seL4 Enforces Integrity

Thomas Sewell[1], Simon Winwood[1,2], Peter Gammie[1], Toby Murray[1,2], June Andronick[1,2], and Gerwin Klein[1,2]

[1] NICTA, Sydney, Australia[*]
[2] School of Computer Science and Engineering, UNSW, Sydney, Australia

{first-name.last-name}@nicta.com.au

Abstract. We prove the enforcement of two high-level access control properties in the seL4 microkernel: integrity and authority confinement. Integrity provides an upper bound on write operations. Authority confinement provides an upper bound on how authority may change. Apart from being a desirable security property in its own right, integrity can be used as a general framing property for the verification of user-level system composition. The proof is machine checked in Isabelle/HOL and the results hold via refinement for the C implementation of the kernel.

1 Introduction

Access control is one of the primary security functions that operating system (OS) kernels provide. It is used to enforce confidentiality and its dual, integrity. Confidentiality means that information may not be read without read authority. Integrity means that information may not be written without write authority. These properties have been analysed extensively in classical approaches such as the Bell-LaPadula model [3]. For dynamic access control systems, such as the capability system in the seL4 microkernel, an additional property is needed: authority confinement. Authority confinement means that without explicit authority to do so, authority may not be propagated to other subjects.

In previous work, we have verified the functional correctness of seL4 [12]. In the work presented here we capture a high-level security goal of the kernel, and prove that it correctly enforces two of the above properties: integrity and authority confinement.

Both properties can be seen as an upper bound on the behaviours of the system. Given a policy, an abstraction of the system, integrity constrains the set of possible state mutations. Similarly, authority confinement tells us that however authority spreads dynamically within the system, the propagation of authority is bounded by the policy abstraction. The policy provides mandatory access control bounds; within these bounds access control is discretionary.

While integrity is an important security property on its own, it is of special interest to formal system verification. It provides a simple framing condition for

the execution of user-level components, because it can tell us which parts of the system do not change. In a rely-guarantee framework, the integrity property can give us a powerful guarantee condition without the need to consult the code of the component. By virtue of the policy setup and the integrity theorem on the kernel alone, we can compute an upper bound of the behaviour of a component. This becomes especially important if the system contains components that are otherwise beyond our means for formal code-level verification, such as a Linux guest operating system with millions of lines of code. We can now safely and formally compose such parts with the rest of the system.

Neither framing conditions nor access control properties are new. Proving these properties about a real OS kernel implementation, however, has not been achieved before [10]. Specifically, the novelty and contributions of this work are:

- The Isabelle/HOL formalisation and generalisation of integrity and authority confinement for a real microkernel implementation.
- To the best of our knowledge the first code-level proof of high-level access control properties of a high-performance OS kernel.

Since our proof is rooted in reality via refinement to the C implementation we could not, on encountering complexity and corner cases, just simplify the model and claim no loss of generality. Instead, the choice was to either modify the kernel implementation and its correctness proof, or to treat the full complexity at the level of access control.

The integrity theorem is stated relative to an abstract policy and therefore allows us to make simplifying assumptions in one direction: this policy. By constraining the classes of policies the theorem applies to, we forbid some unnecessary corner cases that, we claim, are not needed in practise. Although we have not yet applied the theorem to a large system, our choice of assumptions has been guided by a previous case study [2] on a secure network access device (SAC), with a dynamic and realistic security architecture.

As far as we are aware, we have made no assumptions that would be invalid for the architecture of such a system. This leaves us with considerable generality: the theorem supports fine grained components, communication between them via memory sharing and message passing, delegation of authority to subsystems, dynamic creation and deletion of objects, controlled propagation of authority, and controlled policy reconfiguration at runtime.

In the following, Sect. 2 gives a brief introduction to access control in general and to the seL4 access control system in particular. Sect. 2 also introduces a part of the aforementioned SAC system as a running example. Sect. 3 gives a summary of the formalisation as well as the final Isabelle/HOL theorems and Sect. 4 discusses the results together with our experience in proving them.

2 Access Control Enforcement and seL4

This section introduces relevant concepts from the theory of access control and our approach to instantiating them for seL4. For ease of explanation we will first
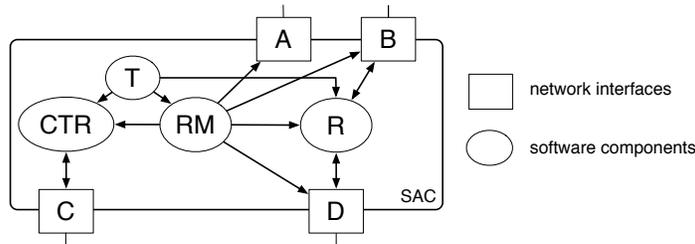
Fig. 1. System snapshot, routing between data D and back-end B network.

introduce a running example, then use it to describe the seL4 security mechanisms available, and then compare to the theory of access control.

