Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 254 (2009) 25–44

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Verified Shared Capability Model

Andrew Boyton

Sydney Research Lab., National ICT Australia[1]*, Australia*
School of Computer Science and Engineering, UNSW, Sydney, Australia
andrew.boyton@nicta.com.au

Abstract

This paper presents a high-level access control model of the seL4 microkernel. We extend an earlier formalisation by Elkaduwe et al with non-determinism, explicit sharing of capability storage, and a delete-operation for entities. We formally prove that this new model can enforce system-global security policies as well as authority confinement. By treating sharing explicitly in the abstract access control model we simplify considerably the refinement proof towards the seL4 implementation. To our knowledge this is the first machine-checked access control model with explicit sharing of authority.

Keywords: Shared Capabilities, Interactive Theorem Proving

1 Introduction

This paper presents an extension to the machine-checked, high-level security analysis [8] of seL4. The seL4 kernel [4, 6] is an evolution of the L4 kernel series [11]. The seL4 kernel aims to specifically support *secure, embedded* devices.

The existing formal access control model by Elkaduwe et al [8] is based on the classic take-grant model [12] and also, in parts, takes inspiration from the EROS capability model [16]. The initial focus of the seL4 security model has been to prove security theorems about the model, showing that it is suitable

[1] NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

doi:10.1016/j.entcs.2009.09.058

for enforcing mandatory, system-global access control policies and authority confinement. Isolation properties have been shown in more recent work [5].

The ultimate aim is to show a refinement relation between the model of Elkaduwe et al and an implementation in C of the seL4 kernel. This relation is to be shown in three steps. Our focus is the first: refinement between the security model and an abstract operational specification, which was developed in the L4.verified project [3]. That project aims to show the other two steps: refinement between the operational specification and an executable specification, and from the latter to an implementation in C [10].

Our attempts to prove refinement between the access control model and the operational specification have shown that features of seL4, which are also present in other capability systems, make proof especially complicated. These features are, in increasing order of complexity: non-determinism, deletion of entities (not just capabilities), and sharing of capability storage. The first, non-determinism, is mostly technical and easy to treat. The second, deletion of entities, is conceptually simple, but introduces a mismatch between entities that occur in the security specification and those that exist in the implementation; an example is presented later. The third, sharing of capability storage, introduces a conceptual problem. Moreover, if one naively formalises the classical approach, a great deal of unnecessary complexity into the relation between abstract and concrete states in the refinement proof.

The contribution of this article is to treat all of these features directly in the security model, extending both the existing formalisation and proof of security. In particular,

- we extend the existing model to include non-determinism for an accurate refinement of failure conditions,
- we extend the model to include deletion of entities, which introduces a change to the assumptions of the security theorems, and
- we provide the first formal model and security analysis of shared capability storage. Shared capability storage changes the basic predicates of the analysis and introduces additional possibilities for transmitting authority and information. We show how the analysis can be adjusted to account for shared authority and we prove that the adjusted system is still suitable for enforcing mandatory, system-global access control policies as well as authority confinement and isolation.

All formal definitions and theorems in this paper are machine-checked in the theorem prover Isabelle/HOL [14].

The resulting security model supports a refinement relation that matches entities and capabilities in the security model almost one-to-one with kernel objects in the operational model. This makes the security analysis slightly harder, but it significantly reduces the effort of showing refinement. The latter is a much larger activity than the security analysis.

While the refinement proof is not yet complete, enough progress has been made to expose a security problem in an early version of the operational specification: even though a *grant* operation to transfer authority from a thread A to another thread B is checked explicitly when started, an interruption may occur, and another check is not made upon resuming the operation, even though the authority could have been revoked from A during that interruption. The completion of the operation would therefore be unauthorised and would not refine the security model where all such operations are atomic. The problem has been fixed in the meantime, and shows how refinement can be used to expose subtle defects such as this one.

Although our work was motivated by the desire to conduct a refinement proof for seL4, we believe that the model of shared capability storage is general and interesting on its own. Sharing is a common performance optimisation and likely to be important in resource constrained devices. Instead of sweeping it under the carpet "without loss of generality", we believe that treating it explicitly improves the clarity of a design and the precision of corresponding analyses.

After introducing notation, we proceed by giving a brief overview of the seL4 kernel and stating precisely what is meant by shared capability storage. We examine how the traditional argument for shared capability storage leads to unnecessary complexity, and then present the details of our extended formal access control model and the associated security theorems.

2 Notation

Our meta-language Isabelle/HOL conforms for the most part with normal mathematical notation. This section introduces some exceptions, as well as a few basic data types and primitive operations on them.

The space of total functions is denoted by $\Rightarrow$. Type variables are written $'a$, $'b$, etc. The notation $t :: \tau$ means that HOL term t has HOL type τ.

Sets (type $'a$ set) follow the usual mathematical convention. We write f ‘ A for a function $f :: 'a \Rightarrow 'b$ applied to a set $'a$ set.

The option type

$$\textbf{datatype } 'a \ \mathsf{option} = \mathsf{None} \mid \mathsf{Some}\ 'a$$

adjoins a new element None to a type $'a$. We use $'a$ option to model partial functions. Function update is written $f(x := y)$ where $f :: 'a \Rightarrow 'b$, $x :: 'a$ and $y :: 'b$ and $f(x \mapsto y)$ stands for $f\ (x := \mathsf{Some}\ y)$.

Isabelle supports tuples with named components. For instance, we write **record** *point* $= \{x :: nat,\ y :: nat\}$ for the type *point* with two components of type *nat*. If p is a *point*, a possible value for p is notated $(\!|\ x = 5,\ y = 2\ |\!)$. The term $x\ p$ stands for the x-component of p. Updating p from a current value $(\!|\ x = 5,\ y = 2\ |\!)$, with the update notation $p(\!|\ x{:=}\ 4\ |\!)$, gives $(\!|\ x = 4,\ y = 2\ |\!)$.

The keyword **types** introduces a type abbreviation.

Implication in proof rules and theorems is denoted by $\Longrightarrow$ and $[\![\ A_1; \ldots; A_n\ ]\!] \Longrightarrow A$ abbreviates $A_1 \Longrightarrow (\ldots \Longrightarrow (A_n \Longrightarrow A)\ldots)$. Implication inside object formulae is written $\longrightarrow$. This distinction is a technical artefact of Isabelle/HOL.

