\in$L.
value_of s' e $\overset{***}{=}$ value_of s e $\overset{*}{=}$ value_of t e $\overset{**}{=}$ value_of t' e
$\wedge$ caps_of s' e $\overset{***}{=}$ caps_of s e $\overset{*}{=}$ caps_of t e $\overset{**}{=}$ caps_of t' e
$\wedge$ subSys s' e $\overset{***}{=}$ subSys s e $\overset{*}{=}$ subSys t e $\overset{**}{=}$ subSys t' e
$\Rightarrow$ aquiv_nonin s' t' L $\Rightarrow$ s' $\overset{L}{\sim}$ t'

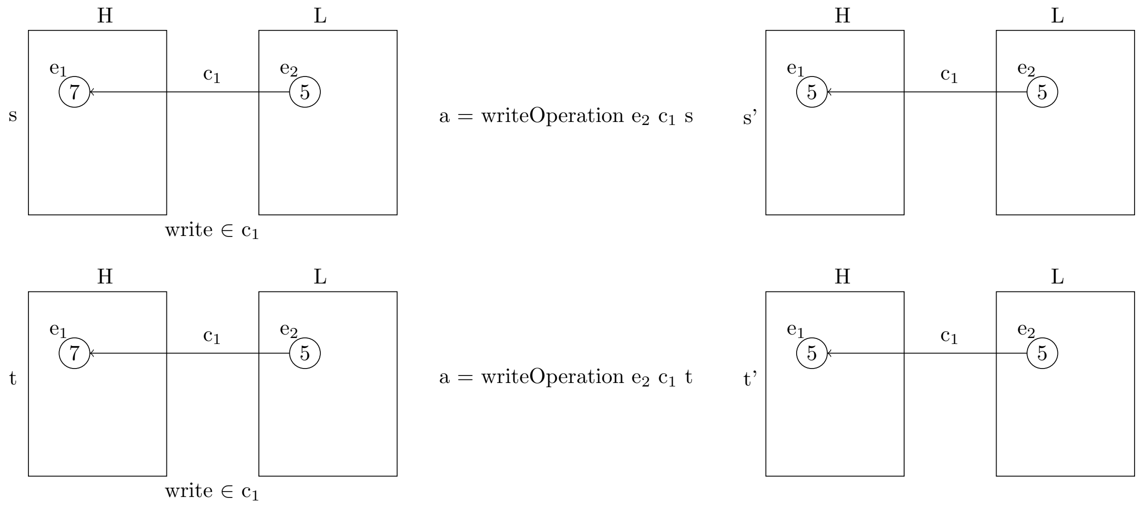

Figure 8: Confidentiality of Write 2

* s $\overset{L}{\sim}$ t $\Rightarrow$ aquiv_nonin s t L
** writeOperation e_2 c_1 s changes e_1 $\in$ H no e $\in$ L
*** writeOperation e_2 c_2 t changes e_1 $\in$ H no e $\in$ L

$\forall$ e$\in$L.

$$\begin{array}{ll} & \text{value\_of s' e} \overset{**}{=} \text{value\_of s e} \overset{*}{=} \text{value\_of t e} \overset{***}{=} \text{value\_of t' e} \\ \wedge & \text{caps\_of s' e} \overset{**}{=} \text{caps\_of s e} \overset{*}{=} \text{caps\_of t e} \overset{***}{=} \text{caps\_of t' e} \\ \wedge & \text{subSys s' e} \overset{**}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{***}{=} \text{subSys t' e} \end{array}$$

$\Rightarrow$ aquiv_nonin s' t' L $\Rightarrow$ s' $\overset{L}{\sim}$ t'

4.1 Redesign of the take-grant-model

This procedure worked until I came to the remove-operation. There I got the problem, that an entity in the given model is allowed to delete a capability an with that also an object in another domain:

