Noninterference for Operating System Kernels

Toby Murray[1,2], Daniel Matichuk[1], Matthew Brassil[1], Peter Gammie[1] and Gerwin Klein[1,2]

[1] NICTA, Sydney, Australia* **
[2] School of Computer Science and Engineering, UNSW, Sydney, Australia
{firstname.lastname}@nicta.com.au

Abstract. While intransitive noninterference is a natural property for any secure OS kernel to enforce, proving that the implementation of any particular general-purpose kernel enforces this property is yet to be achieved. In this paper we take a significant step towards this vision by presenting a machine-checked formulation of intransitive noninterference for OS kernels, and its associated sound and complete unwinding conditions, as well as a scalable proof calculus over nondeterministic state monads for discharging these unwinding conditions across a kernel's implementation. Our ongoing experience applying this noninterference framework and proof calculus to the seL4 microkernel validates their utility and real-world applicability.

Keywords: Information flow, refinement, scheduling, state monads.

1 Introduction

A primary function of any operating system (OS) kernel is to enforce security properties and policies. The classical security property of *noninterference* [8] formalises the absence of unwanted information flows within a system, and is a natural goal for any secure OS to aim to enforce. Here, the system is divided into a number of *domains*, and the allowed information flows between domains specified by means of an information flow *policy* $\rightsquigarrow$, such that $d \rightsquigarrow d'$ if information is allowed to flow from domain d to domain d'. So-called *intransitive noninterference* [10] generalises noninterference to the case in which the relation $\rightsquigarrow$ is possibly intransitive.

While intransitive noninterference is a natural property for any secure OS kernel to enforce, proving that the implementation of any particular general-purpose kernel enforces this property is yet to be achieved. In this paper we

* 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.

** This material is in part based on research sponsored by the Air Force Research Laboratory, under agreement number FA2386-10-1-4105. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

take a significant step towards this vision by presenting a machine-checked formulation of intransitive noninterference for OS kernels, and its associated sound and complete proof obligations (called *unwinding conditions*), as well as a scalable proof calculus over nondeterministic state monads for discharging these unwinding conditions across a kernel's implementation. Both our noninterference formulation and proof calculus are termination-insensitive, under the assumption that a noninterference verification for an OS kernel is performed only after proving that its execution is always defined (and thus every system call always terminates). Our experience applying this noninterference framework and proof calculus to the seL4 microkernel [11] validates their utility and real-world applicability.

Our intransitive noninterference formulation improves on traditional formulations [10, 16, 19, 21] in two ways that make it more suitable for application to OS kernels. Firstly, traditional formulations of intransitive noninterference assume a static mapping dom from actions to domains, such that the domain dom a on whose behalf some action a is being performed can be determined solely from the action itself. No such mapping exists in the case of an OS kernel, which must infer this information at run-time. For instance, when a system call occurs, in order to work out which thread has requested the system call the kernel must consult the data-structures of the scheduler to determine which thread is currently running. This prevents traditional noninterference formulations from being able to reason about potential information flows that might occur via these scheduling data structures. An example would be a scheduler that does not properly isolate domains by basing its decision about whether to schedule a Low thread on whether a High thread is runnable. Our noninterference formulation makes dom dependent on the current state s, in order to overcome this problem, such that the domain associated with some action a that occurs from state s is dom a s. This makes the resulting noninterference formulation entirely state-dependent and complicates the proofs of soundness for our unwinding conditions. Proving that a system satisfies these unwinding conditions (and therefore our formulation of noninterference) requires showing that the scheduler does not leak information via its scheduling decisions.

Secondly, while phrased for (possibly) nondeterministic systems, our noninterference formulation is preserved by refinement. As explained later, this requires it to preclude all *domain-visible* nondeterminism, which necessarily abstracts away possible sources of information. Being preserved by refinement is important in allowing our noninterference formulation to be proved of real kernels at reasonable cost, as it can be proved about an abstract specification and then transported to the more complex implementation by refinement. In the case of seL4, this allows us to prove noninterference about a mostly-deterministic refinement [13] of its abstract *functional* specification, which its C implementation has been proved to refine [11], in order to conclude it of the implementation. Our experience to date suggests that reasoning about seL4's functional specification requires an order of magnitude less effort than reasoning directly about the implementation [12].