3 seL4 and shared capability storage

As mentioned in the introduction, seL4 is a microkernel in the L4 family. It comprises 8,700 lines of C code and provides the following basic services: inter-process communication (IPC), threads, virtual memory, interrupts, and capability-based access control. Access control governs all kernel services; in order to perform any system call, a user process must present capabilities that have sufficient access rights for the requested service. Consider, as two examples, thread communication and creation. Threads do not address one another directly, but rather through communication endpoints maintained within the kernel. Two capabilities are required on an endpoint to send a message through it. The sending thread requires a *write* capability and the receiving thread requires a *read* capability. Creating a thread involves allocating both a thread control block and a new capability to access that block. Thus, for one thread to create another it must possess two distinct capabilities for accessing kernel memory: one with space sufficient to store the thread control block and another with space sufficient to store the new capability.

To support highly dynamic and configurable systems, the kernel allows capabilities to be copied and revoked, and kernel objects, like the endpoints and thread control blocks mentioned above, to be deleted. The required data structures are the most complex part of the seL4 implementation [3]. Copying occurs when threads *grant* capabilities, and hence delegate authority, to other threads via IPC. If a capability is later revoked the kernel must transitively revoke all copies made from it, which simplifies the removal of access to resources from entire subsystems but necessitates careful bookkeeping in the kernel. An object can only be deleted if it cannot be accessed from other parts of the system, since the corresponding memory may be reused subsequently for unrelated purposes.

Capabilities are stored in kernel objects called CNodes. Naturally, access to the CNode objects themselves is governed by capabilities of another type. But since the security model abstracts from the types of capabilities, we instead mark capabilities that provide access to CNode storage with a special *store* right. Each thread control block contains a *store* capability for a CNode, which may then contain *store* capabilities for other CNodes, and so on, giving the associated thread access to a directed graph of capability storage—termed a CSpace. In practice, the CSpace is often a tree or acyclic graph, but no restriction is made in our model. In terms of this paper, the most interesting observation is that a CNode, and thus the other CNodes reachable from it, may be linked into multiple CSpaces: capabilities may be shared between threads.

Capability models must address the issue of sharing if conclusions drawn from them are to be valid. In a traditional capability model, the sharing of CNodes can lead to spooky action at a distance. Consider three threads A, B, and C, where A has a *grant* capability to B, but not to C, and where B and C share all capability storage. Were A to grant a capability to B, it would appear in the storage of B, and, because of the sharing, also in the storage of C—even though A is not authorised to grant to C! Similarly, were a capability deleted from B, it would also disappear from C. A high-level security analysis that neglects the possibility of transfers through the indirect channel of shared storage would be incorrect.

Sharing is traditionally addressed by arguing around it: that B and C share capability storage amounts to grant authority between them in both directions and should be modelled as such. Any action on the capabilities of B must be immediately mirrored on those of C. There is nothing conceptually wrong with this argument, but as sharing structures become more complicated, so too does the relationship between the abstract model and the details of an implementation. The security model must express both normal grant capabilities as well as additional ones to account for sharing. The formalisation of the relationship between it and an implementation model is complex, and proving that it holds is cumbersome. Moreover, there is no longer a simple correspondence between operations executed in the two models: actions in the security model depend on the state of the implementation model.

We contend, in this paper, that it is much easier and more convenient to model sharing explicitly and to account for it directly during security analysis. Further, any security monitor that operates according to the main theorems of the model will have to account for sharing anyway. An explicit model of sharing thus also benefits implementors of security monitors.

4 A Formal Model of Shared Capability Storage

This section presents an extended model of access control in seL4. Where necessary, we repeat definitions from Elkaduwe et al [8]. The extensions are non-determinism, deletion of entities (not just capabilities), and sharing of capability storage.

The model is presented in several parts as we work toward the goal of stating and proving a security theorem about all possible system behaviours. We start by defining capabilities and global state in Sect. 4.1. We then, in Sect. 4.2 define what it means for authority to leak, and, in Sect. 4.3, how the various operations change the state of the model. The main lemmas of the security proof are stated in Sect. 4.4 and generalised to subsystems in Sect. 4.5.

4.1 Capabilities and global state

Following the Elkaduwe model, we do not distinguish between active (e.g. a thread) and passive (e.g. memory) objects, but rather call all kernel objects *entities.* Formally, an entity is just a set of capabilities, which is the only property of interest at this level.

types entity = cap set

Elkaduwe et al [8] model the global state of the security model with a total function and a separate explicit domain. We instead use a partial function from entity-ids to *entities*, to later make the delete command easier to include.

types state = entity-id $\Rightarrow$ entity option

Testing for None suffices to determine whether an entity is part of the state.

is-entity :: state $\Rightarrow$ entity-id $\Rightarrow$ bool
is-entity $s\ e \equiv s\ e \neq$ None

Capabilities are likewise defined as a record with two fields: (a) an identifier which names a target 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}

where **datatype** rights = Read | Write | Grant | Create | Store

The datatype rights defines the five primitive access rights in our model. Read and Write signify the ability to read and write information. Possessing the Create right allows an entity the creation of new entities. An entity with a Grant right to another is able to grant its capabilities to this other entity. The Store right models the concept of shared capabilities. If an entity has a Store rights to another entity, then it has direct access to all the capabilities stored in that entity. Since multiple entities can have a Store capability to a single entity, this allows sharing of capabilities. We use the term all-rights to denote the set of all access rights; formally all-rights = {Read, Write, Grant, Create, Store}.

We say entities possess both direct capabilities (those possessed by the entity itself) and indirect capabilities (those possessed by entities that are *store connected*).

Thus to get the complete set of capabilities of an entity, caps-of s e, we get the direct-caps-of of all of the entities that are store-connected to e, where store-connected is defined as the transitive, reflexive closure of store-connected-direct, as shown below.