2.1 Example

The secure access controller (SAC) was designed in an earlier case study of ours [2] and will serve as a running example in this work. The purpose of the SAC is to switch one front-end terminal between different back-end networks of separate classification levels. The security goal of the system is to avoid information flow between the back-end networks.

Fig. 1 shows the main components of the SAC as nodes and their logical authority connections as edges. To the outside, the SAC provides four network interfaces: two back-end networks A and B, one control network C, and one data network D. Networks C and D are attached to a front-end terminal. The purpose of the SAC is to connect either A to D or B to D at a time without information flow between A and B. Internally, we have four components: a timer T, a controller user interface CTR, the router manager RM, and a router instance R. The router manager RM will upon a switch request, tear down R, remove all access from it, create and start a fresh R component, and connect it to the requested networks. RM is a small, trusted component and has access to all networks. Verification needs to show that it does not abuse this access. The router R, on the other hand, is a large, untrusted instance of Linux. It will only ever be given access to one of the back-end networks during its lifetime.

In previous work [2], we have shown that, assuming a specific policy setup, assuming correct behaviour of RM, and assuming that the kernel correctly enforces access control, the security goal of the system will be enforced.

The work in this paper helps us to discharge the latter two assumptions: we can to use the integrity property as a framing condition for verifying the behaviour of RM, and we can use the same property to make sure that R stays within its information flow bounds. To complete the verification, we would additionally need the confidentially side of access control as well as a certified initial policy set up. Both are left for future work.

2.2 seL4

The seL4 microkernel is a small operating system kernel. As a microkernel, it provides a minimal number of services to applications: interprocess communication, threads, virtual memory, access control, and interrupt control.

As mentioned, seL4 implements a capability-based access control system [6]. Services, provided by a set of methods on kernel implemented objects, are invoked by presenting to the kernel a capability that refers to the object in question and has sufficient access rights for the requested method. For the purposes of this paper, the following four object classes are the most relevant.

CNodes Capabilities are stored in kernel-protected objects called *CNodes*. These CNodes can be composed into a *CSpace*, a set of linked CNodes, that defines the set of capabilities possessed by a single thread. CNode methods allow copying, insertion and removal of capabilities. For a thread to use a capability, this capability must be stored in the thread's CSpace. CNodes can be shared across CSpaces. The links in Fig. 1 mean that the collective CSpaces of a component provide enough capabilities to access or communicate with another component.

Virtual Address Space Management A virtual address space in seL4 is called a *VSpace*. In a similar way to CSpaces, a VSpace is composed of objects provided by the microkernel. On ARM and Intel IA32 architectures, the root of a VSpace consists of a *Page Directory* object, which contains references to *Page Table* objects, which themselves contain references to *Frame* objects representing regions of physical memory. A Frame can appear in multiple VSpaces, and thereby implement shared memory between threads or devices such as the SAC networks.

Threads Threads are the unit of execution in seL4, the *subjects* in access control terminology. Each thread has a corresponding *TCB* (thread control block), a kernel object that holds its data and provides the access point for controlling it. A TCB contains capabilities for the thread's CSpace and VSpace roots. Multiple threads can share the same CSpace and VSpace or parts thereof. A component in the SAC example may consist of one or more threads.

Inter-process Communication (IPC) Message passing between threads is facilitated by *Endpoints* (EP). The kernel provides *Synchronous Endpoints* with rendezvous-style communication, and *Asynchronous Endpoints* (AEP) for notification messages. Synchronous endpoints can also be used to transfer capabilities if the sender's capability to the endpoint has the *Grant* right. The edges in the SAC example of Fig. 1 that are physical communication links, are implemented using endpoints.

The mechanisms summarised above are flexible, but low-level, as customary in microkernels. Fig. 2 shows parts of the implementation of the link between the CTR and RM components of Fig. 1. The link is implemented via a synchronous endpoint EP. The CTR and RM components both consist of one main thread, each with a CSpace containing a capability to the endpoint (among others), and
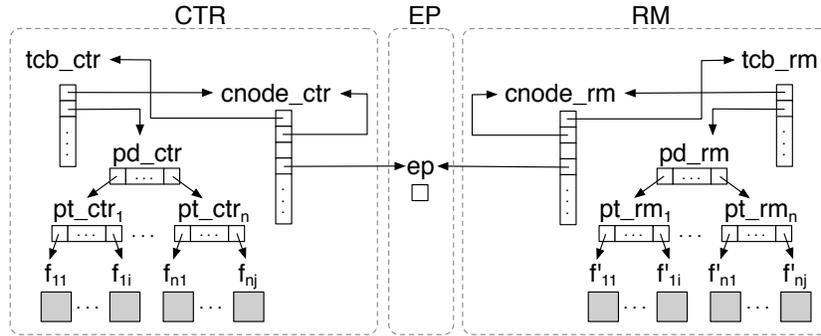
Fig. 2. SAC capabilities (partial).

each with a VSpace containing page directories (pd), page tables (pt), and frames f_n implementing private memory.

