seL4: from General Purpose to a Proof of Information Flow Enforcement

Toby Murray*†, Daniel Matichuk*, Matthew Brassil*, Peter Gammie*, Timothy Bourke*,
Sean Seefried*, Corey Lewis*, Xin Gao* and Gerwin Klein*†
*NICTA and †University of New South Wales, Sydney, Australia
Email: firstname.lastname@nicta.com.au

***Abstract*—In contrast to testing, mathematical reasoning and formal verification can show the absence of whole classes of security vulnerabilities. We present the, to our knowledge, first complete, formal, machine-checked verification of information flow security for the implementation of a general-purpose microkernel; namely seL4. Unlike previous proofs of information flow security for operating system kernels, ours applies to the actual 8,830 lines of C code that implement seL4, and so rules out the possibility of invalidation by implementation errors in this code. We assume correctness of compiler, assembly code, hardware, and boot code. We prove everything else. This proof is strong evidence of seL4's utility as a separation kernel, and describes precisely how the general purpose kernel should be configured to enforce isolation and mandatory information flow control. We describe the information flow security statement we proved (a variant of intransitive noninterference), including the assumptions on which it rests, as well as the modifications that had to be made to seL4 to ensure it was enforced. We discuss the practical limitations and implications of this result, including covert channels not covered by the formal proof.**

I. Introduction

Provably secure operating systems have been a research topic for more than 30 years [11], [17], [47], [53], and while there are a number of small high-assurance separation kernels [25] in deployment such as INTEGRITY-178B [44], even the strongest levels of security evaluation schemes such as Common Criteria do not demand implementation-level proofs, which have been widely thought infeasible. Modern mainstream OSes are riddled with security problems as is to be expected for large monolithic systems, and even implementations of the proposed mainstream fix for this problem, *hypervisors* like Xen [7], have been shown to exhibit a number of critical vulnerabilities [39].

This paper presents the first formal, fully machine-checked, mathematical proof that a high-performance, general-purpose microkernel *C code implementation* enforces strong information flow control. Together with the existing proofs of seL4's functional correctness [27] and integrity enforcement [51], this work shows that seL4 provably enforces strong access control mechanisms, in particular the high-level security properties of confidentiality and integrity. Our proof assumptions explicitly state how to configure this general-purpose OS kernel to enforce isolated partitions with controlled communication channels. Unlike previous information flow verifications ours applies to the actual 8,830 lines of C code that implement seL4, rather than to a manually abstracted model of its behaviour. Our main proof assumptions stem from the foundational work on functional correctness [27]. We assume correctness of compiler, assembly code, hardware, and boot code; we prove everything else. The verified seL4 kernel runs on commodity ARMv6 and ARMv7 hardware and is available commercially under the product name OKL4:verified, as well as freely in binary form for academic use [41].

The information flow property we prove for seL4 is a variant of intransitive noninterference [19], [40], [46] proposed for OS kernels [36]. This property enforces confidentiality on storage channels and logical time. As is usual for noninterference results, fine-grained timing behaviour of the hardware is not covered by the formal baseline model and therefore neither by the proof. This means covert timing channels still need to be mitigated by complementary (e.g. probability-based) techniques. We argue that for modern commodity hardware the absence of such channels is not fully enforceable and that the best that can be expected is to reduce channel bandwidth based on the risk profile of a particular deployment.

We analyse the limitations and strength of the proof statement in detail in Section V. By formally stating the assumptions under which seL4 enforces information flow security, this result also provides an unambiguous description of how seL4 should be configured to enforce a particular information flow policy. The strongest restriction on such configurations is the absence of direct memory access (DMA) for devices. This is not an unusual restriction, and is shared with commercial separation kernels in deployment. New hardware mechanisms such as IOMMUs [3] may enable us to relax this in the future.

In detail, the technical contributions of this work are:

- to our knowledge, the most detailed and extensive machine-checked formal verification of information flow security ever for a general-purpose OS kernel;
- a formal, and thus precise and unambiguous, description on how to configure the general-purpose seL4 microkernel to enforce a given information flow policy;
- achieving the above results without sacrificing performance or preventing the use of the dynamic general-purpose microkernel API inside partitions.

To achieve this result, we extended the seL4 API only minimally by adding a static partition-based scheduler com-

1081-6011/13 $26.00 © 2013 IEEE
DOI 10.1109/SP.2013.35

mon to separation kernels. We disallow none of the seL4 API other than requiring that seL4 be configured to prevent asynchronous interrupt delivery to user-space partitions which would introduce an information channel. This means that device drivers must poll for device interrupts via memory mapped IO, which is typical behaviour for separation kernels [42] and high-assurance systems.

While traditional separation kernels typically provide no system calls at all after initial configuration, in our general-purpose seL4 setting, all other kernel facilities are available within partitions, including dynamic memory allocation and revocation, (intra-partition) inter-thread messaging, capability transfer and shared memory. Between partitions, seL4 provides asynchronous notifications and shared memory to facilitate uni-directional communication. The information flow security theorem implies that these primitives are free of storage back-channels, and that the dynamic intra-partition services do not violate information flow security either. A modified version of the paravirtualised Linux system Wombat [29] allows an entire Linux instance and its applications to run within a partition. This enables mandatory information flow control policies to be enforced between untrusted and legacy applications, with guarantees on information flow provided by a strong machine-checked theorem.

During this proof of noninterference we did not find any information-flow problems in the seL4 kernel that required code changes, as we had hoped given the previous intensive work on proving functional correctness and integrity. We did, however, find a number of API features that had to be explicitly forbidden for inter-partition use. For experts, most of these were not surprising. For instance it is well-known that synchronous IPC will introduce an information flow back-channel, but the proof clearly identified all instances. This included a number of cases that were not immediately obvious, such as capability deletion potentially and observably crossing partition boundaries. All of these could be excluded by reasonable restrictions on supported seL4 configurations, detailed in Section IV and summarised in Section V.

Our security proof—like all others—is not an iron-clad statement of general security for seL4, but rather a powerful piece of evidence about seL4's security mechanisms and its suitability as a separation kernel. Importantly, any system-level security evaluation can now concentrate its effort on validating proof assumptions, which is a much simpler job than asserting information flow security, and on using complementary techniques for the remaining items that are not covered by our proof (detailed in Section V). Furthermore, the proof statement addresses whole high-level classes of attack such as information leakage to confined subjects without access to timing sources, and the evaluation can be guided by precise formal statements of policy and system deployment assumptions. The security proof gives precise predictions about the strength of mechanisms; the previous functional correctness proof shows that these mechanisms will be reliably enforced by the implementation.

The remainder of this paper is organised as follows. We briefly summarise the attacker/threat model in Section II. We then discuss background material in Section III, including the seL4 API and how it can be used to implement separation and information flow control, as well as past formal seL4 verification work that our proof builds upon. We present our formal statement of information flow security and discuss its proof in Section IV. We consider its implications and limitations in Section V. Section VI situates our result in the context of related work before we conclude.

II. Threat Model

Our target deployment scenario is a physically secured, high-assurance, uni-processor system with trusted hardware. The system may potentially have access to untrusted networks and run mutually distrusting application software. The security goal for the kernel is to only permit information flows according to the partitions and information flow policy it was configured with.

The threat model assumes that all user-level code after system initialisation is malicious and acting in concert to break the information flow policy. The attacker's goal is to read or indirectly infer the contents (i.e. private state) of a partition that according to the information flow policy should remain secret to it. The attacker is assumed to know the configuration and code of the entire system, including the kernel, but not the contents of other partitions.

As mentioned, timing channels are not in the scope of this proof and must be analysed by complementary techniques. The channels that are in scope are storage channels and causal deduction. The attacker may run any sequence of instructions or attempt any kernel call, to break the information flow policy. Our proof statement is that, subject to the limitations and assumptions discussed in Section V, the attacker will not succeed.

III. Background

A. The seL4 Kernel API

The seL4 microkernel provides a minimal set of mechanisms for implementing secure systems: threads, capability management, virtual address spaces, inter-process communication (IPC), and interrupt delivery. The state of each instance of a service is maintained within the kernel by data structures termed *kernel objects*. For example, for each thread in a system there is a *thread object* that stores the information about the thread relevant to scheduling, execution, and access control. User-space programs can only refer to kernel objects indirectly through *capabilities* [16], each of which combines a reference to a specific object with a set of access rights. For example, a thread cannot start, stop, or read or modify the registers of another unless it possesses a capability for the corresponding thread object.

Capabilities are managed by the kernel, and stored in kernel objects called *CNodes*. Each thread object also contains several distinguished capabilities. One of these defines the root of a thread's *CSpace*: a directed graph of the CNodes the thread can access. A user-space program refers to capabilities in its CSpace via addresses that specify paths relative to its CSpace root. It is possible to fabricate an address but never a capability; invalid addresses are simply rejected during decoding. We say that a thread *invokes* a capability when it passes the address of that capability to a system call. For instance, a CNode capability can be invoked to copy, move, delete, or *derive* (reduce the associated access rights of) the capabilities contained within it.