Our proof calculus resembles prior language-based frameworks for proving termination-insensitive confidentiality (and other relational) properties of pro-

grams [1, 2, 4]; however, it is better suited than these frameworks for general-purpose OS kernels. Firstly, our calculus aims not at generality but rather at scalability, which is essential to enable its practical application to entire OS kernels. Secondly, it is explicitly designed for reasoning about systems for which no complete static assignment of memory locations or program variables to security domains exists. As is the case with a general-purpose OS kernel like seL4 that implements a dynamic access control system, whether a memory location is allowed to be read by the currently running thread depends on the access rights that the current thread has, if any, to that location. In a microkernel like seL4 that implements virtual memory, this of course depends on the current virtual memory mappings for the currently running thread. Thus, like the mapping of actions to domains, the mapping of memory locations to domains is also state-dependent in a general-purpose OS kernel. Our proof calculus is tuned to tracking and discharging these kinds of state-dependent proof obligations that arise when reasoning about confidentiality in such a system. These manifest themselves as preconditions on confidentiality statements about individual function calls that we discharge using a monadic Hoare logic and its associated VCG [6]. Our calculus is specially tuned so that this same VCG engine can automate its application, without modification, to automatically prove confidentiality statements for those functions that do not read confidential state (i.e. the vast majority of them), given appropriate user-supplied loop invariants.

Our experience applying this calculus to seL4 suggests that it scales very well to real-world systems. So far we have used it to prove confidentiality for 98% of the functions in the abstract seL4 specification in under 15 person-months. The remaining fraction comprises nondeterministic functions that abstract away from sources of confidential information — i.e. parts of the specification that are too abstract to allow correct reasoning about confidentiality. We are in the process of making these parts of the specification more concrete to produce a refinement of the functional specification, which seL4's C implementation refines in turn, suitable for reasoning about confidentiality [13]. We have already done this and proved confidentiality for the *revoke* system call, which is the kernel's most complex code path. The remaining functions are currently in progress.

In this paper, Section 2 presents our noninterference formulation for OS kernels and its associated unwinding conditions. In Section 3 we present our proof calculus for discharging these unwinding conditions across an entire kernel. Section 4 considers related work before Section 5 concludes. All theorems and definitions in this paper have been generated directly from the interactive theorem prover Isabelle/HOL [14] in which all of our work was carried out.

2 Noninterference

Our noninterference formulation for OS kernels extends von Oheimb's notion of *noninfluence* [21]. We formalise noninterference over de Roever-Engelhardt style *data types* [7], which can be thought of as automata with a supported theory of refinement, allowing us to prove that our noninterference formulation is preserved

under refinement. We first introduce the data type formalism and the notion of refinement, before presenting our noninterference formulation and its associated unwinding conditions. We prove that the unwinding conditions are sound and complete for our noninterference formulation. We explain how our unwinding conditions (and, hence, our noninterference formulation) require us to prove the absence of information leaks through scheduling decisions. Lastly we show how our noninterference formulation is preserved by refinement.

2.1 Data Types and Refinement

We model an OS kernel as a state machine whose transitions include processing an interrupt or exception, performing a system call, and ordinary user-level actions like reading and writing user-accessible memory. We use the terms *event* and *action* interchangeably to refer to an automaton's individual transitions.

A data type automaton A is simply a triple comprising three functions: an initialisation function $\mathsf{Init}_A :: \mathit{state} \Rightarrow \mathit{istate\ set}$ that maps individual *observable* states to sets of corresponding internal states, an internal step relation $\mathsf{IStep}_A :: \mathit{event} \Rightarrow (\mathit{istate} \times \mathit{istate})\ \mathit{set}$, and a final projection function $\mathsf{Fin}_A :: \mathit{istate} \Rightarrow \mathit{state}$ that maps individual internal states to corresponding individual observable states. For a data type A and initial observable state $s :: \mathit{state}$ and sequence of transitions as, let $\mathsf{execution}\ A\ s\ as$ denote the set of observable states that can be reached by A performing as. $\mathsf{execution}\ A\ s\ as$ operates by first applying Init_A to the observable state s to produce a set of corresponding initial internal states. It then computes a set of resulting internal states by repeatedly applying IStep_A to each event a in as in turn to arrive at a set of final internal states. To each of these it applies Fin_A to obtain the set of final observable states.