These mechanisms present a difficulty for access control due to their fine grained nature. Once larger components are running, there can easily exist hundreds of thousands of capabilities in the system. In addition, seL4 for ARM has 15 different capability types, several of which have specific rights and variations. We will prune this complexity down by making some simplifying observations. One such observation is that many kinds of capabilities will not be shared without the sharing components trusting each other. For example sharing part of a CSpace with an untrusted partner makes little sense, as seL4 does not provide a mechanism for safely using a capability of unknown type.

2.3 Access Control Enforcement

An access control system controls the access of *subjects* to *objects* [14], by restricting the operations that subjects may perform on objects in each state of the system. As mentioned above, in seL4 the subjects are threads, the objects are all kernel objects, including memory pages and threads themselves.

The part of the system state used to make access control decisions, i.e., the part that is examined by the kernel to decide which methods each subject may perform on which object, is called the *protection state*. In seL4, this protection state is mostly represented explicitly in the capabilities present in the CSpace of each subject. This explicit, fine-grained representation is one of the features of capability-based access control mechanisms. In reality, however, some implicit protection state remains, for instance encoded in the control state of a thread, or in the presence of virtual memory mappings in a VSpace.

The protection state governs not only what operations are allowed to be performed, but also how each subject may modify the protection state. For instance, the authority for capabilities to be transmitted and shared between subjects is itself provided by CNode or endpoint capabilities.

As seen previously in Fig. 2, the protection state of a real microkernel can be very detailed, and therefore cumbersome to describe formally. It is even more
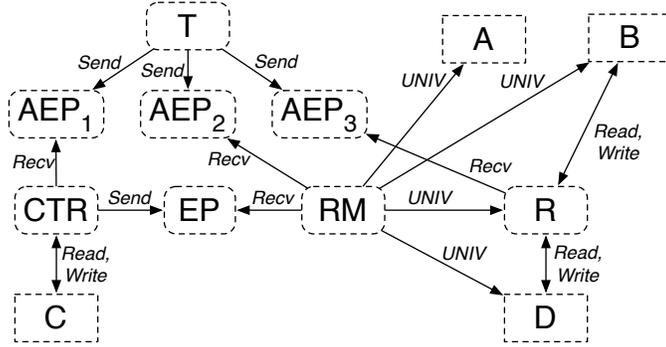
Fig. 3. SAC authority (except self-authority).

cumbersome to describe precisely what the allowed effects of each operation are at this level. Hence we make use of the traditional concept of a policy which can be seen as an abstraction of the protection state: we assign a label to each object and subject, and we specify the access rights between labels as a directed graph. The abstraction also maps the many kinds of authority in seL4 protection states into a simple enumerated set of authority types. This simplifies the setup in three ways: the number of labels can be much smaller than the number of objects in the system, the policy is static over each system call whereas the protection state may change and, finally, we can formulate which state mutations are allowed by consulting the policy, rather than the more complex protection state.

The abstraction level of the policy is constrained only by the wellformedness assumptions we make in Sect. 3. Within these bounds it can be chosen freely and suitably for any given system.

Fig. 3 shows an abstract policy, only mildly simplified for presentation, that corresponds to a possible protection state of the SAC at runtime. The objects in the system are grouped by labels according to the intention of the component architecture. The RM label, for instance, includes all objects in the RM box of Fig. 2. The communication endpoints between components have their own label to make the direction of information flow explicit. The edges in the figure are annotated with access rights, *UNIV* indicates the universal set of all rights.

Correct access control enforcement, with respect to a security policy, can be decomposed into three properties about the kernel: *integrity*, *confidentiality* and *authority confinement*. The two former properties are the same as introduced in Sect. 1, only the notion of what is permitted is taken from the policy rather than the security state. The latter property ensures the current subject cannot escalate its authority (or another subject's) above what the policy allows.

Note that we may have some components in the system, such as RM in the SAC, which have sufficient authority to break authority confinement. The authority confinement theorem will assume these components are not the current subject, and we will be obliged to provide some other validation of their actions.

3 Formalisation of Integrity Preservation

This section sketches our Isabelle/HOL formalisation of the integrity and authority confinement properties. While it is impossible in the space constraints of a paper to give the full detail, we show the major definitions and the toplevel theorems to provide a flavour of the formalisation. For formal details on the kernel-level concepts beyond our description below we refer the interested reader to the published Isabelle/HOL specification of the kernel [1] that the definitions here build on.