Besides access control, capabilities also serve to manage memory. Unallocated memory is made available through *Untyped memory* capabilities that can be *retyped* into kernel objects like CNodes or thread objects. Retyping creates fresh capabilities that refer to the newly created objects, and which are *derived* from the original Untyped capability. The *revoke* system call deletes all capabilities derived from the invoked Untyped capability, which effectively reclaims all resources allocated from within it and is a useful way of destroying an entire subsystem in a single system call.

Untyped memory can also be allocated to kernel objects representing page directories, page tables, and pages. Another of the distinguished capabilities stored in each thread object refers to the page directory at the root of the associated thread's virtual address space (VSpace). By invoking this capability, and those for the associated page tables and pages, a thread can direct the kernel to configure the memory management unit of the underlying platform. In particular, multiple threads that each possess capabilities to the same physical pages may each map those pages into their respective VSpaces and thereafter communicate by reading and writing to the shared memory.

Threads can also communicate and synchronise by sending messages through IPC *endpoints*. There are two types of endpoint: synchronous, where a sender blocks waiting for a receiver; and asynchronous, where a sender never blocks. Capabilities to either kind of endpoint may carry the Send and Receive access rights; a thread with a Send capability to an endpoint can use it to transmit messages to any thread that has a Receive capability to the same endpoint. In addition, capabilities for synchronous endpoints may also carry the Grant right that allows senders to transmit capabilities to receivers in the style of classical take-grant systems [30].

Two special capability types are used to control the kernel's interrupt delivery mechanism. An InterruptControl capability confers the authority to create new InterruptHandler capabilities for specific interrupt request (IRQ) numbers, each of which confers the authority to receive an IRQ.

Interrupts are disabled in kernel mode to avoid in-kernel concurrency. Low interrupt latencies are ensured by adding *preemption points* to certain long-running system calls, such as revoke which iteratively deletes derived capabilities. Interrupts are polled at these points and, if necessary, an active system call may be suspended until the requesting thread is next scheduled to run.

B. Using seL4 as a Separation Kernel

As a microkernel, seL4 is designed to provide mechanisms only; policy is implemented in user-space. On startup, the kernel hands control to the initial user-level thread, which has complete authority, and whose job it is to configure the system including implementing any security policy that is to be enforced by carefully distributing subsets of its authority to the entities that it sets up.

This means that, when used as a separation kernel, the initial thread is responsible for creating each of the partitions and any communication channels between them. The initial thread is then responsible for destroying itself, and once this is completed correctly the system is configured to enforce separation. It is from this state, after this configuration has occurred, that our proof of information flow security applies.

Each partition spans both user-space and kernel-space: it contains not just user memory mapped into the VSpaces of the threads within the partition, but may also include kernel objects that those threads have capabilities for. Concretely, each partition will typically contain a number of thread objects with associated page directories, page tables and pages to implement their VSpaces, as well as a number of CNode objects to implement their CSpaces, and any other kernel objects used within the partition such as endpoints for intra-partition messaging and capability transmission. The partition may also contain Untyped Memory capabilities to allow new objects to be created and intra-partition subsystems to be destroyed. All kernel services, other than interrupt delivery (see below), are available to partitions. This allows partitions to host large and complex applications, not least a paravirtualised version of Linux to host legacy applications.

As mentioned in Section I, shared memory and asynchronous endpoints allow uni-directional communication between partitions. These facilities must be set up by the initial thread when partitions are created, which involves distributing the necessary capabilities for these facilities to each partition that is going to use them. This is because capabilities cannot be transferred between partitions without breaking *authority confinement* [51], which is a necessary condition for our information flow security property.

Figure 1 depicts a small example system with two partitions, that communicate via shared memory and asynchronous endpoints. Partition 2 on the right has read access to a shared page in Partition 1 on the left, and Partition 1 has send rights to an asynchronous endpoint in Partition 2. This allows information to flow only from Partition 1 to Partition 2, in accordance with our information flow theorem.

To implement separation in seL4, we had to extend its existing priority-based scheduler to implement *partition*

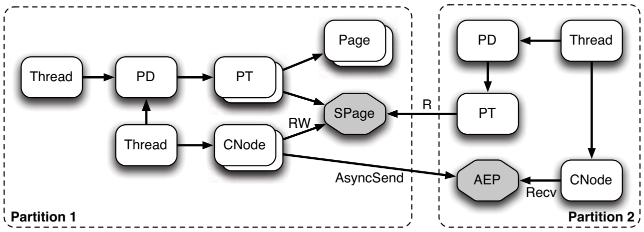


Figure 1. A small example system, with two partitions.

scheduling. Here, the scheduler follows a static round-robin schedule between partitions, with fixed-length time slices per partition, while doing dynamic priority-based round-robin scheduling of threads within each partition. This ensures that the choice of which partition is currently running depends only on this fixed schedule, while the choice about which thread is running in the current partition depends only on the internal state of the current partition. Thus scheduler choices do not leak information between partitions.

A limitation of the current implementation that we discuss further in Section V is that partitions can overrun their time-slices by performing system calls just before a partition switch would have occurred. This happens because interrupts are disabled in kernel mode, which prevents a partition switch from being serviced until after the system call has been handled or a preemption point has been reached. While there exist obvious solutions to this problem, deciding on the most appropriate fix necessarily involves a trade-off between performance and timing channel bandwidth which can only be properly decided within the context of a specific deployment scenario. For this reason, we have currently left a general treatment of this limitation as future work. This channel can trivially be drastically reduced by configuring seL4 to not permit any intra-partition system calls at all, as in a traditional separation kernel; a classic "yellow-light" mechanism, whereby kernel services are denied near the end of a partition's timeslice, based on a sound upper bound for the worst-case execution time of the kernel [12] in this configuration could fully eliminate this channel.

Another limitation mentioned earlier in Section I is that the initial thread must ensure that the kernel APIs for interrupt delivery are not exposed to any partition. It does so trivially by never giving any partition an InterruptControl or InterruptHandler capability, and never using any such capabilities that it might possess. As explained earlier, this forces partitions to interact with devices solely through memory-mapped IO.

C. Functional Correctness Proof

Our information flow security proof builds upon two earlier verification results for the seL4 microkernel. The first is a proof of *functional correctness* [27], reported by Klein et al. in 2009. The second is a proof of integrity enforcement for seL4 [51], completed by Sewell et al. in 2011.

The functional correctness proof for seL4 is a classical proof of *refinement* [1], [15], showing that seL4's C implementation refines (or implements) an abstract specification of its functional behaviour, which we call simply the *abstract specification*. Each level is formalised as a state machine whose transitions include processing an interrupt or exception, performing a system call, and ordinary user-level operations like reading and writing user-accessible memory.

For an automaton A and initial *observable* state s and sequence of transitions as, let $\textsf{execution}\ A\ s\ as$ denote the set of observable states that can be reached by A performing as. Then, an automaton C refines A, written $A \sqsubseteq C$, when C's behaviours are a subset of A's.

$$A \sqsubseteq C \equiv \forall s\ as.\ \textsf{execution}\ C\ s\ as \subseteq \textsf{execution}\ A\ s\ as$$

This proof took around 25 person-years to complete [27]. Much of that effort was devoted to proving invariants, which the kernel maintains and which are necessary preconditions in the refinement proof. We make direct use of these invariants when reasoning about information flow in seL4.

Importantly, the functional correctness proof enabled us to perform our information flow security proof over the abstract specification, which is far simpler to reason about than the kernel's C code. The formulation of information flow security that we adopt is preserved by refinement [36]. This means that once proved for the abstract specification, we can compose this result with the refinement theorem to derive information flow control for the kernel's C code.

By proving information flow security over the abstract specification, we gain a substantial saving in effort (see Section V-D) that we estimate is about an order of magnitude.

D. Integrity and Authority Confinement

Sewell et al.'s proof of integrity and authority confinement for seL4 [51], also over the abstract specification, provides the second foundation for our proof of information flow. Roughly, the integrity result says that all changes to the system state are authorised by the capabilities of the currently running thread. Authority confinement says that, in suitable system configurations, no thread's authority will increase.

A more formal account of both properties requires 1) an *access control policy* that captures the subjects in a system, and the authorities that each has to each of the others, 2) an *abstraction function* that partitions kernel objects and memory locations between subjects in an access control policy, and, 3) the *wellformedness constraints* necessary to ensure that the authority of subjects cannot increase. We will now describe each element in more detail.

An access control policy is essentially a directed graph with *subject labels* on nodes and *authority types* on edges. For example, the policy in Figure 2, derived from the system depicted in Figure 1, shows subjects labelled 'S1' and 'S2', where S1 has Read and Write authority to itself and

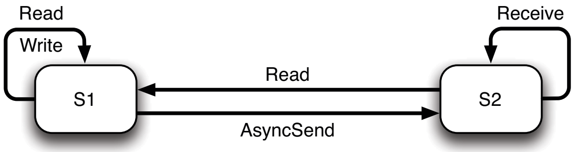


Figure 2. Example access control policy.