$\mathsf{direct\text{-}caps\text{-}of} :: \mathsf{state} \Rightarrow \mathsf{entity\text{-}id} \Rightarrow \mathsf{cap\ set}$
$\mathsf{direct\text{-}caps\text{-}of}\ s\ \mathit{sref} \equiv \mathsf{case}\ s\ \mathit{sref}\ \mathsf{of\ None} \Rightarrow \{\} \mid \mathsf{Some}\ e \Rightarrow e$

$\mathsf{caps\text{-}with\text{-}store} :: \mathsf{state} \Rightarrow \mathsf{entity\text{-}id} \Rightarrow \mathsf{cap\ set}$
$\mathsf{caps\text{-}with\text{-}store}\ s\ e \equiv \{c' \in \mathsf{direct\text{-}caps\text{-}of}\ s\ e.\ \mathsf{Store} \in \mathsf{rights}\ c'\}$

$\mathsf{store\text{-}connected\text{-}direct} :: \mathsf{state} \Rightarrow (\mathsf{entity\text{-}id} \times \mathsf{entity\text{-}id})\ \mathsf{set}$
$\mathsf{store\text{-}connected\text{-}direct}\ s \equiv \{(e_x,\ e_y).\ e_y \in \mathsf{entity}\ `\ \mathsf{caps\text{-}with\text{-}store}\ s\ e_x\}$

$\mathsf{store\text{-}connected} :: \mathsf{state} \Rightarrow (\mathsf{entity\text{-}id} \times \mathsf{entity\text{-}id})\ \mathsf{set}$
$\mathsf{store\text{-}connected}\ s \equiv (\mathsf{store\text{-}connected\text{-}direct}\ s)^*$

$\mathsf{caps\text{-}of} :: \mathsf{state} \Rightarrow \mathsf{entity\text{-}id} \Rightarrow \mathsf{cap\ set}$
$\mathsf{caps\text{-}of}\ s\ e \equiv \bigcup \mathsf{direct\text{-}caps\text{-}of}\ s\ `\ \{e'.\ (e,\ e') \in \mathsf{store\text{-}connected}\ s\}$

Example

To better understand the above definitions, let us consider a small example, where the state s is defined by:

$s\ id_i = \mathsf{Some}\ e_i,\ i = 0, 1, 2$

where the 3 entities are defined by:

$e_0 \equiv \{(\!|\mathsf{entity} = id_1,\ \mathsf{rights} = \{\mathsf{Store}\}|\!)\}$

$e_1 \equiv \{(\!|\mathsf{entity} = id_2,\ \mathsf{rights} = \{\mathsf{Grant}\}|\!)\}$

$e_2 \equiv \{\}$

We can then examine the various capabilities of these entities.

$$\begin{aligned}
&\mathsf{direct\text{-}caps\text{-}of}\ s\ id_0 = \{(\!|\mathsf{entity} = id_1,\ \mathsf{rights} = \{\mathsf{Store}\}|\!)\}\\
&\mathsf{direct\text{-}caps\text{-}of}\ s\ id_1 = \{(\!|\mathsf{entity} = id_2,\ \mathsf{rights} = \{\mathsf{Grant}\}|\!)\}\\
&\mathsf{direct\text{-}caps\text{-}of}\ s\ id_2 = \{\}
\end{aligned}$$

We can then work out which entities are store-connected to each other. Any entity is store-connected to itself by definition, and also to those that are connected via a series of Store rights. Thus,

$$\mathsf{store\text{-}connected}\ s = \{(id_0,\ id_0),\ (id_1,\ id_1),\ (id_2,\ id_2),\ (id_0,\ id_1)\}$$

From this, we can determine the capabilities of each of the entities, which are those directly possessed by the entity themselves and those possessed by store-connected entities.

$$\begin{aligned}
&\mathsf{caps\text{-}of}\ s\ id_0 = \{(\!|\mathsf{entity} = id_1,\ \mathsf{rights} = \{\mathsf{Store}\}|\!),\ (\!|\mathsf{entity} = id_2,\ \mathsf{rights} = \{\mathsf{Grant}\}|\!)\}\\
&\mathsf{caps\text{-}of}\ s\ id_1 = \{(\!|\mathsf{entity} = id_2,\ \mathsf{rights} = \{\mathsf{Grant}\}|\!)\}\\
&\mathsf{caps\text{-}of}\ s\ id_2 = \{\}
\end{aligned}$$

Whilst it is possible to understand this model using the formal definitions, the use of diagrams can greatly assist. Following the notation of Lipton [12], we express the model as a graph, with entities as nodes and capabilities as vertices. Thus we would represent the above state using the following Fig. 1. Note that edge labels, e.g. Store $\in c_1$, means that the right, here Store, is a member of the rights component of c_1, where c_1 is a capability in e_0 whose entity component is e_1.

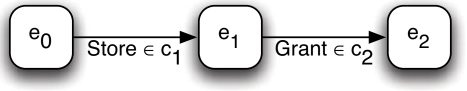


Fig. 1. Diagram representation of the example.

Because we are generally interested in the capabilities (rather than just the direct capabilities) of an entity, we introduce the dashed arrow to represent these capabilities. Intuitively, the dashed arrow represents a series of zero or more Store capabilities, followed by one capability of a certain type as shown in Fig. 2.

4.2 Leak

As in the Elkaduwe model, the central lemma of the security analysis predicts, given the current state, the future distribution of authority. This reduces to the question of whether authority, viz capabilities, can *leak* between entities

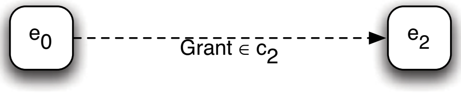


Fig. 2. Alternate diagram representation of the example.

in a state, which in turn enables us to identify partitions of the system where collective authority does not increase.

We now define what it means for a capability to leak from one subsystem to another. The definitions are needed later when we prove that a capability can only be transferred from one entity to another if those entities are already *transitively connected*. We write $s \vdash x \rightarrow y$ to denote that entity x has the ability to leak authority to entity y in state s, which is possible if x has a grant capability to y, if either has store access to the other, or if they both have store access to a common entity.