We have already introduced the notion of a policy, an upper bound on the protection state of the system, and an accompanying abstraction, a mapping from detailed subject and object names up to a smaller set of component labels. We roll these objects together with a subject label into the PAS record (policy, abstraction, subject) which is an input to all of our access control predicates.

$$
\begin{aligned}
\textbf{record } {}'l \text{ PAS } = \quad & \text{pasPolicy} :: ({}'l \times \text{auth} \times {}'l) \text{ set} \\
& \text{pasObjectAbs} :: \text{obj-ref} \Rightarrow {}'l \\
& \text{pasIRQAbs} :: \text{irq} \Rightarrow {}'l \\
& \text{pasASIDAbs} :: \text{asid} \Rightarrow {}'l \\
& \text{pasSubject} :: {}'l
\end{aligned}
$$

The type parameter ${}'l$ here is any convenient type for component labels. The policy field is a graph (a set of triples ${}'l \times \text{auth} \times {}'l$), whose vertices are policy labels and whose edges are access rights from the type auth which will be discussed shortly. Abstraction functions are provided for seL4's namespaces: objects (i.e. system memory), interrupt request numbers and address space identifiers. Each of these is mapped up to a policy label.

The subject in the PAS record identifies the label of the current subject. We must associate all (write) actions with a subject in order to define the integrity property we are proving. We will pick the subject associated with any kernel actions at kernel entry time, choosing the label of the currently running thread. This coarse division of actions between subjects causes some problems for message transfers, as will be discussed below.

The identification of a subject makes all of our access control work *subjective*. The integrity property depends on which subject carries out an action, because whether that action is allowed or not depends on the allowed authority of the subject performing it. Wellformedness of PAS records, encapsulating our policy assumptions, will be defined in a subjective manner as well, so for any given system and policy the authority confinement proof may be valid only for less-trusted subjects which satisfy our policy assumptions.

3.1 Authority Types

We made the observation in Sect. 2 that most kinds of objects in seL4 are not shared by mutually distrusting components. We found that the objects that could be safely shared were those where authority to that object could be

partitioned between capabilities. Endpoints are a good example: Capabilities to endpoints can be send-only or receive-only. Every endpoint in the SAC has a single sending component and a single receiving component, as is typical in seL4 system architectures. Memory frames may also be read only or read-write and a typical sharing arrangement has a single writer, though possibly many readers.

This led us to our chosen formalisation of authority types.

datatype auth = Receive | Send | Reset | Grant | Write | Read | Everything

The Receive, Send, Read and Write authorities have been described above. Capabilities to endpoints may have the Grant right which permits other capabilities to be sent along with messages. The Reset authority is conferred by all capabilities to endpoints, since these capabilities can sometimes cause an endpoint reset even if they have zero rights. Finally, the Everything authority is used to represent any other kind of authority in the system; we conservatively represent any other authority type as conferring complete control over the target.

3.2   Subjective Policy Wellformedness

We aim to show authority is confined by a policy. This will not be true for all kinds of policies. If a policy gives the subject a Grant authority to any other component in addition to some authority which the other component should not have, that policy can clearly be invalidated. We define wellformedness criteria on policies, and thereby system architectures, given a specific subject, as follows. In our example Fig. 3, we would expect the policy to be wellformed for the subjects CTR, T, and R, but not RM.

policy-wellformed $policy\ irqs\ subject \equiv$
$\quad\quad (\forall\, y.\ (subject,\ \mathsf{Everything},\ y) \in policy \longrightarrow subject = y) \,\wedge$
$\quad\quad (\forall\, a.\ (subject,\ a,\ subject) \in policy) \,\wedge$
$\quad\quad (\forall\, s\ r\ ep.$
$\quad\quad\quad (s,\ \mathsf{Grant},\ ep) \in policy \wedge (r,\ \mathsf{Receive},\ ep) \in policy \longrightarrow$
$\quad\quad\quad (s,\ \mathsf{Everything},\ r) \in policy \wedge (r,\ \mathsf{Everything},\ s) \in policy) \,\wedge$
$\quad\quad (\forall\, irq\ p.\ irq \in irqs \wedge (irq,\ \mathsf{Send},\ p) \in policy \longrightarrow (subject,\ \mathsf{Send},\ p) \in policy)$

The first requirement is that the subject cannot have Everything authority over another component. If it did there would be no point in separating these components, as the subject might coerce the target into taking actions on its behalf.

The second requirement is that the subject has all kinds of authority to itself. We always consider components permitted to reconfigure themselves arbitrarily.

The Grant restriction observes that successful capability transfers over messages are as problematic for access control as Everything authority. In each direction this restriction could be lifted if we introduced more complexity.

In the sending direction the problem is that the sender can transfer an arbitrary capability into the receiver's capability space, giving the sender the new authority to rescind capabilities from the receiver's capability space in the future. It may be possible in seL4 for a receiver to partition its capability space to make this safe, but we know of no use case that justifies the resulting complexity.

In the receiving direction the problem is in the way we fix the subject of the message send. Synchronous sends in seL4 complete when both sender and receiver are ready. If the sender is ready when our subject makes a receive system call, it may appear that the receiver has broken authority confinement by inventing new rights. In fact the rights belonged to the sender, which was involved as a subject in some sense, but not in a manner that is easy for us to capture.