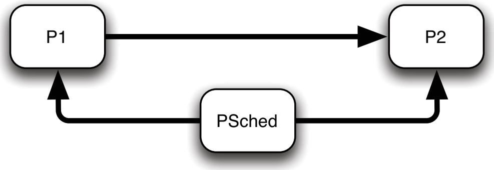


Figure 3. Example information flow policy.

AsyncSend authority to S2, and S2 has Receive authority to itself and Read authority to S1; other self-edges are implicit. The other possible authorities are Grant, SyncSend, Reset, and Control. SyncSend and AsyncSend represent the authority to send on synchronous and asynchronous endpoints respectively. Reset represents the authority to reset an object to its original state. Control authority implies complete control over a target subject; it exists, for instance, when one subject has a thread capability to another, allowing it to overwrite the other's registers.

An abstraction function maps each kernel object and memory location to a subject label, partitioning all system resources between access control subjects. Integrity and authority confinement are formulated over a triple *pas* that contains an access control Policy, an Abstraction function, and the label of the currently active Subject. We often refer to this triple as simply an *access control policy*.

Given a specific *pas* and a state *s*, pas-refined *pas s* states that *pas* both conservatively over-approximates the authority distribution of *s* and that it meets certain wellformedness constraints [51]. The wellformedness constraints include standard requirements like assuming full reflexive authority and the absence of Grant authority between distinct subjects, which could trivially allow a subject's authority to exceed that prescribed by *pas*.

Given these elements, the proof of authority confinement shows that for all access control policies *pas* and states *s* that satisfy the invariants of the abstract specification, if pas-refined *pas s* holds, then for all states s' directly reachable from *s*, pas-refined *pas* s' must also hold. In other words, *pas* is an upper bound on authority within a system.

The integrity property, on the other hand, is captured by the predicate integrity *pas s* s' between any pair of states *s* and s'. It shows that any modification that the current subject can perform is permitted by the authority represented in *pas*, thus giving a bound on the differences between *s* and s'. The main integrity theorem states that for all states *s* that satisfy the invariants, if pas-refined *pas s* holds, then for all directly reachable states s', integrity *pas s* s' also holds. In terms of the example of Figure 2, integrity says that whenever the subject S2 executes, including during system calls it might make, nothing in subject S1 changes, because S2 has only Read authority to S1.

Combining the integrity and authority confinement theorems allows the conclusion that integrity is preserved across all sequences of transitions (for wellformed access control policies). This is important, because we use integrity to help prove information flow security for seL4.

IV. Information Flow Control for seL4

In this section, we describe how information flow control was formalised for seL4 and discuss its formal proof.

A. *Information Flow Policy*

Our formulation of information flow security builds on the integrity and authority confinement proofs for seL4. We begin by mapping an access control policy to a corresponding information flow policy, which allows us to re-use the integrity and authority confinement results to help us prove information flow security.

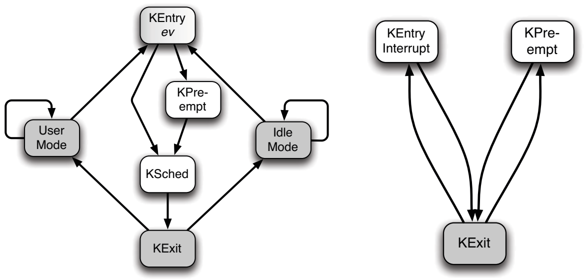
Each access control subject *Si* induces a corresponding information flow partition *Pi*. We also include a partition PSched for the scheduler, as required by our formulation of information flow security [36]. The information flow policy $\leadsto$ is computed in two steps as follows.

We first compute the *extent* of each partition *Pi* that is not PSched. The extent of *Pi* is simply the set of all access control subjects that *Pi* can directly infer information about. This includes those that *Pi* can read directly as well as those that the kernel reads during a system call and then reveals to *Pi* in the results or effects of the system call. This computation is a function of just the access control policy, and is governed by a set of simple rules. For instance, subject *Si* is necessarily in the extent of partition *Pi*; if subject *Si* has Read authority to a subject *Sj* then *Sj* is in the extent of partition *Pi* etc. Referring to Figure 2, the extent of the partition P1 induced by subject S1 is simply the subject S1; the extent of the partition P2 induced by subject S2 is both subjects S1 and S2 because S2 has Read authority to S1.

Having computed the extent of each partition *Pi*, the information flow policy $\leadsto$ is computed using the following two rules. Here *Pi* and *Pj* are non-PSched partitions.

1) $Pi \leadsto Pj$ if the access control policy allows *Si* to affect any subject in *Pj*'s extent.
2) PSched $\leadsto$ *Pi* for all *Pi*, and PSched $\leadsto$ PSched.

The calculation of whether *Si* is allowed to affect some subject *Sk* according to the access control policy is derived from the integrity theorem: *Si* can affect *Sk* according to policy *pas* whose current subject is *Si*, when there exist states



(a) Small-step transition system.

(b) Big-step system.

Figure 4. Transition systems for the seL4 abstract specification. Shading indicates whether the scheduling partition PSched is active (unshaded) or not (shaded). For KEntry *ev*, PSched is active iff $ev =$ Interrupt.

s and s' that differ only for some object in Sk for which pas-refined $pas\ s$ holds and integrity $pas\ s\ s'$ holds.

It is trivial to prove that, under these rules, $\leadsto$ allows no partition other than PSched to send information to PSched; that $\leadsto$ is reflexive; and that PSched can send information to all partitions. This last result is expected since the scheduler can necessarily affect any other partition by scheduling a thread within it. The first result ensures that the scheduler cannot then become a global transitive channel through which information can flow from any partition to any other.

Figure 3 depicts the information flow policy, excluding self-edges, derived from Figure 2's access control policy.

B. System Model

Before describing how we formalise information flow security for seL4, we first describe the formal model of the kernel over which this property is defined. This model is essentially a state machine with unlabelled transitions.

Figure 4(a) depicts the transition system of the seL4 abstract specification for which we proved information flow security, comprising 4,970 source lines of Isabelle/HOL. The transitions do not carry labels; rather, all information about each state transition is encoded in the pre- and post-states of the transition. One piece of information encoded in the state is the current abstract *mode* of execution, which labels the nodes in Figure 4(a). The KEntry mode is parameterised by a kernel event *ev*, indicating the reason for kernel entry, whose values include Interrupt, to model the arrival of device interrupts; SyscallEvent, to model the occurrence of user traps; and others that represent virtual memory faults and exceptions etc. The abstract modes that model kernel-mode execution begin with "K". The transition from KEntry to KPreempt models the kernel handling a system call, reaching a preemption point and preemption occurring because an interrupt has arrived. The following transition models the kernel handling the preemption—i.e. responding to the just-arrived interrupt. The transition from KEntry to KSched models the kernel handling an event without preemption occurring.

The transition from KSched models the execution of the scheduler, which is invoked at the end of every kernel event before returning to user-space. When invoked, the scheduler examines the remaining time slice of the current partition. When the remaining time slice is zero the scheduler switches to the next partition in the static partition schedule; when it is non-zero the scheduler schedules the highest priority runnable thread in the current partition or the idle thread if there are no runnable threads to choose.

The current partition's remaining time slice is decremented upon the arrival of timer interrupts, which are seen only by the kernel. Recall, from Section III-B, that we assume the system is initialised so that all other interrupts are disabled. This is formalised by the property only-timer-irq. It is relatively straightforward to prove that only-timer-irq is invariant across all executions. So the arrival of each interrupt marks the passage of another timer tick.

The transitions from KExit model exiting the kernel back to user-mode. We distinguish the cases in which a user-level thread is running (UserMode) from those in which the idle thread is running (IdleMode), because in the latter the only way to enter the kernel is by the arrival of a device interrupt (KEntry Interrupt).

The self-loop transition on UserMode models the ordinary user-level actions of the currently running thread, for instance reading and writing to the physical memory that is mapped in its virtual address space. The self-loop transition on IdleMode represents the passage of time as the idle thread awaits the arrival of the next interrupt.

The transition system in Figure 4(a) differs a little from the transition system of the kernel about which functional correctness was originally proved [27]. The original transition system did not have the states KPreempt, KSched and KExit and had just single transitions from kernel entry to user-mode and to idle-mode respectively. We modified it (and then re-established functional correctness) to break these kernel transitions into a number of smaller steps because the actions of handling timer interrupts and scheduling a new partition occur on behalf of the scheduling partition, PSched, which is distinct from all other partitions *Pi*. These must therefore be represented as separate transitions.

The machine model on which the kernel specifications sit uses an *interrupt oracle* to model the arrival of device interrupts. This oracle is simply an infinite stream of values in the range 0x00–0xFF (where 0xFF means no interrupt is currently active), with a natural number variable that records the current position in this stream. The machine function getActiveIRQ returns the currently active IRQ. When called, it advances the current position of the interrupt oracle and then examines the value at the new position in the interrupt stream. getActiveIRQ returns this value if it is not 0xFF and is allowed by the current interrupt masks; otherwise getActiveIRQ returns a null value. Under the assumption only-timer-irq, which implies that all non-timer interrupts are masked off, getActiveIRQ can return only one of two possible answers each time it is called: either a timer interrupt has arrived, or no interrupt has arrived since getActiveIRQ was last called. We prove information flow security for all such oracles, and assume that partitions are allowed to know the position of all timer interrupts in the oracle stream as well as the current oracle position. This implies that all partitions are allowed to observe the passage of global time.