$$\mathsf{execution}\ A\ s\ as \equiv \mathsf{Fin}_A\ `\ \mathsf{foldl}\ (\lambda S\ a.\ \mathsf{IStep}_A\ a\ ``\ S)\ (\mathsf{Init}_A\ s)\ as$$

Here $R\ ``\ S$ and $f\ `\ S$ are the relational images of the set S under the relation R and function f respectively, and foldl is the standard fold function on lists.

A data type C refines data type A, written $A \sqsubseteq C$, when its behaviours are a subset of A's.

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

2.2 System Model

Let A be an automaton, whose observable state is of type *state*, and s_0::*state* denote the initial observable state from which execution of A begins. Let $\mathsf{reachable}\ s$ denote that observable state s is reachable from s_0:

$$\mathsf{reachable}\ s \equiv \exists\, as.\ s \in \mathsf{execution}\ A\ \mathsf{s}_0\ as$$

As occurs in OS kernels generally, we assume that every event is always enabled.

$$\mathsf{reachable}\ s \longrightarrow (\exists\, s'.\ s' \in \mathsf{execution}\ A\ s\ as)$$

Let the function Step characterise the single-step behaviour of the system:

$$\mathsf{Step}\ a \equiv \{(s,\ s') \mid s' \in \mathsf{execution}\ A\ s\ [a]\}$$

For the information flow policy, we assume a set of security domains and a reflexive relation $\rightsquigarrow$ that specifies the allowed information flows between domains, where $d \rightsquigarrow d'$ implies that information is allowed to flow from domain d directly to d'. Noninterference asserts that no information flows outside of $\rightsquigarrow$ can occur.

For each domain d, let $\overset{d}{\sim}$ be an equivalence relation on observable states, such that $s \overset{d}{\sim} t$ if and only if domain d's state is identical in s and t. Here, d's state will include the user-visible state that d can directly read, but might also include kernel-level state that the kernel might legitimately reveal to d. This relation is sometimes called an *unwinding relation*. When the system transitions directly from state s to state s' and $s \overset{d}{\sim} s'$ for example, domain d has not been observably affected by this transition. For a set of domains D, let $s \overset{D}{\approx} t \equiv \forall\, d{\in}D.\ s \overset{d}{\sim} t$.

Traditional noninterference formulations associate a security domain $\mathsf{dom}\ e$ with each event e that occurs, which defines the domain that performed the event. Recall from Section 1 that when a system call event occurs, the kernel must consult the data structures of the scheduler to determine which thread performed the system call, which will be the thread that is currently active. So events are not intrinsically associated with domains; rather, this association depends on part of the current state of the system which records the currently running domain.

Therefore, let $\mathsf{dom} :: \mathit{event} \Rightarrow \mathit{state} \Rightarrow \mathit{domain}$ be a function such that $\mathsf{dom}\ e\ s$ gives the security domain that is associated with event e in state s. When the scheduler's state is identical in states s and t, we expect that $\mathsf{dom}\ e\ s = \mathsf{dom}\ e\ t$ for all events e. Formally, let $\mathsf{s\text{-}dom} :: \mathit{domain}$ be an arbitrary domain, whose state encompasses that part of the system state that determines which domain is currently active. s-dom stands for *scheduler domain*. Then we assume that for all events e and states s and t

$$s \overset{\mathsf{s\text{-}dom}}{\sim} t \longrightarrow \mathsf{dom}\ e\ s = \mathsf{dom}\ e\ t$$