The final policy assumption relates to both interrupts and subject allocation. Interrupts may arrive at any time, delivering a message to a thread waiting for that interrupt. We must allow the current subject to send this message. We hope to revisit our simple notion of the current subject in future work.

3.3   Policy/Abstraction Refinement

We define a kernel state s as being a refinement of the policy p as follows.

pas-refined p s $\equiv$
      policy-wellformed (pasPolicy p) (range (pasIRQAbs p)) (pasSubject p) $\wedge$
      irq-map-wellformed p s $\wedge$
      auth-graph-map (pasObjectAbs p) (state-objs-to-policy s) $\subseteq$ pasPolicy p $\wedge$
      state-asids-to-policy p s $\subseteq$ pasPolicy p $\wedge$
      state-irqs-to-policy p s $\subseteq$ pasPolicy p

The kernel state refines the policy if the various forms of authority contained within it, when labelled by the abstraction functions, are a subset of the policy. The full definitions of the extraction functions for authority from the kernel state are too detailed to describe here. In summary, a subject has authority over an object for one of these reasons:

- it possesses a capability to the object.
- it is a thread which is waiting to conclude a message send. For performance reasons the capability needed to start the send is not rechecked on completion, and thus the thread state is an authority in its own right.
- it possesses the parent capability in the capability derivation tree (cdt) of a capability stored in the object.
- the page tables link the subject to the object.
- the active virtual address space database names the object as the toplevel page table for an address space identifier the subject owns.
- the interrupt despatch mechanism lists the object as the receiver for an interrupt request number the subject owns.

Note that none of this authority extraction is subjective. The pas-refined predicate is subjective only because it also asserts policy-wellformed. The reason for this is convenience: these properties are almost always needed together in the proof.

3.4   Specification of Integrity

We define access control integrity subjectively as follows:

integrity $p\ s\ s' \equiv$

$\quad\quad$ ($\forall x.$ object-integrity p (pasObjectAbs $p\ x$) (kheap $s\ x$) (kheap $s'\ x$)) $\wedge$

$\quad\quad$ ($\forall x.$ memory-integrity $p\ x$ (tcb-states-of-state s) (tcb-states-of-state s')

$\quad\quad\quad\quad$ (auth-ipc-buffers s) (memory-of $s\ x$) (memory-of $s'\ x$)) $\wedge$

$\quad\quad$ ($\forall x.$ cdt-integrity $p\ x$ (cdt $s\ x$, is-original-cap $s\ x$) (cdt $s'\ x$, is-original-cap $s'\ x$))

This says that a transition from s to s' satisfies access control integrity if all kernel objects in the kernel object heap (kheap s), user memory and the cdt (capability derivation tree) were changed in an acceptable manner.

$\quad$ The object level integrity predicate is defined by eight introduction rules. The following three rules are representative:

$$\frac{x = \text{pasSubject } p}{\text{object-integrity } p\ x\ ko\ ko'} \qquad \frac{ko = ko'}{\text{object-integrity } p\ x\ ko\ ko'}$$

$$\frac{\begin{array}{c} ko = \text{Some (TCB } tcb) \qquad ko' = \text{Some (TCB } tcb') \\ \exists\, ctxt'.\ tcb' = tcb(\!|\text{tcb-context} := ctxt', \text{tcb-state} := \text{Running}|\!) \\ \text{receive-blocked-on } ep\ (\text{tcb-state } tcb) \\ (\text{pasSubject } p, \text{Send}, \text{pasObjectAbs } p\ ep) \in \text{pasPolicy } p \end{array}}{\text{object-integrity } p\ x\ ko\ ko'}$$

These cases allow the subject to make any change to itself, to leave anything unchanged, and to send a message through an endpoint it has Send access to and into the registers (called the tcb-context here) of a waiting receiver. Note that it is guaranteed that the receiver's registers are changed only if the message transfer completes and that the receiver's state is changed to Running. It is likewise guaranteed by memory-integrity that a receiver's in-memory message buffer is changed only if the message transfer completes, which is the reason for the complexity of the arguments of the memory-integrity predicate above.

$\quad$ The additional object-integrity cases include a broadly symmetric case for receiving a message and resuming the sender, for resetting an endpoint and evicting a waiting sender or receiver, for updating the list of threads waiting at an endpoint, and for removing a virtual address space the subject owns from the active virtual address space database.

$\quad$ The cdt-integrity predicate limits all cdt changes to the subject's label.

$\quad$ The crucial property about integrity is transitivity:

Lemma 1. integrity $p\ s_0\ s_1 \wedge$ integrity $p\ s_1\ s_2 \longrightarrow$ integrity $p\ s_0\ s_2$

This must be true at the top level for our statement about a single system call to compose over an execution which is a sequence of such calls.