To formalise information flow, we had to transform the transition system of Figure 4(a) into one with larger execution steps. While we do not model the passage of time explicitly, it is exposed (albeit quite coarsely) in our formal model of the kernel via the current position of the interrupt oracle. Advancing this position, by calling getActiveIRQ, allows the passage of time to be observed. The amount that the oracle position is advanced when handling a pre-emptible system call necessarily depends on how many preemption points were traversed, which depends on the input data to the system call. Thus the change in the oracle position on any individual transition of Figure 4(a) could leak information if observed by another partition.

Fortunately, in reality, no other partition can observe anything about the new interrupt state until it next executes—i.e. until after the next partition switch. We transform the transition system of Figure 4(a) to reflect this reality, arriving at the transition system depicted in Figure 4(b). This transformation coalesces together all transitions from one KExit until the next (timer) interrupt is processed, in which case the system's new mode is either KPreempt (if the interrupt arrived during a preemptible system call) or KEntry Interrupt (otherwise). Because only the timer interrupt is enabled these transitions represent all activity by the currently running partition in between one timer tick and the next, and show up in the final transition system as a single transition from KExit to KPreempt or KEntry Interrupt respectively. We also coalesce all transitions from KPreempt to the next KExit and similarly for KEntry Interrupt. These transitions represent activity by the scheduling partition PSched.

The amount that the interrupt oracle position is advanced on any transition of this new transition system depends only on its previous position and the position of the next timer interrupt: on a transition from KExit it is advanced to the position of the next timer interrupt in the stream; on a transition to KExit it is advanced once only, during servicing of the just-arrived interrupt.

C. Formalising Information Flow Security

With the system model as depicted in Figure 4(b) we may now formalise information flow security, given an information flow policy $\rightsquigarrow$ derived as explained in Section IV-A.

We adopt a variation of intransitive noninterference proposed in earlier work [36] for operating system kernels. An intransitive noninterference variant is most appropriate here because the information flow policy $\rightsquigarrow$ may, in general, be *intransitive*, in that it may allow information flows from Pi to Pj and Pj to Pk, without allowing a direct flow from Pi to Pk. Crucially, however, our definition admits systems in which the association between each transition and the partition on whose behalf the transition is said to occur depends on the pre-state of the transition. This is required since, as in other operating system kernels, when an event like a system call happens in seL4, the kernel must consult the scheduling data structures to determine which partition is currently active in order to decide which partition the system call has been made by. The mapping from transitions to partitions is thus state-dependent.

Our definition of information flow security is also preserved by refinement [36], which is vital in allowing us to prove it about seL4's abstract specification and then conclude that it must hold for seL4's C implementation by virtue of the functional correctness result.

Our definition of information flow security is a descendant of von Oheimb's notion of *nonleakage* [40]. This condition forbids partitions from being able to learn about the *contents* of others but, unlike traditional *purge*-based noninterference definitions [19], [46] and more recent improvements thereof [34], it does not prevent them from learning about the occurrence of *transitions* of others. Before defining our condition formally it is worth explaining why it is most appropriate here, over purge-based formulations.

Recall, from Section III-B, that seL4 schedules partitions in accordance with a fixed, pre-determined schedule. At any point in time, therefore, the scheduler always knows exactly which partitions have executed in which order. The system model, depicted in Figure 4(b), over which our information flow security proof is conducted has two kinds of transitions: those leading to KExit that model actions of the scheduling partition PSched, and the others that model actions of the ordinary partitions Pi. These latter model the entire execution of a partition from one timer tick to the next. The static schedule pre-determines exactly how many timer ticks each partition should run for before advancing to the next position in the schedule. Therefore whenever it executes the scheduler partition PSched knows the exact number of (unlabelled) transitions each partition has performed so far.

Recall also that the information flow policy $\rightsquigarrow$, derived from the system's access control policy, allows no other partition to send information to the scheduler partition PSched. Because we cannot prevent PSched from learning the number of transitions that have occurred so far, it does not make sense to adopt a purge-based noninterference condition that would forbid partitions learning about the occurrence of others' transitions. Instead, it is more appropriate to require that partitions be unable to learn about the contents of others. In our model, with unlabelled transitions, the information about each transition is encoded in the partition-contents. Thus such a condition still prevents partitions from learning the nature of others' transitions; it allows otherwise isolated partitions to learn only the number of transitions each other performs, as pre-determined by the static schedule.

Such noninterference variants have been well explored in the context of programming language security [48], for instance to assert that the final contents of low-classification variables should never depend on the initial contents of high-classification ones. Our condition [36], called *nonleakage*, is an extension of the original nonleakage formulation of von Oheimb [40].

Nonleakage is defined formally as follows. Let the initial state of the system be s_0. Let $\mathsf{reachable}\ s$ denote when state s is reachable from s_0 by zero or more transitions.

For two states s and t and a partition Pi, let $s \overset{Pi}{\sim} t$ denote when, for each entity e in Pi's extent, e's state is identical in s and t. $s \overset{\mathsf{PSched}}{\sim} t$ when s and t agree on the scheduler's private state, which includes which domain is currently running and how many timer ticks it has left on its timeslice. For a set of partitions P, let $s \overset{P}{\approx} t$ mean $\forall p{\in}P.\ s \overset{p}{\sim} t$.

Let Step denote the step-relation on states of the unlabelled transition system depicted in Figure 4(b): $(s,\ s') \in \mathsf{Step}$ when the system can transition from state s to state s'.

Let $\mathsf{part}\ s$ denote the currently running partition in state s, determined by examining the private state of the scheduler. It is easily shown that $s \overset{\mathsf{PSched}}{\sim} t \longrightarrow \mathsf{part}\ s = \mathsf{part}\ t$.

Then the function $\mathsf{sources}$ is used to calculate the partitions that are permitted to send information to a specific partition p when a sequence of n (unlabelled) transitions occurs from a state s; this is the set $\mathsf{sources}\ n\ s\ p$.

$$
\begin{aligned}
&\mathsf{sources}\ \mathit{0}\ s\ p && = \{p\} \\
&\mathsf{sources}\ (n + \mathit{1})\ s\ p && = \\
&\quad \bigcap \{\mathsf{sources}\ n\ s'\ p \mid (s,\ s') \in \mathsf{Step}\} \cup \\
&\quad \{w \mid w = \mathsf{part}\ s\ \wedge \\
&\qquad\quad (\forall s'.\ (s,\ s') \in \mathsf{Step} \longrightarrow \\
&\qquad\qquad\quad (\exists v.\ \mathsf{part}\ s \rightsquigarrow v \wedge v \in \mathsf{sources}\ n\ s'\ p))\}
\end{aligned}
$$

The first equation says that partition p is always permitted to send information to itself. The second says that, when one or more transitions occur from state s, firstly: any partition is permitted to send information to p that is always permitted to do so after the first transition has occurred; and secondly: that the current partition is permitted to send information to p if it is always permitted to send information to a partition who is subsequently permitted to send to p.

Nonleakage asserts that the only partitions that can influence the contents of an arbitrary partition p after n transitions have occurred from an arbitrary reachable state s are PSched and those in $\mathsf{sources}\ n\ s\ p$. This condition is phrased by considering the counterfactual case in which the state s is modified to produce a new reachable state t such that $s \overset{\mathsf{sources}\ n\ s\ p}{\approx} t$ and $s \overset{\mathsf{PSched}}{\sim} t$, and then asserting that $s' \overset{p}{\sim} t'$ for all states s' and t' reached after performing n transitions from s and t respectively, abbreviated $s \overset{p}{\sim}_n t$.

$$
\begin{aligned}
&\mathsf{nonleakage} \equiv \forall n\ s\ t\ p.\ \mathsf{reachable}\ s \wedge \mathsf{reachable}\ t \wedge s \overset{\mathsf{PSched}}{\sim} t \\
&\wedge\ s \overset{\mathsf{sources}\ n\ s\ p}{\approx} t \longrightarrow s \overset{p}{\sim}_n t
\end{aligned}
$$

This definition considers pairs of finite executions of identical length n, in line with the observation above that purge-based definitions are not appropriate in our setting.

This definition is also entirely *termination insensitive*, because it is trivially satisfied when non-termination occurs. However, the functional correctness proof for seL4 proves that its execution is always defined, implying that all transitions for the transition system depicted in Figure 4(a) always terminate. Under the assumption that the interrupt oracle delivers an infinite stream of timer interrupts, it is relatively straightforward to prove that the coalesced transitions of the transition system depicted in Figure 4(b) always terminate too—since a timer interrupt will always arrive that then causes the scheduling partition to take over. Hence, non-termination is not an issue and a termination insensitive formulation of information flow security is appropriate here.

D. Information Flow Theorem

Let $\mathcal{M}_A$ be the automaton for the seL4 abstract specification, and $\mathsf{nonleakage}_A$ denote nonleakage applied to $\mathcal{M}_A$. The top-level information flow theorem we prove for $\mathcal{M}_A$, simplified for presentation, is the following.