$\mathsf{leak} :: \mathsf{state} \Rightarrow \mathsf{entity\text{-}id} \Rightarrow \mathsf{entity\text{-}id} \Rightarrow \mathsf{bool}$

$s \vdash x \rightarrow y \equiv \mathsf{grant\text{-}cap}\ y :< \mathsf{caps\text{-}of}\ s\ x \lor \mathsf{shares\text{-}caps}\ s\ x\ y$

where $\mathsf{grant\text{-}cap}\ x \equiv (\!|\mathsf{entity} = x,\ \mathsf{rights} = \{\mathsf{Grant}\}|\!)$
$\mathsf{shares\text{-}caps}\ s\ x\ y \equiv \exists\, e_i.\ (x,\ e_i) \in \mathsf{store\text{-}connected}\ s \land (y,\ e_i) \in \mathsf{store\text{-}connected}$
s
and $c :< C \equiv \exists\, c' \in C.\ \mathsf{entity}\ c = \mathsf{entity}\ c' \land \mathsf{rights}\ c \subseteq \mathsf{rights}\ c'$

where the notation $c :< C$ means that the capability set C provides at least as much authority as the capability c.

If an entity can leak a capability to another, we say that the entities are connected, which we define as $s \vdash x \leftrightarrow y = s \vdash x \rightarrow y \lor s \vdash y \rightarrow x$. The connected relation is drawn with doubled lines and dual arrows in diagrams. The three ways two entities can be connected are shown in Fig. 3.

The invariant property of the system relating to propagation of authority is the symmetric, transitive closure of the leak relation, denoted $s \vdash x \leftrightarrow^* y$. The remaining subsections argue informally that none of the operations of the system can connect disconnected entities, which will imply that we can create authority-confined subsystems. We introduce the formal definitions of the operations as we progress through the proof. The entire proof is around 3000 lines of Isabelle script; here we show only the key lemmas.

4.3 Operations

Operations transform the state of the security model. In the Elkaduwe model they must be deterministic but in our model they may be non-deterministic.

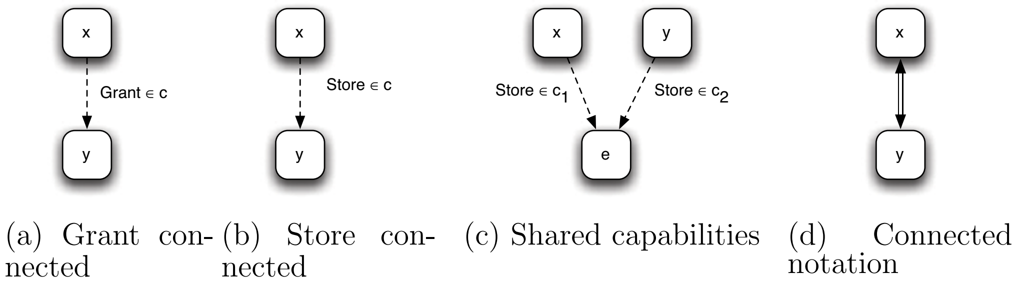


Fig. 3. Different types of connections between entities.

Non-determinism allows us to refine the explicitly checked failure conditions of the operational model. The main difference is that an execution step now returns a set of states rather than a single state.

We will first give informal definitions of the operations and argue that their respective executions can never connect previously disconnected entities; neither through transitive *grant* capabilities, nor through the sharing of capabilities. The precise definitions of the operations are shown in Fig. 7. The arguments are formalised as lemmas in the next subsection.

Operations are only executed if they are legal, that is, if certain preconditions about their arguments and the state are true, which are also shown in Fig. 7.

Neither the SysRead $e\ c$ nor the SysWrite $e\ c$ operation change the state of capabilities in the system. They clearly do not connect disconnected entities.

The SysCreate $e\ n\ c_1\ c_2$ operation creates a new entity (n) by using free memory provided by an existing entity (e_1) and by assigning a new capability for controlling access to n to existing capability storage (e_2). It is only legal if the initiating entity e has both a capability (c_1) with create rights to e_1, and a capability (c_2) with store and write rights to the CNode e_2 where the new entity's capability is stored.

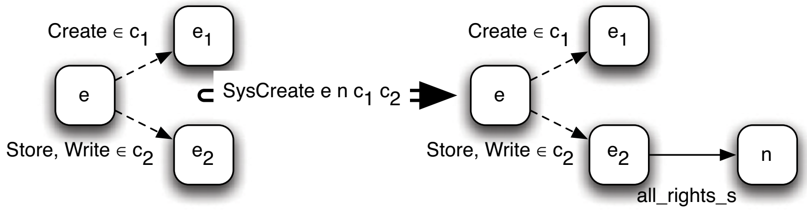


Fig. 4. Create Operation

This operation cannot connect disconnected entities as the only new connection is to a new entity.

The SysGrant $e\ c_1\ c_2\ R\ c_3$ operation gives (a possibly reduced) copy of an existing capability to another entity.

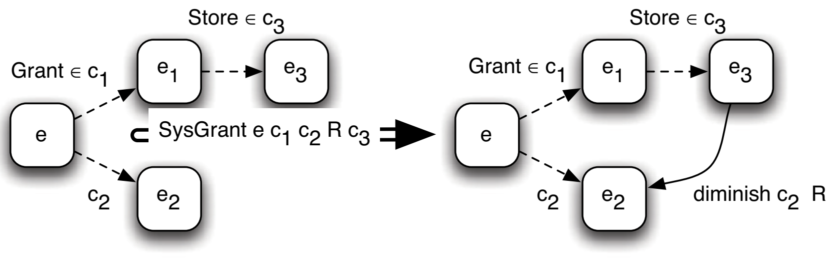
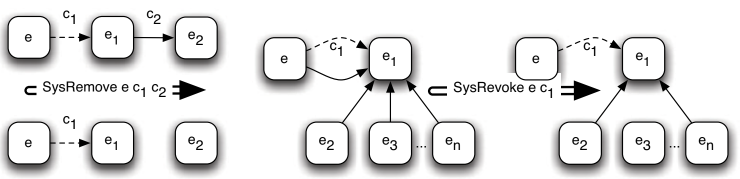


Fig. 5. Grant Operation

This operation clearly has the possibility of adding a capability to an entity that did not previously possess this capability. In fact, if an entity x is store connected to e_3 and e_2 store connected to y, then introducing a store connection between e_3 and e_2 will connect x and y. However a connection between e_3 and e_2 is only ever created if a connection existed between e and e_2, and since e and e_3 are already connected, any connections introduced are already transitively present beforehand.