$\quad$ Integrity is also trivially reflexive:

Lemma 2. integrity $p\ s\ s$

3.5 Toplevel Statements

Both integrity and pas-refined should be invariants of the system, which can be demonstrated using Hoare triples in a framework for reasoning about state

monads in Isabelle/HOL. We have previously reported on this framework in depth [5]. In summary, the precondition of a Hoare triple in this framework is a predicate on the pre-state, the post condition is a predicate on the return value of the function and the post-state. In the case below, the return value is *unit* and can be ignored in the post condition. The framework provides a definition, logic, and automation for assembling such triples.

Theorem 1 (Integrity). *The property* integrity *pas st holds after all kernel calls, assuming that the protection state refines the policy, assuming the general system invariants* invs *and* ct-active *(current thread is active) for non-interrupt events, assuming the policy subject is the current thread, and assuming that the kernel state st is the state at the beginning of the kernel call. In Isabelle:*

$\{$pas-refined $pas \cap$ invs $\cap (\lambda s.\ ev \neq$ Interrupt $\longrightarrow$ ct-active $s) \cap$
 is-subject $pas \circ$ cur-thread $\cap (\lambda s.\ s = st)\}$
call-kernel ev
$\{\lambda\text{-}.\ $integrity $pas\ st\}$

We have shown in previous work [12] that the preconditions invs and $ev \neq$ Interrupt $\longrightarrow$ ct-active s hold for any system execution at kernel entry.

Theorem 2 (Authority Confinement). *The property* pas-refined *pas is invariant over kernel calls, assuming again the general system invariants* invs *and* ct-active *for non-interrupt events, and assuming that the current subject of the policy is the current thread.*

$\{$pas-refined $pas \cap$ invs $\cap (\lambda s.\ ev \neq$ Interrupt $\longrightarrow$ ct-active $s) \cap$
 is-subject $pas \circ$ cur-thread$\}$
call-kernel ev
$\{\lambda\text{-}.\ $pas-refined $pas\}$

We discuss the proof of these two theorems in the next section.

Via the refinement proof shown in previous work [12], both of these properties transfer to the C code level of the kernel. The guarantees that integrity *pas st* makes about user memory transfer directly, but the guarantees pas-refined *pas* and integrity *pas st* make about kernel-private state are mapped to their image across the refinement relation, which means we may lose some precision. The protection state of the kernel maps across the refinement relation precisely, the only difference between the model-level and C-level capability types being encoding.

The remainder of the system state does not translate so simply, but we contend that this does not matter. We envision the integrity theorem being useful mainly as a framing rule, with a component programmer appealing to the integrity theorem to exclude interference from other components and to the kernel model to reason about the component's own actions. In this case the programmer is interested not in the precise C state of the private kernel data, but about the related kernel model state. The integrity theorem will then provide exactly what is needed.

For proving confidentiality in the future, we may have to be more careful, because abstraction may hide sources of information that exist in the C system.

4 Proof and Application

4.1 Proof

The bulk of the proof effort was showing two Hoare triples for each kernel function: one to prove pas-refined as a postcondition, and one to prove integrity. These lemmas are convenient to use within the Hoare framework as we can phrase them in a predicate preservation style. In the case of the integrity predicate, we use a form of the Hoare triple which encapsulates transitivity and is easy to compose sequentially. This form is equivalent to the more explicit form as a consequence of reflexivity and transitivity:

$$(\forall st. \{\!| \lambda s.\ \mathsf{integrity}\ pas\ st\ s \land P\ s |\!\}\ f\ \{\!| \lambda rv.\ \mathsf{integrity}\ pas\ st |\!\}) =$$
$$(\forall st. \{\!| \lambda s.\ s = st \land P\ s |\!\}\ f\ \{\!| \lambda rv.\ \mathsf{integrity}\ pas\ st |\!\})$$

The proof was accomplished by working through the kernel's call graph from bottom to top, establishing appropriate preconditions for confinement and integrity for each function. Some appeal was made to previously proven invariants.

The proof effort generally proceeded smoothly because of the strength of the abstraction we are making. We allow the subject to change arbitrarily anything with its label, we map most kinds of authority to the Everything authority, and we require anything to which the subject has Everything access to share the subject's label. These broad brushstroke justifications are easy to apply, and were valid for many code paths of the kernel.

For example, CSpace updates always occur within the subject. As preconditions for various functions such as cap-move and cap-insert we assert that the address of the CSpace node being updated has the subject's label and, for pas-refined preservation, that all authority contained in the new capabilities being added is possessed by the subject in the policy. These preconditions are properties about the static policy, not the dynamic current state, which makes them easy to propagate through the proof.

The concept of a static policy gave us further advantages. By comparison, we had previously attempted two different variations on a proof that seL4 directly refines the take-grant security model [15]. These proof attempts were mired in difficulties, because seL4 execution steps are larger than take-grant steps. In between the take-grant steps of a single seL4 kernel call, their preconditions may be violated because capabilities may have moved or disappeared, and so the steps could not be composed easily. This seemed particularly unfortunate considering that the take-grant authority model has a known static supremum. Comparing against this static graph instead yields something like our current approach.