Theorem 1: **seL4's abstract specification enforces information flow security.** Let s_0 denote the initial state of the system, after configuration, and *pas* be an access control policy, and $\rightsquigarrow$ the corresponding information flow policy. Then if s_0 satisfies the kernel invariants, *pas* is consistent with s_0 and wellformed for all subjects, and all interrupts other than the timer interrupt are disabled in s_0, and all subject-crossing capabilities are safe in the sense described below in Section IV-E, then nonleakage is enforced:

$$
\begin{aligned}
&\mathsf{invs}\ s_0 \wedge \mathsf{pas\text{-}refined\text{-}wellformed}\ \mathit{pas}\ s_0 \wedge \mathsf{only\text{-}timer\text{-}irq}\ s_0\ \wedge \\
&\mathsf{sscc}\ \mathit{pas}\ s_0 \longrightarrow \mathsf{nonleakage}_A
\end{aligned}
$$

Here, sscc is a condition on capabilities that cross partition boundaries, described later in Section IV-E. Intuitively, it ensures that partition-crossing communication channels can never be destroyed, as destroying an otherwise unidirectional channel signals to both sender and receiver.

Letting $\mathcal{M}_C$ be the corresponding automaton for the seL4 C implementation, and $\mathsf{nonleakage}_C$ denote nonleakage applied to $\mathcal{M}_C$, the functional correctness proof [27] implies that: $\mathcal{M}_A \sqsubseteq \mathcal{M}_C$. We then have that because nonleakage is preserved by refinement: $\mathcal{M}_A \sqsubseteq \mathcal{M}_C \wedge \mathsf{nonleakage}_A \longrightarrow \mathsf{nonleakage}_C$. Information flow security for seL4's C implementation then follows trivially.

Theorem 2: **seL4's C implementation enforces information flow security.** Let s_0 denote the initial state of the system, after configuration, and *pas* be an access control policy, and $\rightsquigarrow$ the corresponding information flow policy. Then:

$$
\begin{aligned}
&\mathsf{invs}\ s_0 \wedge \mathsf{pas\text{-}refined\text{-}wellformed}\ \mathit{pas}\ s_0 \wedge \mathsf{only\text{-}timer\text{-}irq}\ s_0\ \wedge \\
&\mathsf{sscc}\ \mathit{pas}\ s_0 \longrightarrow \mathsf{nonleakage}_C
\end{aligned}
$$

E. *Proving Information Flow Security*

Like other noninterference variants, nonleakage is proved by discharging proof obligations called *unwinding conditions* that examine individual execution steps. The following unwinding condition, called confidentiality-u, is sound and complete for (i.e. is equivalent to) nonleakage [36].

$$\textsf{confidentiality-u} \equiv \forall p\ s\ t.\ \textsf{reachable}\ s \wedge \textsf{reachable}\ t \wedge s \overset{p}{\sim} t \wedge s \overset{\textsf{PSched}}{\sim} t \wedge (\textsf{part}\ s \rightsquigarrow p \longrightarrow s \overset{\textsf{part}\ s}{\sim} t) \longrightarrow s \overset{p}{\sim}_1 t$$

It says that the contents of each partition p after each step can depend only on the contents of the following partitions before the step: p, PSched and the currently running partition part s when it is allowed to send information to p. In other words, information may flow to p only from PSched and the current partition in accordance with the information flow policy $\rightsquigarrow$.

To prove this condition for the execution steps of our transition system (depicted in Figure 4(b)), we consider the following cases.

Case 1 — part $s \rightsquigarrow p$: In this case confidentiality-u collapses to the following property, noting that part $s \rightsquigarrow p \wedge p = \textsf{PSched} \longrightarrow \textsf{part}\ s = \textsf{PSched}$ because, $\rightsquigarrow$ is purposefully constructed so that $\forall p'.\ p' \rightsquigarrow \textsf{PSched} \longrightarrow p' = \textsf{PSched}$:

$$\forall p\ s\ t.\ \textsf{reachable}\ s \wedge \textsf{reachable}\ t \wedge s \overset{p}{\sim} t \wedge s \overset{\textsf{PSched}}{\sim} t \wedge \textsf{part}\ s \rightsquigarrow p \wedge s \overset{\textsf{part}\ s}{\sim} t \wedge (p = \textsf{PSched} \longrightarrow \textsf{part}\ s = \textsf{PSched}) \longrightarrow s \overset{p}{\sim}_1 t \qquad (1)$$

This property we discharge using a *relational* proof calculus [36], similar in style to the seminal work of Benton [10] and other reasoning systems for confidentiality properties [4], [5], with an automated verification condition generator [14].

We prove Property 1 for each of the small transitions of Figure 4(a) to conclude it holds for the coalesced transitions of Figure 4(b).

Case 2 — part $s \not\rightsquigarrow p$: In this case, we consider two sub-cases.

a) $p = \textsf{PSched}$

In this case, we prove the following condition, noting that part $s \not\rightsquigarrow p \wedge p = \textsf{PSched} \longrightarrow \textsf{part}\ s \neq \textsf{PSched}$ because $\rightsquigarrow$ is reflexive:

$$\forall s\ t.\ \textsf{reachable}\ s \wedge \textsf{reachable}\ t \wedge s \overset{\textsf{PSched}}{\sim} t \wedge \textsf{part}\ s \neq \textsf{PSched} \longrightarrow s \overset{\textsf{PSched}}{\sim}_1 t \qquad (2.a)$$

This requires us to show that the scheduling partition's contents after a transition of another partition depends only on its contents beforehand. All of PSched's contents remains unchanged during the execution of other partitions except the current position of the interrupt oracle (Section IV-B). As explained earlier, however, the transition system of Figure 4(b) is purposefully constructed to reflect the reality that the oracle position after the execution of a non-scheduling partition will always be precisely the position of the next timer interrupt in the stream. The location of all timer interrupts in the oracle stream and the current oracle position are included in PSched's contents, under the assumption that all partitions are allowed to learn about the passage of global time. Hence, Property 2.a above follows easily.

b) $p \neq \textsf{PSched}$

In this final case we use Sewell et al.'s integrity theorem [51] for seL4 (Section III-D) to prove the following property, which says that the current transition may not alter p at all. confidentiality-u then follows from symmetry and transitivity.

$$\forall p\ s\ s'.\ \textsf{reachable}\ s \wedge p \neq \textsf{PSched} \wedge \textsf{part}\ s \not\rightsquigarrow p \wedge (s, s') \in \textsf{Step} \longrightarrow s \overset{p}{\sim} s' \qquad (2.b)$$

Integrity holds for all of the small transitions of Figure 4(a), and thus holds for the coalesced transitions of Figure 4(b), and so implies Property 2.b.

Of these cases, Case 1 is the most interesting and consumed the bulk of the work. We briefly describe the most illuminating aspects of this effort.

As mentioned above, to prove Property 1 across the compound transitions of Figure 4(b), we proved it across each of the individual component transitions of Figure 4(a). There are essentially two distinct sets of transitions to consider here, namely those for the scheduling partition and those for non-scheduling partitions.

The proofs for the scheduling partition necessarily cover the case where the scheduler does a partition switch when the timeslice of the current partition expires. In this case, proving Property 1 for the situation in which p is the new partition being scheduled involves showing that the partition switch leaves no *residual information* behind from the old partition that was previously running. The ARM implementation of seL4 maintains a fixed page of memory, called the *globals frame*, that is shared between all threads and at any point in time contains information relevant to the currently running thread. When scheduling a thread, the kernel writes new data to this page for the now active thread. Part of proving Property 1 therefore involved proving that the globals frame, which essentially moves between partitions on each partition switch, contains no residual information after a partition switch and so cannot serve as a covert storage channel. The same also had to be done for the machine context, which includes the CPU registers that are saved and restored on kernel entry and exit respectively, as well as all other global resources that are shared between partitions like the system idle thread.

The proofs for Property 1 for non-scheduling transitions mostly involve reasoning that the state read by the kernel when performing a system call for the currently active partition is contained within the extent of the current partition—i.e. that the kernel reads only state that it is permitted to reveal to the current partition. These proofs were relatively straightforward, and benefited from substantial automation.

The exceptions to this rule, however, are system calls that facilitate communication between partitions. One such system call is that for sending a message on an asynchronous endpoint, send-async-ipc. Proving Property 1 for this system call requires considering the case in which *p* is the current partition who is sending on the asynchronous endpoint. In this case, we must prove that *p*'s contents after the system call do not depend on the contents of the asynchronous endpoint being sent on (which necessarily lives outside of *p*'s extent) nor the contents of any partition who is waiting on the endpoint in question and so who will receive the message being sent. In other words, proving Property 1 involves proving that the kernel's implementation of asynchronous endpoints has no covert storage back-channels.

The kernel necessarily reads the internal state of the endpoint being sent on, so we must prove that the kernel reveals none of this read state to *p*. This involves considering all possible internal states that the endpoint being sent on might be in, and proving that the effects of send-async-ipc as observed by the sending partition are identical in all cases. These proofs had to be performed more manually, and were amongst the most challenging for the entire effort.