Both the SysRemove $e\ c_1\ c_2$ and SysRevoke $e\ c$ operations remove capabilities: in the former case from the entity pointed to by c_1 and in the latter case from a whole set of entities. As in the Elkaduwe model, we do not specify explicitly which set of entities is removed by revoke, because this set is tracked in a complex data structure cdt (capability derivation tree) in the implementation that adds nothing that is relevant for our purposes to the security analysis. Our formulation with nondeterminism makes this more natural than before. Given this set, the revoke operation is then just a repeated call of remove.



(a) Remove Operation

(b) Revoke Operation

Fig. 6. Revoke and remove operations

Clearly neither remove or revoke connect disconnected entities.

SysDelete e is a new operation introduced here. *SysDelete* removes an entity from the state. In the seL4 implementation this allows the kernel to re-use

$$\begin{array}{ll}
\mathsf{legal} :: \mathsf{Operations} \Rightarrow \mathsf{state} \Rightarrow \mathsf{bool} & \\
\mathsf{legal}\ (\mathsf{SysRead}\ e\ c)\ s & = \mathsf{is\text{-}entity}\ s\ e \wedge c \in \mathsf{caps\text{-}of}\ s\ e \wedge \mathsf{Read} \in \mathsf{rights}\ c \\
\mathsf{legal}\ (\mathsf{SysWrite}\ e\ c)\ s & = \mathsf{is\text{-}entity}\ s\ e \wedge c \in \mathsf{caps\text{-}of}\ s\ e \wedge \mathsf{Write} \in \mathsf{rights}\ c \\
\mathsf{legal}\ (\mathsf{SysCreate}\ e\ n\ c_1\ c_2)\ s & = \mathsf{is\text{-}entity}\ s\ e \wedge \neg\ \mathsf{is\text{-}entity}\ s\ n \wedge \{c_1, c_2\} \subseteq \mathsf{caps\text{-}of}\ s\ e \wedge \\
& \quad \mathsf{Create} \in \mathsf{rights}\ c_1 \wedge \mathsf{Write} \in \mathsf{rights}\ c_2 \wedge \mathsf{Store} \in \mathsf{rights}\ c_2 \\
\mathsf{legal}\ (\mathsf{SysGrant}\ e\ c_1\ c_2\ r\ c_3)\ s & = \mathsf{is\text{-}entity}\ s\ e \wedge \{c_1, c_2\} \subseteq \mathsf{caps\text{-}of}\ s\ e \wedge c_3 \in \mathsf{caps\text{-}of}\ s\ (\mathsf{entity}\ c_1) \wedge \\
& \quad \mathsf{Grant} \in \mathsf{rights}\ c_1 \wedge \mathsf{Store} \in \mathsf{rights}\ c_3 \\
\mathsf{legal}\ (\mathsf{SysRemove}\ e\ c_1\ c_2)\ s & = \mathsf{is\text{-}entity}\ s\ e \wedge c_1 \in \mathsf{caps\text{-}of}\ s\ e \\
\mathsf{legal}\ (\mathsf{SysRevoke}\ e\ c)\ s & = \mathsf{is\text{-}entity}\ s\ e \wedge c \in \mathsf{caps\text{-}of}\ s\ e \\
\mathsf{legal}\ (\mathsf{SysDelete}\ e)\ s & = \mathsf{is\text{-}entity}\ s\ e \wedge e \notin \mathsf{entity}\ \text{`}\ \mathsf{all\text{-}caps}\ s
\end{array}$$

$$\begin{array}{ll}
\mathsf{step}' :: \mathsf{Operations} \Rightarrow \mathsf{state} \Rightarrow \mathsf{state\ set} & \\
\mathsf{step}'\ (\mathsf{SysCreate}\ e\ n\ c_1\ c_2)\ s & = \{\mathsf{let}\ \mathit{new\text{-}cap} = (\!|\mathsf{entity} = n,\ \mathsf{rights} = \mathsf{all\text{-}rights}|\!); \\
& \qquad \mathit{newTarget} = \{\mathit{new\text{-}cap}\} \cup \mathsf{direct\text{-}caps\text{-}of}\ s\ (\mathsf{entity}\ c_2) \\
& \quad \mathsf{in}\ s(n \mapsto \mathit{null\text{-}entity},\ \mathsf{entity}\ c_2 \mapsto \mathit{newTarget})\} \\
\mathsf{step}'\ (\mathsf{SysGrant}\ e\ c_1\ c_2\ R\ c_3)\ s & = \{s(\mathsf{entity}\ c_3 \mapsto \{c_2(\!|\mathsf{rights} := \mathsf{rights}\ c_2 \cap R|\!)\} \\
& \quad \cup\ \mathsf{direct\text{-}caps\text{-}of}\ s\ (\mathsf{entity}\ c_3))\} \\
\mathsf{step}'\ (\mathsf{SysRemove}\ e\ c_1\ c_2)\ s & = \{\mathsf{removeOperation}\ e\ c_1\ c_2\ s\} \\
\mathsf{step}'\ (\mathsf{SysRevoke}\ e\ c)\ s & = \mathsf{if}\ \mathsf{is\text{-}entity}\ s\ e \wedge c \in \mathsf{caps\text{-}of}\ s\ e\ \mathsf{then}\ \mathsf{revokeOp}\ e\ s\ \text{`}\ \mathsf{cdt}\ s\ c\ \mathsf{else}\ \{s\} \\
\mathsf{step}'\ (\mathsf{SysDelete}\ e)\ s & = \{s(e := \mathsf{None})\}
\end{array}$$

where

$$\begin{array}{ll}
\mathsf{removeOperation}\ e\ c_1\ c_2\ s & = \mathsf{case}\ s\ (\mathsf{entity}\ c_1)\ \mathsf{of}\ \mathsf{None} \Rightarrow s \mid \mathsf{Some}\ C \Rightarrow s(\mathsf{entity}\ c_1 \mapsto C - \{c_2\}) \\
\mathsf{revokeOp}\ \mathit{sRef}\ s\ \mathit{xs} & = \mathsf{foldr}\ (\mathsf{removeCaps}\ \mathit{sRef})\ \mathit{xs}\ s \\
\mathsf{removeCaps}\ e\ (c, \mathit{cs})\ s & = \mathsf{foldr}\ (\mathsf{removeOperation}\ e\ c)\ \mathit{cs}\ s
\end{array}$$

Fig. 7. Definition of, and preconditions for executing operations.

memory and the preconditions and book-keeping required for this operation to be safely usable is complex. On the security level it can be expressed very abstractly and nicely: An entity e is safe to delete, if none of the entities in the state have capabilities that point to the entity to be deleted. In the security model it would be sufficient to say that no *other* entity in the state has a capability to e, but the seL4 implementation will clean up e itself first in any case, therefore it is no restriction to simplify the formula.