Another advantage we have in this proof effort is the existing abstraction from the C code up to our kernel model. The cdt (capability derivation tree) and endpoint queues described already must be implemented in C through pointer linked datastructures. In C the cdt is encoded in prefix order as a linked list. When a subject manipulates its own capabilities it may cause updates to pointers in list-adjacent capabilities it does not control. In the abstract kernel model the cdt is represented as a partial map from child to parent nodes, making all of our

subject's CSpace operations local. It would be possible to phrase an integrity predicate on the C level which allowed appropriate pointer updates, but we think it would be extremely difficult to work with.

The combined proof scripts for the two access control properties, including the definitions of all formalisms and the SAC example, comprise 10500 lines of Isabelle/HOL source. The proof was completed over a 4 month period and consumed about 10 person months of effort. Relative to other proofs about the kernel model this was rapid progress. Modifications to this proof have also been fast, with the addition of the cdt-integrity aspect of the integrity property being finished in a single day.

During the proof we did not find any behaviour in seL4 that would break the integrity property nor did we need to change the specification or the code of seL4. We did encounter and clarify known unwanted API complexity which is expressed in our policy wellformedness assumption. One such known problem is that the API optimisation for invoking the equivalent of a remote procedure call in a server thread confers so much authority to the client that they have to reside in the same policy label. This means the optimisation cannot be used between trust boundaries. An alternative design was already scheduled, but is not implemented yet.

We have found a small number of access control violations in seL4's specification, but we found them during a previous proof before this work began. These problems related to capability rights that made little sense, such as read-only Thread capabilities, and had not been well examined. The problem was solved by purging these rights.

4.2 Application

The application scenario of the integrity theorem is a system comprising trusted as well as untrusted components such as in the SAC example. Such scenarios are problematic for purely mandatory access control systems such as subsystems in a take-grant setting [15,7], because the trusted component typically needs to be given too much authority for access control alone to enforce the system's security goals (hence the need for trust). Our formulation of integrity enforcement per subject provides more flexibility. Consider a sample trace of kernel executions on behalf of various components in the SAC.

$$s_0 \text{- -} \boxed{\text{T}} \text{- } s_1 \text{- -} \boxed{\text{CTR}} \text{- } s_2 \text{- -} \boxed{\text{RM}} \text{- } s_3 \text{- -} \boxed{\text{R}} \text{- } s_4 \text{- -} \boxed{\text{T}} \text{- - -}$$

The example policy in Fig. 3 satisfies our wellformedness condition for the components R, T, and CTR. It does not satisfy wellformedness for RM. This means, given the initial state s_0, we can predict bounds for authority and state mutation using our theorems up to s_2 before RM runs. Since RM is the trusted component, we do not apply the integrity theorem, but reason about its (known) behaviour instead and get a precise characterisation of s_3. From here on, the integrity theorem applies again, and so on. Suitably constructed, the bounds will

be low enough to enforce a system wide state invariant over all possible traces which in turn, if chosen appropriately, will imply the security goal.

As hinted at in Sect. 2, at state s_2, the RM component could use its excessive authority to reconfigure the system such that a new R is now connected to networks A and D. This setup would be incompatible with the policy applied to the transitions before, but a new policy reflecting the reconfiguration can be constructed that applies from there on. If that new policy and the reconfiguration are compatible with the system invariant we can conclude the security goal for a system that dynamically changes its high-level policy even if the integrity theorem itself assumes a static policy per kernel event.

For reasoning about such systems, it is convenient to lift wellformedness and therefore pas-refined to sets of allowed subjects instead of a single current actor. This means the same instantiation of the theorem can be applied and chained over traces without further proof. We have formulated and proved the lifted version, but omit the details here. They add no further insight.

The set versions of integrity and pas-refined are also useful in a more restricted, but common scenario where the kernel is employed as a pure separation kernel or hypervisor. In this scenario, all components would be considered untrusted, and there can be one system-wide policy that is wellformed for the set of all components. Wellformedness can be shown once per system and the subjective integrity theorem collapses to a traditional access control formulation of integrity that applies to all subjects.

4.3 Limitations

The limitations of the theorem as presented mostly reflect API complexities that we circumvented by making assumptions on the policy.

The strongest such assumption is that we require all componentes with Grant connections to map to the same label. This is no more than what a traditional take-grant analysis amounts to [7], but in our subjective setting there would be no strong reason to forbid Grant from trusted to untrusted components, i.e. from another subject to the currently running subject if the authority transmitted that way is within policy bounds. Stating the latter turned out to be intricate and difficult, and we have not yet needed this case in practise.

Another limitation is that the theorem provides relatively coarse bounds. This is required to achieve the level of abstraction we are seeking, but it is imaginable that the bounds could be too high for a particular frame condition that is required. In this case, one could always fall back to reasoning about the precise kernel behaviour for the event under consideration, but it was of course the purpose of the theorem to be able to avoid this.