Besides the kernel's primitive facility for interrupt delivery, which we exclude by assuming that all non-timer IRQs are disabled, the only other problematic kernel behaviour that we encountered during the proof was object deletion. Specifically, seL4 deletes an object when the last capability to that object is deleted. This capability is called *final*. Thus the behaviour of the kernel, when a capability is deleted, depends on which other partitions possess a capability to the same object. This opens up a potential storage channel.

As with interrupt delivery, we avoided this problem by placing an assumption on the initial configuration of the system. We assume that when the partitions were set up that an extra inert CNode was also created, to which no partition was granted access, and that a copy of every *subject-crossing* capability was placed into this inert CNode. A subject-crossing capability is one that refers to a subject that is different from the one that possess the capability, with subject boundaries defined with reference to the access control policy *pas*. Since these copies can never be deleted (because no subject has access to the inert CNode that holds them), this ensures that only non-subject-crossing capabilities can ever become final. This assumption is formalised by the property sscc, which stands for *safe subject-crossing capabilities*. It is relatively easy to prove that sscc is invariant. Under sscc, the behaviour when deleting a capability depends only on the currently running partition, and so ensures that confidentiality is not violated. Intuitively, this restriction enforces that communication interfaces between partitions should be static, because any change in that interface causes a bidirectional information flow.

V. Discussion

Having presented our proof and statement of information flow security for seL4, we now analyse its strengths and limitations, and relate the result to its practical significance.

The history of computer security is littered with published security proofs that were later broken by novel exploits and attacks. This happens when: (1) the proof is incorrect, i.e. not logically valid, (2) the proof's assumptions are not realistic, or (3) the property proved was not strong enough or does not mean what we thought it did. We consider each in turn.

A. Proof Correctness

Our proof for seL4 is machine-checked, and carried out in the interactive theorem prover Isabelle/HOL [38], which is a prover for higher-order logic in the so-called LCF family with strong soundness properties: all derivations must pass through a small proof kernel. While a defect in the Isabelle proof kernel may permit errors in our proof, this possibility can be made arbitrarily small by extracting the proof term from Isabelle and running it through a series of small, independently written proof-checking programs. Errors in the proof itself are therefore a non-issue in practice [21].

B. Assumptions

The assumptions on which our proof rests are realistic, and amenable to validation. Our proof makes three explicit assumptions about the seL4 configurations to which it applies. Firstly, it assumes that the system has been correctly configured to enforce information flow control, by asserting that the access control policy is consistent with the initial state and wellformed. Secondly, it assumes that only the timer interrupt, used by the kernel to control scheduling, is enabled. Thirdly, it assumes that there exist inert copies of all subject-crossing capabilities to prevent any such capability from becoming final (see Section IV-E). The first of these is an obvious requirement. The second and third are required to ensure that kernel functionality that is potentially problematic for separation is never invoked. Each of these assumptions is easily ensured by careful system initialisation, which itself is amenable to formal verification [6]. Only the latter two place limitations on the kinds of systems that can be constructed, by forcing device drivers to poll for interrupts and preventing inter-partition communication channels from being destroyed respectively. Neither of these limitations is uncommon in deployed separation kernels.

Our proof also makes a number of extra-logical assumptions. Many of these are inherited directly from the seL4 functional correctness proof [27] on which our result builds. These inherited assumptions are that the C compiler and linker used to build the kernel correctly implement the formal C semantics [52] of the proof on the platform of deployment (i.e. compiler and linker correctness), that the behaviour of the deployment hardware matches the formal machine model on which the proofs rest (i.e. hardware correctness), that the kernel's 450 lines of assembly code correctly match their specification, including that caching- and TLB-operations are placed correctly, and that the kernel's initialisation code that runs before it hands control to the initial thread correctly establishes the kernel invariants.

Many of these assumptions are themselves amenable to formal verification, particularly compiler/linker correctness [13], [50], assembly and context-switching code correctness [37] and correctness of the initialisation code, an earlier version of which was verified at the design level [27]. The hardware model that we share with the seL4 functional correctness proofs effectively assumes that DMA is disabled.

Our proof also brings a few new implicit assumptions. Our formulation of information flow security assumes that the global static partition schedule is allowed to be known by all partitions. Hence it does not prevent one partition from knowing about the existence of another, nor does it prevent a newly scheduled partition from inferring that the previously running partition must have exhausted its timeslice.

Our model of interrupts, described in Section IV-B, implies that partitions can observe the passage of global time. Stronger separation where time is not visible to all partitions could be imagined, but our proof does not enforce this.

A technicality placed on the interrupt oracle by our termination insensitive formulation of information flow security is that the oracle delivers an infinite stream of timer interrupts. This ensures that partition steps always terminate.

As mentioned in Section I, we implicitly assume that DMA is not enabled. We also assume that user-space threads have direct access to only those sources of information that we model: machine registers and memory pages mapped with read rights, so that user-space threads may be modelled as a deterministic function of these inputs. Thus we implicitly assume that the initial configuration prevents partitions communicating via external devices.

C. Covert Channels

Our noninterference property, while true of the C implementation, is phrased against, and so has meaning at the level of, the kernel's abstract specification. As explained in Section I, the formal machine model on which our proofs rest does not model time explicitly. While exposed coarsely through the interrupt oracle (Section IV-B), our proof says little about covert timing channels. Timing channels must still be analysed and mitigated using traditional means.

As mentioned earlier, our partition-based scheduling implementation is known to suffer from jitter, in that it allows a partition to overrun its timeslice by performing a system call just before the arrival of the next timer interrupt. Other obvious timing channels that certainly exist in the current seL4 implementation but are likewise not addressed by our proof include timing channels due to shared caches or devices. Each of these has obvious mitigation strategies, such as preventing system calls during the last n timer ticks of a partition, flushing caches on partition switches or dividing caches between partitions using cache colouring and so on. However, deciding on a particular implementation necessarily involves a trade-off between performance and covert channel bandwidth that can only be made within the context of a particular deployment scenario. For seL4, this analysis can also be made with reference to sound worst-case execution time (WCET) calculations for the kernel [12].

Our proof says much more about the absence of covert storage channels, particularly those that might be in the kernel. We list several channels uncovered by the proof in Section V-D; all were eliminated either through making the abstract specification more concrete (see below) or by adding assumptions on the initial configuration. Our proof certainly rules out all storage channels present in the seL4 abstract specification, including all user-accessible physical memory, CPU registers, and machine state such as the interrupt masks. It also includes the abstract representation of the kernel heap present in the abstract specification. What is more, because our formulation of information flow security is preserved under refinement, it also rules out user-visible kernel storage channels below the level of abstraction of the abstract specification, such as the raw memory of the C kernel heap. This is because any such channel must show up as user-visible nondeterminism exhibited by the kernel. In order to be preserved by refinement, our information flow security formulation tolerates no user-visible nondeterminism [36]: such nondeterminism could always be refined by an insecure implementation that resolves the nondeterminism by examining a secret piece of state, and therefore cannot be tolerated by a refinement-closed security condition [35].

Indeed, our proof of information flow security uncovered a number of cases where the kernel uses state present in the C kernel heap to make choices that are visible to user-space, but where that state was below the level of abstraction of the abstract specification. Each of these showed up as user-visible nondeterminism in the abstract specification, and was flagged by the proof as a potential covert storage channel. To prove information flow security, we had to make the kernel specification more concrete to remove the user-visible nondeterminism [33], and then re-establish the functional correctness and integrity results for the augmented specification. The remaining nondeterminism in the abstract specification is never revealed by the kernel to user-space—our proof guarantees this—and includes for instance the mapping between physical address space identifiers (ASIDs) and the virtual pool of such ASIDs that seL4 maintains.

Our proof does not rule out the possibility of covert storage channels that are below the level of abstraction of the abstract specification, but that the kernel never reads. For instance, suppose the kernel were ported to a new platform that included extra CPU registers that the kernel never reads, but that the port was done incorrectly such that the kernel fails to clear these registers on a partition switch. It is possible our proof would still hold despite the presence of an obvious covert storage channel. Formally, this is captured by the hardware correctness assumption. Guarding against it requires validating that the formal machine model corresponds to the platform of deployment.

D. Lessons Learned

The proof was carried out over 21 months, and required a total effort of roughly 51 person-months (pm). This includes adding the partition scheduler ($\approx$ 2 pm), making the abstract specification more deterministic ($\approx$ 23 pm), and the information flow security proofs themselves ($\approx$ 26 pm).

The proof of information flow security comprises 27,756 lines of Isabelle/HOL not including whitespace and comments. This figure excludes the changes made to the abstract specification, the repair to the functional correctness proofs, and the necessary evolution and strengthening of the integrity and authority confinement results needed for them to hook up with the information flow statement.