Since we define the state as a partial function from entity-ids to *entities*, the definition of executing delete is trivial. *SysDelete* simply removes an entity; therefore it cannot connect disconnected entities. We will see in the next subsection that delete has an interesting side effect that slightly changes the formal statement of theorem 4.4 and theorem 4.5 as well as theorem 5.1.

Operations are governed by the functions legal and step′, which are both presented in Fig. 7. The former gives preconditions for operations, which are only permitted if the entity involved e exists and has the required capabilities. The latter describes the effect of operations on the state.

We can now define full single-step execution and lift it to sequences of operations.

A function step combines precondition checks on the current state, from legal, with the effects of operations, from step′. In contrast to the Elkaduwe model, operations now return a set of possible states. The definition of step ensures that the result always includes the initial state, thereby accounting for the possibility that an implementation of seL4 may abort an operation for reasons that are ignored in step′. For example, an implementation of the SysCreate operation will fail if it is not provided with sufficient memory, but this kind of detail is irrelevant in the security model.

$\mathsf{step} :: \mathsf{Operations} \Rightarrow \mathsf{state} \Rightarrow \mathsf{state\ set}$
$\mathsf{step}\ cmd\ s \equiv \mathsf{if\ legal}\ cmd\ s\ \mathsf{then\ step}'\ cmd\ s \cup \{s\}\ \mathsf{else}\ \{s\}$

A list of operations is executed by lifting step over sets of states and iterating. Commands are read from right to left.

$\mathsf{execute} :: \mathsf{Operations\ list} \Rightarrow \mathsf{state} \Rightarrow \mathsf{state\ set}$

$\mathsf{execute}\ []\ s = \{s\}$
$\mathsf{execute}\ (cmd\ \#\ cmds)\ s = \bigcup \mathsf{step}\ cmd\ `\ \mathsf{execute}\ cmds\ s$

4.4 Execution

This subsection introduces the formal statements of the main lemmas in the security proof. We define sane as an invariant that is a precondition to most theorems in our security model. We call a state sane if all capabilities point to entities that exist.

$\mathsf{sane} :: \mathsf{state} \Rightarrow \mathsf{bool}$
$\mathsf{sane}\ s \equiv \forall\, c{\in}\mathsf{all\text{-}caps}\ s.\ \mathsf{is\text{-}entity}\ s\ (\mathsf{entity}\ c)$
where $\mathsf{all\text{-}caps}\ s \equiv \bigcup_{e}\ \mathsf{direct\text{-}caps\text{-}of}\ s\ e$

We have shown that sane is invariant over execution:

$[\![\mathsf{sane}\ s;\ s' \in \mathsf{execute}\ cmds\ s]\!] \Longrightarrow \mathsf{sane}\ s'$

We have argued informally in the last subsection that none of the operations in the kernel will connect previously disconnected entities. Formally this is the following statement.

Lemma 4.1 *If two entities in state s are connected after an execution step, they must have been transitively connected before:*

$[\![\mathsf{sane}\ s;\ s' \in \mathsf{step}\ cmd\ s;\ \mathsf{is\text{-}entity}\ s\ x;\ \mathsf{is\text{-}entity}\ s\ y;\ s' \vdash x \leftrightarrow y]\!]$
$\Longrightarrow s \vdash x \leftrightarrow^{*} y$

The high-level structure of the proof is the same as in Elkaduwe et al [7]. As there, we cannot directly lift lemma 4.1 to the transitive and reflexive closure, such that $[\![s' \in \mathsf{step}\ cmd\ s;\ s' \vdash x \leftrightarrow^* y]\!] \Longrightarrow s \vdash x \leftrightarrow^* y$. We break the proof into two parts: the *SysCreate* operation that introduces a complication for this lifting step and *transporter* commands which move or remove capabilities (all other operations). The second part has the simpler formal statement:

Lemma 4.2 *Transporters preserve connected in sane states:*[2]

$$\begin{array}{l}[\![\mathsf{sane}\ s;\ s' \in \mathsf{step}\ cmd\ s;\ \mathsf{is\text{-}entity}\ s\ x;\ \mathsf{is\text{-}entity}\ s'\ x;\\ \ \forall\, e\ n\ c_1\ c_2.\ cmd \neq \mathsf{SysCreate}\ e\ n\ c_1\ c_2;\ s' \vdash x \leftrightarrow^* y]\!]\\ \Longrightarrow s \vdash x \leftrightarrow^* y\end{array}$$

Note that we picked up an additional precondition compared to the Elkaduwe model. We now need to know that entity x still exists in the post-state. This is to get us later through the induction on sequences of operations with *SysDelete*. To see why this is the case, consider the case when an entity with id_1 and an entity with id_2 are disconnected. If entity 2 gets deleted, and entity 1 possesses a create right, then entity 1 is free to create a new entity with id_2 in a later step. This new entity 2 is now connected to entity 1. Of course, entity 1 did not gain access to the original entity 2, but just to a new, different entity stored in the same location.

Instead of strengthening the precondition, we could get around this by introducing names for entities that are unique not only over the lifetime of the entity, but over the lifetime of the whole system. These names would have to come from an infinite set. We chose not to do so, because this problem exists in the implementation as well, and there is no way to implement such system life-time names in reality unless the system runs for a known finite time only. As for shared capability storage, we believed it better to bring the problem out into the open and reflect it explicitly in the security analysis.

The *create* lemma remained almost the same, we merely need to add a failure case. Note that the definition of $\leftrightarrow^*$ and therefore the proof is now significantly more complex, though.