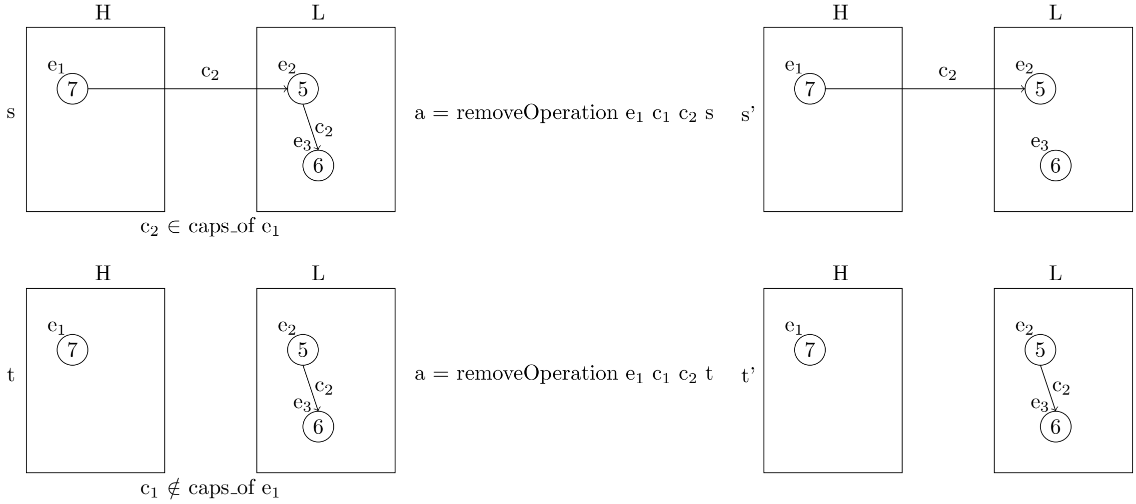

Figure 9: No confidentiality for Remove

To research into this problem I desided to classify the entities by their types,corresponding to the kernel specification er.

4.1.1 Write 3

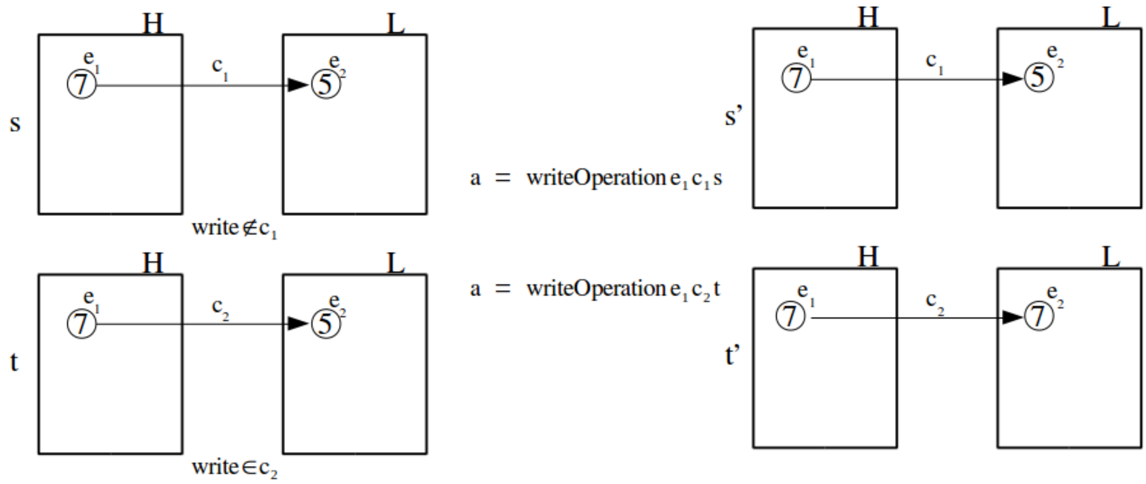

Figure 10: Confidentiality of Write 3

Precondition of confidentiality: s $\overset{L}{\sim}$ t ∧ ... ∧ (H $\rightsquigarrow$ L → s $\overset{H}{\sim}$ t)
In this case: H $\rightsquigarrow$ L but not s $\overset{H}{\sim}$ t because write $\notin c_1 \in$ s but write $\in c_2 \in$ t
⇒ false → true = true