While the total effort for information flow control is higher than the 10 person-months reported by Sewell et al. for integrity and authority confinement [51], it is still far below the 25 person-years of effort required for the original functional correctness proofs for seL4 [27]. As with the previous proof of integrity, we gained a significant saving in effort by being able to prove information flow security for seL4's C implementation over its abstract specification. Sewell et al. estimate that proving integrity over seL4's C implementation directly would have required on the order of the original 25 person-years to complete. We estimate an even higher figure for information flow security, even assuming an initial proof of integrity on which to build on. Unlike with integrity, however, proving information flow security over seL4's abstract specification came at the cost of having to remove much of the nondeterminism from the abstract specification (see also Section V-C). Because the effort required to do so ($\approx$ 23 pm) was low in comparison, proving information flow security over the abstract specification was still undoubtedly the right thing to do.

The proof uncovered many channels in the kernel, some of which were initially surprising even to those who had worked with seL4 for years. Very early on in the proof, the problem of seL4's object deletion semantics (see Section IV-E), in which an object is deleted only when the last capability in existence to it is deleted, became apparent. That this behaviour could give rise to a potential channel was something that had not been explicitly considered before. We decided to address this by adding an assumption on the initial configuration. We were then obliged to prove that this assumption was sufficient to remove the potential channel.

Another channel uncovered by the proof was connected to the kernel's interrupt delivery mechanism, namely that the kernel does not isolate the interrupts of one partition from another. Taking advantage of the fact that polling for device interrupts is common practice in separation kernels, we again decided to add the assumption that this API facility is disabled at startup instead of rewriting seL4's interrupt handling code. The proof again forced us to show that this was sufficient to remove the channel.

Other channels that the proof forced us to reason about were anticipated from the beginning: our rules for constructing the information flow policy $\leadsto$ from the access control policy explicitly allow a two-way flow of information between partitions connected by a synchronous endpoint, for instance. The proof still forced us to show that synchronous endpoints, while allowing a bidirectional flow between sender and receiver, do not leak information to anyone else.

Similarly, the original seL4 scheduler, before partition scheduling was implemented, was known not to enforce isolation. We could not prove information flow security until we had fully and correctly specified the partition scheduler in the updated abstract specification. Proving information flow security then required us to show that the scheduler's choice about which partition to schedule next can never be affected by any other partition, as one would expect.

Apart from one minor change to simplify verification, the partition scheduler was the only change required to the seL4 C code, which is what we expected when we began. It provides some evidence of the security that can be gained by going through the process of rigorously designing and verifying a microkernel, even without a formal proof of security. However, our formal proof of security is what separates optimistic hope from well-founded confidence, grounded in formal proof.

Ultimately, our proof of information flow security for seL4 makes seL4 no more secure than it was to begin with (excepting the implementation changes mentioned above). However, it provides a very strong piece of evidence about the security of seL4 and its suitability as a separation kernel—the strongest such piece of evidence ever constructed for a general-purpose kernel.

VI. Related Work

seL4 is a general-purpose microkernel, whose implementation we have proved can be configured to enforce static information flow security in the form of intransitive noninterference. A number of kernels are designed specifically to enforce information flow control, such as HiStar [54] whose size is comparable to seL4's. HiStar implements a simple semantics for enforcing information flow control, based on object labels and category ownership. However, to our knowledge, there exists no formal proof that these rules correctly model the behaviour of the HiStar implementation, nor a formal connection between these rules and a high-level security property like intransitive noninterference.

The first serious attempts to verify an OS kernel appeared in the late 1970s with UCLA Secure Unix [53] and the Provably Secure Operating System (PSOS) [17], and in the 1980s with KIT [11]. The design methodology of PSOS was later used for the Kernelized Secure Operating System (KSOS) [43] by Ford Aerospace. The Secure Ada Target (SAT) [19] and the Logical Coprocessor Kernel (LOCK) [49] are also inspired by the PSOS design and methodology. The proof efforts of this time primarily aimed at achieving functional correctness; security proofs of the style presented here were not yet feasible. Klein [26] provides a more comprehensive overview of this early work. Functional correctness at the code level has only recently been achieved by Klein et al. [27], on which we build, and also independently by the Verisoft project [2].

Our proof further builds on the seL4 integrity proof by Sewell et al. [51], and on the noninterference property and associated proof calculus developed in earlier work [36].

Proofs of information flow security for models of OS kernels and hypervisors are not new. Below, we summarise other recent work with similar goals. To our knowledge, ours is the only mechanised proof that applies to the C code of a general-purpose OS kernel/hypervisor. The C code level is significant, because no manual checking is needed to verify the validity of the proof for the running artefact and it is easy to validate that the proof still applies after code changes. The proof check is mechanical and fully automatic, and all steps from the C code on down are automatically generated by compiler and linker, so any remaining errors are systematic and not subject to error-prone human validation for every new deployment or code version.

The work that comes closest to the one presented here is INTEGRITY-178B, which is a real-time operating system for which a machine-checked information flow proof has been completed [44]. However, unlike ours, this proof applies to a hand-written, detailed formal model of the kernel that is not linked to its implementation by formal proof but instead by careful informal argument. This leaves open the possibility of implementation errors in INTEGRITY-178B that invalidate the proof of isolation, and risks that the proof is not adequately updated when code or API change. The isolation proved for INTEGRITY-178B is based on the GWVr2 property [18], which bears similarities to our formulation of information flow security for seL4. The exact relationship between the two deserves further study.

The Mathematically Analyzed Separation Kernel (MASK) was also proved to enforce a notion of information flow control [31], [32]. Their property resembles traditional unwinding conditions for noninterference, and was shown to hold for a low-level design model that is close to an implementation. Again, it was ultimately connected to the C implementation only by manual translation. Like many of the other kernels summarised here, MASK is not a general-purpose kernel such as seL4, but instead designed primarily to enforce static separation. This means that the verification of seL4 is more complex and, at the same time, that more flexible kernel services are available inside partitions.

Heitmeyer et al. [22], [23] present a verification of separation for an unnamed separation kernel, whose main purpose is to enforce data separation. Their formulation of separation involves a number of different properties: no exfiltration, no infiltration, temporal separation, control separation and kernel integrity. We can derive analogues for each of these properties for seL4 from our proof of information flow security. No exfiltration is a consequence of integrity; no infiltration a consequence of confidentiality; temporal separation corresponds to an absence of residual information on each partition switch, which is required by our formulation of nonleakage (see Section IV-E); control separation requires that only one partition executes at a time, and for seL4 is a consequence of functional correctness; and, finally, kernel integrity is also a consequence of functional correctness for seL4. They give a machine checked proof for an abstract model of the separation kernel, which is related to its C implementation by a pen-and-paper proof. The size of the separation kernel was reported at 3,000 lines of C and assembly, which is under a half the size of seL4.

Hardin et al. [20] verified information-flow control properties of the AAMP7 microprocessor [45], which implements the functionality of a simple static separation kernel in hardware. Similar to other work above, the proof, based on a comparatively detailed model, is connected to the (in this case micro-)code by careful manual inspection.

Krohn and Tromer [28] presented a pen-and-paper proof of noninterference for the Flume operating system. This proof applied to a very abstract CSP [24] model of the Flume system, unconnected to its implementation by proof. Unlike seL4, Flume is Linux-based and so includes the entire Linux kernel as part of its trusted computing base (TCB).

Recently, Barthe et al. [8] presented a formalisation and machine-checked proof of isolation for a high-level *idealised* model of a hypervisor. More recent work in this vein [9] has also looked at analysing cache leakage, which our proof does not, but again only for an idealised hypervisor model.

VII. Conclusion

We have presented the most detailed and extensive machine-checked formal verification of information flow security ever performed for a general-purpose OS kernel, specifically a proof of intransitive noninterference for seL4. Our proof holds for seL4's C implementation, and builds on the previous verification results for seL4 that established functional correctness of its C implementation and integrity enforcement. Taken together with these previous results, the verification for seL4 now covers properties from the level of integrity and confidentiality, over functional correctness, down to the C implementation, accomplishing a 30-year stated research goal of the field.

While this proof, like any other assurance mechanism, has limitations such as making no claims about timing channels, a number of the current proof assumptions can be strengthened. For instance, recent work on eliminating the compiler and linker assumption by extending the functional correctness proof to the binary level [50] could be adapted to automatically derive binary-level noninterference for seL4.

The additional effort for proving noninterference here was much reduced in comparison to the effort for the previous functional correctness proof, but still substantial for many applications. The main application domain of kernels like this is high-assurance systems, such as space, aviation, vehicles, and critical infrastructure, where expending this kind of effort is justified to save lives and prevent substantial damage. However, seL4 is a general, high-performance microkernel, capable of hosting entire legacy applications and operating systems like Linux. Because this verification need be performed only once for each architecture, nothing prevents us using this kind of system far more broadly.

While a result like this has intrinsic value on its own, we see it merely as one step in a bigger vision. The formal statement of security mechanism enforcement is a tool for reasoning about the security goals of entire systems built on top of the OS kernel. By having this formal tool available, such reasoning now becomes feasible for the first time.

Acknowledgements

Thanks to Gilles Barthe, Andrei Sabelfeld. Chris North, the anonymous reviewers, Cătălin Hriţcu and Gernot Heiser for their feedback on earlier drafts of this paper.