Lemma 4.3 *Given entities x and y connected in the state after* SysCreate e n c_1 c_2*, given that x exists in the pre-state s, and given that* sane s*, we know $s \vdash x \leftrightarrow^* e$ if y is the new entity just created, or $s \vdash x \leftrightarrow^* y$, or the operation failed. Formally:*

$$\begin{array}{l}[\![\mathsf{sane}\ s;\ s' \in \mathsf{step}\ (\mathsf{SysCreate}\ e\ n\ c_1\ c_2)\ s;\ \mathsf{is\text{-}entity}\ s\ x;\ s' \vdash x \leftrightarrow^* y]\!]\\ \Longrightarrow (\mathsf{if}\ y = n\ \mathsf{then}\ \neg\ \mathsf{is\text{-}entity}\ s\ y \wedge s \vdash x \leftrightarrow^* e\ \mathsf{else}\ s \vdash x \leftrightarrow^* y) \vee s = s'\end{array}$$

[2] is-entity $s\ y$ is implied by the other preconditions, thus is not necessary.

Bringing the two lemmas together, we get the final theorem on the *connected* relation:

Theorem 4.4 *If two entities in a sane state s are transitively connected after execution, they already have been transitively connected in s:*

$$\llbracket \mathsf{sane}\ s;\ s' \in \mathsf{execute}\ cmds\ s;\ \mathsf{is\text{-}always\text{-}entity}\ cmds\ s\ x;\ \mathsf{is\text{-}always\text{-}entity}\ cmds\ s\ y;\ s' \vdash x \leftrightarrow^* y \rrbracket \Longrightarrow s \vdash x \leftrightarrow^* y$$

The new predicate is-always-entity states that an entity exists in all intermediary execution states. It is the consequence of our strengthened precondition in lemma 4.2.

4.5 Subsystems

With the notion of connected established, we can now generalise it to subsystems as before and show that it is still possible to implement authority confinement between subsystems in seL4 when shared capability storage is present.

Authority confinement means we can partition entities into *subsystems* such 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 an authority is already present, then we show that it cannot be increased. Note that subsystems can grow over time. We identify a subsystem by an entity within it, thus we require that this entity remain in existence for the period in question. As in the main theorem on the *connected* relation, this requirement is new and due to the delete command.

We can produce a group of subsystems by having a master entity first creating some child entities, and then by removing the grant and store capabilities to these child entities.

Following Elkaduwe et al, we define such subsystems using the symmetric, transitive closure over the leak relation. This partitions entities up into equivalence classes.

$$\mathsf{subsys}\ s\ x \equiv \{e_i.\ s \vdash e_i \leftrightarrow^* x\}$$

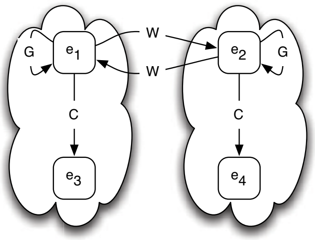


Fig. 8: An example of two isolated subsystems. They can communicate, but the capabilities between them cannot increase.

We also introduce the same notion of a *dominates* operator :> and use it to express that a subsystem cannot increase its authority over a specific entity. For this, we also need to collect all entities of a subsystem with the subsys-caps function:

$$\mathsf{subsys\text{-}caps}\ s\ x \equiv \bigcup \mathsf{caps\text{-}of}\ s\ \text{`}\ \mathsf{subsys}\ s\ x$$

$$c :> C \equiv \forall\, c'{\in}C.\ \mathsf{entity}\ c' = \mathsf{entity}\ c \longrightarrow \mathsf{rights}\ c' \subseteq \mathsf{rights}\ c$$

The definitions are unchanged from Elkaduwe et al. The subsys-caps function takes the set of entities in the subsystem, and then the union of all their capabilities. Here, these are all capabilities that the entities transitively have access to. A capability c dominates a capability set C ($c :> C$) if C provides at most as much authority as capability c over the entity c points to.

Since capabilities of an entity are those that are stored directly in entities that are store connected, and since those entities would also be part of the same subsystem, we can prove an equivalent definition for subsys-caps. This is easier to work with, because it removes one level of transitive closure.

$$\mathsf{subsys\text{-}caps}\ s\ x = \bigcup \mathsf{direct\text{-}caps\text{-}of}\ s\ \text{`}\ \mathsf{subsys}\ s\ x$$

The equality allows us to ignore the impact of shared capabilities on the transfer of capabilities between subsystems, but the SysDelete operation still requires the added precondition that the entities in question exist continually. Following essentially the same logic as [8], we can prove the result:

Theorem 4.5 (Confinement of authority). *Given a sane state s, a non-empty subsystem spanned by x in s, and a capability c with a target identity y 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 for as long as x and y both exist.*

$$\begin{array}{l}[\![\mathsf{sane}\ s;\ s' \in \mathsf{execute}\ \mathit{cmds}\ s;\ \mathsf{entity}\ c = y;\ \mathsf{is\text{-}always\text{-}entity}\ \mathit{cmds}\ s\ x;\\ \ \mathsf{is\text{-}always\text{-}entity}\ \mathit{cmds}\ s\ y;\ c :> \mathsf{subsys\text{-}caps}\ s\ x]\!]\\ \Longrightarrow c :> \mathsf{subsys\text{-}caps}\ s'\ x\end{array}$$

5 Isolation

The security analysis so far was concerned with de-jure rights, i.e. rights that are directly conferred by capabilities. If we are interested in the flow of information through the system, then we need to consider de-facto rights. As mentioned previously, de-facto rights model entities that may try to collaborate to transport information through indirectly authorised channels. If A has read access to B, and C has write access to B, then de facto, A has read access to C even though de jure, no read operation from A to C will ever be authorised by the kernel.

With entities divided up into subsystems as in the last section, we can examine the flow of information between subsystems in this sense. We do not examine the flow of information within a subsystem as capabilities can be transported between entities within a subsystem by definition already. Bishop's [1] analysis of information flow between *islands* is essentially the same concept.

Unlike in Sect. 4.5 where we examined the symmetric closure of the leak relation, information flow is a directed relation. Elkaduwe's original extension to showing isolation in the information flow sense is still bidirectional [5].

Since we know that the capabilities between subsystems cannot increase, we can conclude that paths for information flow cannot increase between subsystems over time. This motivates the following definition.

We say that information can *flow* from one set of entities X to another set of entities Y if either an entity of X has a *write* capability to an entity in Y, or if an entity of Y has a *read* capability to an entity in X, which we write as $(X,\ Y) \in \textit{set-flow}\ s$.