4.1.2 Write 4

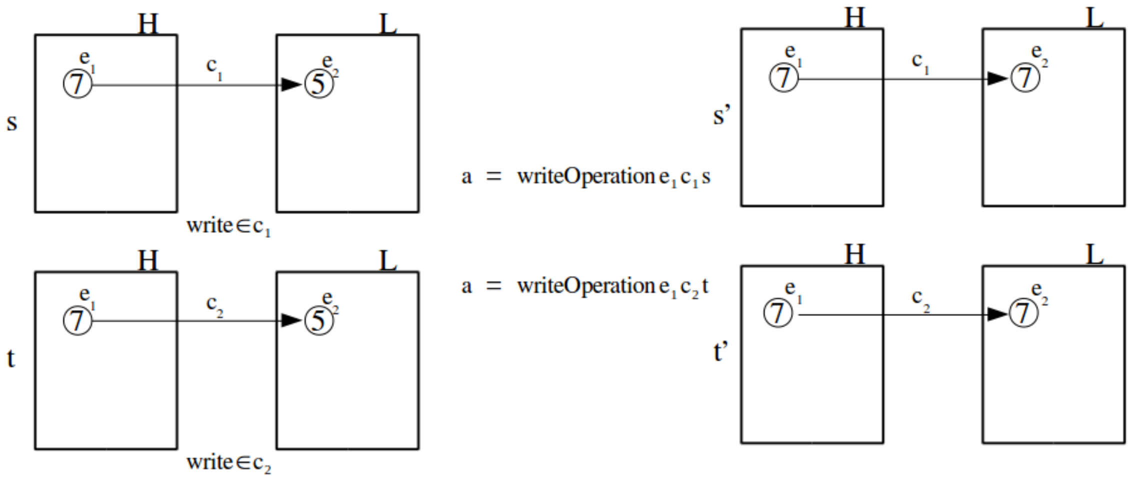

Figure 11: Confidentiality of Write 4

* writeOperation e_1 c_1 s $\stackrel{**}{=}$ writeOperation e_1 c_2 t
** $c_1 = c_2 \wedge$ value_of e_1 s = value_of e_1 t

$\forall$ e$\in$L.
value_of s' e $\stackrel{*}{=}$ value_of t' e
$\wedge$ caps_of s' e $\stackrel{*}{=}$ caps_of t' e
$\wedge$ subSys s' e $\stackrel{*}{=}$ subSys t' e
$\Rightarrow$ aquiv_nonin s' t' L $\Rightarrow$ s' $\stackrel{L}{\sim}$ t'

4.2 readOperation

4.2.1 Read 1

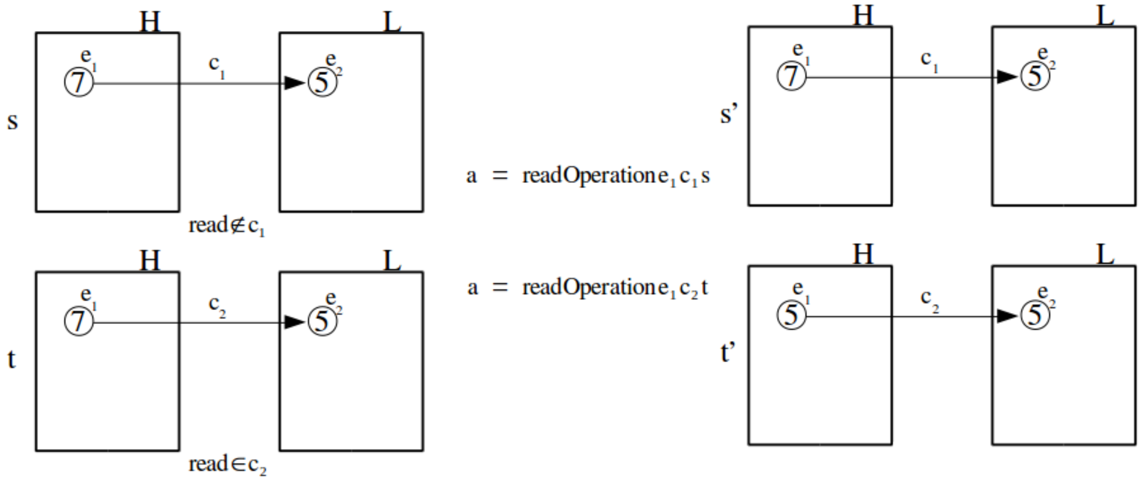

Figure 12: Confidentiality of Read 1

* $s \overset{L}{\sim} t \Rightarrow$ aquiv_nonin s t L
** readOperation e_1 c_2 t changes e_1 $\in$ H no e $\in$ L
*** readOperation e_1 c_1 s = s' $\overset{****}{=}$ s
**** legal(SysRead e_1 c_1) s = false

$\forall$ e$\in$L.

$$\begin{aligned} & \text{value\_of s' e} \overset{***}{=} \text{value\_of s e} \overset{*}{=} \text{value\_of t e} \overset{**}{=} \text{value\_of t' e} \\ \wedge\ & \text{caps\_of s' e} \overset{***}{=} \text{caps\_of s e} \overset{*}{=} \text{caps\_of t e} \overset{**}{=} \text{caps\_of t' e} \\ \wedge\ & \text{subSys s' e} \overset{***}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{**}{=} \text{subSys t' e} \end{aligned}$$