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. Part of the research reported in this document was performed in connection with Contract Number DAAB W15P7T-12-C-A105 with the US Army CECOM LCMC Command. The views and conclusions contained in this document are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of the US Army CECOM LCMC Command or the US Government unless so designated by other authorized documents. Citation of manufacturers or trade names does not constitute an official endorsement or approval of the use thereof. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

[1] M. Abadi and L. Lamport, “The existence of refinement mappings,” *Theor. Comput. Sci.*, vol. 82, pp. 253–284, 1991.

[2] E. Alkassar, W. Paul, A. Starostin, and A. Tsyban, “Pervasive verification of an OS microkernel,” in *VSTTE 2010*, ser. LNCS, vol. 6217. Springer, Aug 2010, pp. 71–85.

[3] *AMD I/O Virtualization Technology (IOMMU) Specification*, AMD, 2009, rev 1.26. http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf.

[4] T. Amtoft and A. Banerjee, “Information flow analysis in logical form,” in *SAS ’04*, ser. LNCS, vol. 3148. Springer, 2004, pp. 33–36.

[5] ——, “Verification condition generation for conditional information flow,” in *FMSE ’07*. ACM, 2007, pp. 2–11.

[6] J. Andronick, G. Klein, and A. Boyton, “Formal system verification - extension, AOARD 114070,” NICTA, Sydney, Australia, Tech. Rep. 1833-9646-5926, May 2012.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in *19th SOSP*, Oct 2003, pp. 164–177.

[8] G. Barthe, G. Betarte, J. D. Campo, and C. Luna, “Formally verifying isolation and availability in an idealized model of virtualization,” in *17th FM*, ser. LNCS, vol. 6664. Springer, 2011, pp. 231–245.

[9] ——, “Cache-leakage resilient OS isolation in an idealized model of virtualization,” in *25th CSF*, 2012, pp. 186–197.

[10] N. Benton, “Simple relational correctness proofs for static analyses and program transformations,” in *POPL 2004*. ACM, 2004, pp. 14–25.

[11] W. R. Bevier, “Kit: A study in operating system verification,” *Trans. Softw. Engin.*, vol. 15, no. 11, pp. 1382–1396, 1989.

[12] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, “Timing analysis of a protected operating system kernel,” in *32nd RTSS*, Nov 2011, pp. 339–348.

[13] S. Blazy, Z. Dargaye, and X. Leroy, “Formal verification of a C compiler front-end,” in *14th FM*, ser. LNCS, vol. 4085. Springer, 2006, pp. 460–475.

[14] D. Cock, G. Klein, and T. Sewell, “Secure microkernels, state monads and scalable refinement,” in *21st TPHOLs*, ser. LNCS, vol. 5170. Springer, Aug 2008, pp. 167–182.

[15] W.-P. de Roever and K. Engelhardt, *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[16] J. B. Dennis and E. C. Van Horn, “Programming semantics for multiprogrammed computations,” *CACM*, vol. 9, pp. 143–155, 1966.

[17] R. J. Feiertag and P. G. Neumann, “The foundations of a provably secure operating system (PSOS),” in *AFIPS Conf. Proc., 1979 National Comp. Conf.*, Jun 1979, pp. 329–334.

[18] D. A. Greve, “Information security modeling and analysis,” in *Design and Verification of Microprocessor Systems for High-Assurance Applications*, D. S. Hardin, Ed. Springer, 2010, pp. 249–300.

[19] J. T. Haigh and W. D. Young, “Extending the noninterference version of MLS for SAT,” *Trans. Softw. Engin.*, vol. 13, pp. 141–150, Feb 1987.

[20] D. S. Hardin, E. W. Smith, and W. D. Young, “A robust machine code proof framework for highly secure applications,” in *ACL2 ’06*. ACM, 2006, pp. 11–20.

[21] G. Heiser, T. Murray, and G. Klein, "It's time for trustworthy systems," *IEEE: Security & Privacy*, vol. 2012, no. 2, pp. 67–70, Mar 2012.

[22] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean, "Formal specification and verification of data separation in a separation kernel for an embedded system," in *13th CCS*. ACM, 2006, pp. 346–355.

[23] ——, "Applying formal methods to a certifiably secure software system," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 82–98, 2008.

[24] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[25] *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*, Information Assurance Directorate, Jun 2007, version 1.03. http://www.niap-ccevs.org/cc-scheme/pp/pp.cfm/id/pp_skpp_hr_v1.03/.

[26] G. Klein, "Operating system verification — an overview," *Sādhanā*, vol. 34, no. 1, pp. 27–69, Feb 2009.

[27] 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*. Big Sky, MT, USA: ACM, Oct 2009, pp. 207–220.

[28] M. Krohn and E. Tromer, "Noninterference for a practical DIFC-based operating system," in *IEEE Symp. Security & Privacy*, 2009, pp. 61–76.

[29] B. Leslie, C. van Schaik, and G. Heiser, "Wombat: A portable user-mode Linux for embedded systems," in *6th Linux.conf.au*, Canberra, Apr 2005.

[30] R. J. Lipton and L. Snyder, "A linear time algorithm for deciding subject security," *J. ACM*, vol. 24, no. 3, pp. 455–464, 1977.

[31] W. B. Martin, P. White, F. Taylor, and A. Goldberg, "Formal construction of the mathematically analyzed separation kernel," in *15th ASE*, 2000, pp. 133 –141.

[32] W. B. Martin, P. D. White, and F. S. Taylor, "Creating high confidence in a separation kernel," *Automated Softw. Engin.*, vol. 9, no. 3, pp. 263–284, 2002.

[33] D. Matichuk and T. Murray, "Extensible specifications for automatic re-use of specifications and proofs," in *10th SEFM*, ser. LNCS, vol. 7504, Oct 2012, pp. 333–341.

[34] van der Meyden, Ron, "What, indeed, is intransitive noninterference?" in *12th ESORICS*, ser. LNCS, vol. 4734. Springer, 2007, pp. 235–250.

[35] T. Murray and G. Lowe, "On refinement-closed security properties and nondeterministic compositions," in *8th AVoCS*, ser. ENTCS, vol. 250, Glasgow, UK, 2009, pp. 49–68.

[36] T. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein, "Noninterference for operating system kernels," in *2nd CPP*, ser. LNCS, vol. 7679. Springer, Dec 2012, pp. 126–142.

[37] Z. Ni, D. Yu, and Z. Shao, "Using XCAP to certify realistic system code: Machine context management," in *20th TPHOLs*, ser. LNCS, vol. 4732, Sep 2007, pp. 189–206.

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

[39] NIST, "National vulnerability database," http://nvd.nist.gov, 2012.

[40] von Oheimb, David, "Information flow control revisited: Noninfluence = noninterference + nonleakage," in *9th ESORICS*, ser. LNCS, vol. 3193, 2004, pp. 225–243.

[41] Open Kernel Labs, "seL4 research and evaluation download," http://ertos.nicta.com.au/software/seL4/, 2011.

[42] P. Parkinson and A. Baker, "High assurance systems development using the MILS architecture," http://www.windriver.com/whitepapers/, Wind River Systems Inc., 2010.

[43] T. Perrine, J. Codd, and B. Hardy, "An overview of the kernelized secure operating system (KSOS)," in *Proceedings of the Seventh DoD/NBS Computer Security Initiative Conference*, Sep 1984, pp. 146–160.

[44] R. J. Richards, "Modeling and security analysis of a commercial real-time operating system kernel," in *Design and Verification of Microprocessor Systems for High-Assurance Applications*, D. S. Hardin, Ed. Springer, 2010, pp. 301–322.

[45] Rockwell Collins, Inc., *AAMP7r1 Reference Manual*, 2003.

[46] J. Rushby, "Noninterference, transitivity, and channel-control security policies," SRI International, Tech. Rep. CSL-92-02, Dec 1992.

[47] J. M. Rushby, "Design and verification of secure systems," in *8th SOSP*, Pacific Grove, CA, USA, Dec 1981, pp. 12–21.

[48] A. Sabelfeld and A. Myers, "Language-based information-flow security," *J. Selected Areas Comm.*, vol. 21, no. 1, pp. 5–19, Jan 2003.

[49] O. Saydjari, J. Beckman, and J. Leaman, "Locking computers securely," in *10th National Computer Security Conference*, Sep 1987, pp. 129–141.

[50] T. Sewell, M. Myreen, and G. Klein, "Translation validation for a verified OS kernel," in *PLDI 2013*. ACM, 2013, to appear.

[51] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein, "seL4 enforces integrity," in *2nd ITP*, ser. LNCS, vol. 6898. Springer, Aug 2011, pp. 325–340.

[52] H. Tuch, G. Klein, and M. Norrish, "Types, bytes, and separation logic," in *34th POPL*. ACM, 2007, pp. 97–108.

[53] B. J. Walker, R. A. Kemmerer, and G. J. Popek, "Specification and verification of the UCLA Unix security kernel," *CACM*, vol. 23, no. 2, pp. 118–131, 1980.

[54] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," *CACM*, vol. 54, no. 11, pp. 93–101, Nov 2011.