Formally:

$$((X,\ Y) \in \textit{set-flow}\ s) =$$
$$(\exists\, x{\in}X.\ \exists\, y{\in}Y.\ \mathsf{read\text{-}cap}\ x :< \mathsf{caps\text{-}of}\ s\ y \vee \mathsf{write\text{-}cap}\ y :< \mathsf{caps\text{-}of}\ s\ x)$$

From this we define the flow of information between subsystems.

$$s \vdash x \leadsto y = ((\mathsf{subsys}\ s\ x,\ \mathsf{subsys}\ s\ y) \in \textit{set-flow}\ s)$$

If we take the transitive closure of this relation we can establish when information can flow between two subsystems over time.

Theorem 5.1 *If information cannot flow between the subsystems containing x and y in the present, then information will never be able to flow between any future subsystems that x and y are in (for as long as x and y both exist).*

$$\llbracket \mathsf{sane}\ s;\ \mathsf{is\text{-}always\text{-}entity}\ \textit{cmds}\ s\ x;\ \mathsf{is\text{-}always\text{-}entity}\ \textit{cmds}\ s\ y;$$
$$s' \in \mathsf{execute}\ \textit{cmds}\ s;\ \neg\ s \vdash x \leadsto^* y\rrbracket$$
$$\Longrightarrow \neg\ s' \vdash x \leadsto^* y$$

This theorem was proven by showing that each operation cannot create a new flow of information. By induction over a sequence of commands, we can deduce the contrapositive of the above theorem. The main difficulty in the proof comes from examining all the various possible cases produced by SysCreate.

Thus, if you can examine the capabilities of the system, you can predict the possibly indirect flows of information through the system they authorise.

6 Related work

The first to formulate an access control analysis were Harrison et al [9]. They introduced the focus on the ability of a subject to obtain a particular authority over another in some future state.

As mentioned previously, our seL4 access control model is inspired by the take-grant model [12]. The original analysis on the take grant model [12, 2] already uses the same approximation to model the exposure of access rights: the transitive, symmetric closure on the given initial graph. The difference here is that we make capability storage and sharing explicit and that we conduct all proofs in Isabelle/HOL.

Snyder [17] and later Bishop [1] introduced explicit de-facto rules into the take-grant model for reasoning about the information flow paths induced by the capability distribution. Their concept of the maximum take-grant connected subgraph is similar to our subsystems. In contrast to Bishop and Snyder, we do not need to introduce additional rules into the specification; instead we model the impact of de-facto rights directly in the isolation relation.

Shapiro [16] applied the *diminish-take* model—another variant of take-grant to capture the operational semantics of the *EROS* system. The ability of seL4 to diminish access rights during the grant operation is inspired by this diminish-take model.

None of the formalisations above model sharing of capabilities explicitly, and none of them formalise and prove their security statements in a theorem prover.

The most closely related work to this paper is the model of seL4 access control by Elkaduwe et al [5, 7, 8] that we directly build on and extend.

Rushby [15] provides a formulation of isolation called non-interference. Non-interference is stronger than the concept of isolation we use in this formalisation, because it goes beyond access control. It also includes covert storage channels, whereas we are only concerned with the overt, explicitly authorised, but still possibly indirect flows of information. The difference is that non-interference would for instance cover things like leaking information by making an observable decision in the program that depends on secret data. Non-interference traditionally talks about different security levels that should be kept separate, whereas we are in this model more interested in which entities can communicate with each other in a highly dynamic setting. Non-interference is not necessarily preserved under refinement, so special care would need to be taken to connect such a property to the operational model and the C implementation of seL4.

7 Conclusion

We have presented three extensions to the existing formal access control model of the seL4 microkernel. The extensions are the inclusion of non-determinism, a delete operation for entities, and an explicit treatment of shared capability storage.

The first extension was easy to add, the second introduced minor changes to the main theorems, because the concept of entity identity becomes more subtle. Explicitly modelling shared capabilities is the main contribution of this paper. Sharing makes the access control model more complex and changes the fundamental notions of the main security theorem. We have adjusted the corresponding definitions and proved that the access control model is still decidable.[3] This is a significant result, because not all access control systems are decidable [9, 13], and previous arguments on capability sharing we have found in the literature were of the very high-level "without loss of generality" kind. This main theorem implies that a monitor for system-global security policies can be implemented.

Our more explicit access control model represents an almost complete one-to-one correspondence to the existing operational model of seL4. The next step in this project will be to complete the refinement proof of the access control to the operational model and therefore to the C implementation of seL4, justifying the *s* in seL4.

Acknowledgement

We would like to thank Gerwin Klein, Timothy Bourke and June Andronick for reading and commenting on drafts of this paper. We also thank the seL4 design team and Dhammika Elkaduwe for discussions on the access control model and their advice.

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 *Proc. 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 O. A. Mohamed, C. M. noz, and S. Tahar, editors, *Proc. 21st TPHOLs*, volume 5170 of *LNCS*, pages 167–182, Montreal, Canada, Aug. 2008. Springer.

[3] As in the classical take-grant-style models, strictly speaking, the predicates in the main theorem decide only an approximation of the future system state.

[4] P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *Proc. ACM SIGPLAN Haskell WS*, Portland, OR, USA, Sept. 2006.

[5] D. Elkaduwe. *A Principled Approach To Kernel Memory Management.* PhD thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Aug. 2008.

[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] D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. Technical report, NICTA, Oct. 2007. Available from http://ertos.nicta.com.au/publications/papers/Elkaduwe_GE_07.pdf.

[8] D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. In J. Woodcock and N. Shankar, editors, *Proc. VSTTE 2008 — Verified Softw.: Theories, Tools & Experiments*, LNCS, Toronto, Canada, Oct. 2008. Springer.

[9] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *CACM*, pages 561–471, 1976.

[10] G. Klein. Operating system verification — an overview. *Sadhana*, 2009. To appear.

[11] J. Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, December 1995.

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

[13] J. MacLean. Reasoning about security models. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 123–131. IEEE, Apr. 187.

[14] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[15] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI, Dec. 1992. http://www.csl.sri.com/papers/csl-92-2/.

[16] J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. pages 166–181, Washington, DC, USA, May 2000.

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