5   Related Work

Security properties have been the goal for OS verification from the beginning: the first projects in this space UCLA Secure Unix [19] and PSOS [8] (Provably

Secure OS) were already aiming at such proofs. For a general overview on OS verification, we refer to Klein [11] and concentrate below on security in particular.

A range of security properties have been proven of OS kernels and their access control systems in the past, such as Guttman et al's [9] work on information flow, or Krohn et al [13] on non-interference. These proofs work on higher-level kernel abstractions. Closest to our level of fidelity comes Richards [17] in his description of the security analysis of the INTEGRITY-178B kernel in ACL2. Even this model is still connected to source code manually.

As mentioned, seL4 implements a variant of the take-grant capability system [15], a key property of which is the transitive, reflexive, and symmetric closure over all Grant connections. This closure provides an authority bound and is invariant over system execution. Similar properties hold for a broader class, such as general object-capability systems [16].

We have previously proved in Isabelle that the take-grant bound holds for an abstraction of the seL4 API [7,4]. The EROS kernel supports a similar model with similar proof [18]. However, these abstractions were simpler than the one presented here and not formally connected to code.

Compared to pure take-grant, our subjective formulation of integrity is finer-grained: the policy is transitive and reflexive over Everything edges, but not necessarily symmetric, and Grant does not always need to be considered Everything.

6   Conclusions

In this paper, we have presented the first formal proof of integrity enforcement and authority confinement for a general-purpose operating system implementation. These properties together provide a powerful framing condition for the verification of whole systems constructed on top of seL4, which we have argued with reference to a case study on a real example system.

We have shown that the real-life complexity in reasoning about a general-purpose kernel implementation can be managed using abstraction. In particular, our formalisation avoids direct reasoning about the protection state, which can change over time, by representing it via a separate policy abstraction that is constant across system calls. Integrity asserts that state mutations must be permitted by this policy, while authority confinement asserts that the protection state cannot evolve to contradict the policy.

This work clearly demonstrates that proving high-level security properties of real kernel implementations, and the systems they host, is now a reality. We should demand nothing less for security-critical applications in the future.

References

1. J. Andronick, T. Bourke, P. Derrin, K. Elphinstone, D. Greenaway, G. Klein, R. Kolanski, T. Sewell, and S. Winwood. Abstract formal specification of the seL4/ARMv6 API. http://ertos.nicta.com.au/software/seL4/, Jan 2011.
2. J. Andronick, D. Greenaway, and K. Elphinstone. Towards proving security in the presence of large untrusted components. In G. Klein, R. Huuck, and B. Schlich, editors, *5th SSV*, Vancouver, Canada, Oct 2010. USENIX.
3. D. Bell and L. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report MTR-2997, MITRE Corp., Mar 1976.
4. A. Boyton. A verified shared capability model. In G. Klein, R. Huuck, and B. Schlich, editors, *4th SSV*, volume 254 of *ENTCS*, pages 25–44. Elsevier, 2009.
5. D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *21st TPHOLs*, volume 5170 of *LNCS*, pages 167–182, Montreal, Canada, Aug 2008. Springer-Verlag.
6. J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9:143–155, 1966.
7. D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. In J. Woodcock and N. Shankar, editors, *VSTTE 2008*, volume 5295 of *LNCS*, pages 99–114, Toronto, Canada, Oct 2008. Springer-Verlag.
8. R. J. Feiertag and P. G. Neumann. The foundations of a provably secure operating system (PSOS). In *AFIPS Conf. Proc., 1979 National Comp. Conf.*, pages 329–334, New York, NY, USA, Jun 1979.
9. J. Guttman, A. Herzog, J. Ramsdell, and C. Skorupka. Verifying information flow goals in security-enhanced linux. *J. Comp. Security*, 13:115–134, Jan 2005.
10. T. Jaeger. *Operating System Security*. Morgan & Claypool Publishers, 2008.
11. G. Klein. Operating system verification—an overview. *Sādhanā*, 34(1):27–69, 2009.
12. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.
13. M. Krohn and E. Tromer. Noninterference for a practical DIFC-based operating system. In *IEEE Symp. Security & Privacy*, pages 61–76, 2009.
14. B. W. Lampson. Protection. In *5th Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, Mar 1971. Reprinted in ACM Operat. Syst. Rev., **8**(1), Jan 1974, pp 18–24.
15. R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *J. ACM*, 24(3):455–464, 1977.
16. T. Murray and G. Lowe. Analysing the information flow properties of object-capability patterns. In *6th FAST*, volume 5983 of *LNCS*, pages 81–95, 2010.
17. R. J. Richards. Modeling and security analysis of a commercial real-time operating system kernel. In D. S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 301–322. Springer-Verlag, 2010.
18. J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *IEEE Symp. Security & Privacy*, pages 166–181, Washington, DC, USA, May 2000.
19. B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *CACM*, 23(2):118–131, 1980.