$\Rightarrow$ aquiv_nonin s' t' L $\Rightarrow s' \overset{L}{\sim} t'$

4.2.2 Read 2

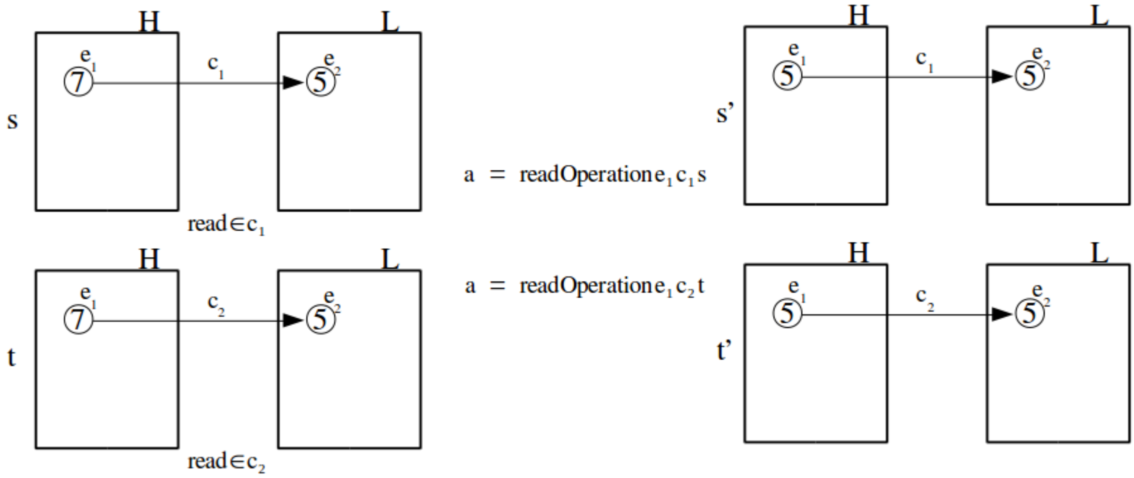

Figure 13: Confidentiality of Read 2

* s $\overset{L}{\sim}$ t ⇒ aquiv_nonin s t L
** readOperation e_1 c_1 s changes e_1 ∈ H no e ∈ L
*** readOperation e_1 c_2 t changes e_1 ∈ H no e ∈ L

∀ e∈L.

$$\text{value\_of s' e} \overset{**}{=} \text{value\_of s e} \overset{*}{=} \text{value\_of t e} \overset{***}{=} \text{value\_of t' e}$$
$$\wedge \quad \text{caps\_of s' e} \overset{**}{=} \text{caps\_of s e} \overset{*}{=} \text{caps\_of t e} \overset{***}{=} \text{caps\_of t' e}$$
$$\wedge \quad \text{subSys s' e} \overset{**}{=} \text{subSys s e} \overset{*}{=} \text{subSys t e} \overset{***}{=} \text{subSys t' e}$$

⇒ aquiv_nonin s' t' L ⇒ s' $\overset{L}{\sim}$ t'

References

[1] T. Murray, D. Matichuk, M. Brassil, P. Gammie and G. Klein:
Noninterference for Operating System Kernels.
International Conference on Certified Programs and Proofs, pp. 126142, Kyoto, Japan, December, 2012

[2] D. Elkaduwe, G. Klein and K. Elphinstone:
Verified Protection Model of the seL4 Microkernel.
Technical Report NRL-1474, NICTA, October, 2007

[3] J. Andronick T. Bourke P. Derrin D. Greenaway D. Elkaduwe, G. Klein and K. Elphinstone R. Kolanski D. Matichuk T. Sewell S. Winwood:
Abstract Formal Specification of the seL4/ARMv6 API.
Version 1.3

[4] D. von Oheimb
Information flow control revisited: Noninfluence = Noninterference + Nonleakage.
In *9th ESORICS*, volume 3193 of *LNCS*, pages 225-243, 2004.

[5] D. Elkaduwe:
A Principled Approach To Kernel Memory Management.
PhD Thesis, UNSW CSE, Sydney, Australia, March, 2010

[6] M. Grosvenor and A. Walker:
seL4 Reference Manual.
Version 10.0.0