Actions of the scheduling domain s-dom naturally include all those that schedule a new domain d to execute. We expect that when a domain d is scheduled, that d will be able to detect that it is now active, and so that an information flow might have occurred from s-dom to d. Since the scheduler can possibly schedule any domain, we expect that a wellformed information flow policy $\rightsquigarrow$ will have an edge from s-dom to every domain d:

$$\mathsf{s\text{-}dom} \rightsquigarrow d$$

In order to prevent s-dom from being a global transitive channel by which information can flow from any domain to any other, we require that information can never flow directly from any other domain d to s-dom, so

$$d \rightsquigarrow \mathsf{s\text{-}dom} \longrightarrow d = \mathsf{s\text{-}dom}.$$

This restriction forces us to prove that the scheduler's decisions about which domain should execute next are independent of the other domains, which is typical scheduler behaviour in a separation kernel.

2.3 Formulating Noninterference

Traditionally [16], intransitive noninterference definitions make use of a sources function, whereby for a sequence of actions as and a domain d, sources as d gives the set of domains that are allowed to pass information to d when as occurs. Because our dom function depends on the current state s, sources must do so as well. Therefore let sources as s d denote the set of domains that can pass information to d when as occurs, beginning in state s. The following definition is an extension of the standard one [16,21] in line with our augmented dom function.

$$\begin{aligned}
&\mathsf{sources}\ []\ s\ d &&= \{d\}\\
&\mathsf{sources}\ (a{\cdot}as)\ s\ d &&=\\
&\bigcup\{\mathsf{sources}\ as\ s'\ d \mid (s,\ s') \in \mathsf{Step}\ a\} \cup\\
&\{w \mid w = \mathsf{dom}\ a\ s \wedge (\exists\, v\ s'.\ \mathsf{dom}\ a\ s \rightsquigarrow v \wedge (s,\ s') \in \mathsf{Step}\ a \wedge v \in \mathsf{sources}\ as\ s'\ d)\}
\end{aligned}$$

Here, we include in sources $(a{\cdot}as)$ s d all domains that can pass information to d when as occurs from all successor-states s' of s, as well as the domain dom a s performing a, whenever there exists some intermediate domain v that it is allowed to pass information to who in turn can pass information to d when the remaining events as occur from some successor state s' of s. An alternative, and seemingly more restrictive, definition would include only those domains that are present in all sources as s' d, and include dom a s only when some such v can be found for each sources as s' d, where $(s,\ s') \in \mathsf{Step}\ a$. However, as a consequence of Lemma 2 introduced later, this yields an equivalent noninterference formulation.

As is usual, the sources function is used to define a purge function, ipurge, in terms of which noninterference is formulated. Traditionally, for a domain d and action sequence as, ipurge d as returns the sequence of actions as with all actions removed that are not allowed to (indirectly) influence d when as occurs [16]. Naturally, we must include the current state s in our ipurge function. However, for nondeterministic systems purging may proceed from a set ss of possible initial states. This leads to the following definition.

$$\begin{aligned}
&\mathsf{ipurge}\ d\ []\ ss &&= []\\
&\mathsf{ipurge}\ d\ (a{\cdot}as)\ ss = &&\ \mathsf{if}\ \exists\, s{\in}ss.\ \mathsf{dom}\ a\ s \in \mathsf{sources}\ (a{\cdot}as)\ s\ d\\
&&&\ \mathsf{then}\ a{\cdot}\mathsf{ipurge}\ d\ as\ (\textstyle\bigcup_{s\in ss}\{s' \mid (s,\ s') \in \mathsf{Step}\ a\})\\
&&&\ \mathsf{else}\ \mathsf{ipurge}\ d\ as\ ss
\end{aligned}$$

Initially, the set ss will be a singleton containing one initial state s. Given a sequence of actions $a{\cdot}as$ being performed from s, ipurge will keep the first action a if dom a s $\in$ sources $(a{\cdot}as)$ s d, i.e. if this action is allowed to affect the target domain d. Purging then continues on the remaining actions as from the successor states of s after a. On the other hand, if the action a being performed is not allowed to affect the target domain d, then it is removed from the sequence. For this reason, purging continues on the remaining actions as from the current state s, rather than its successors. We require the action to be able to affect the target domain in only one of the states $s \in ss$ to avoid purging it. An alternative definition would instead place this requirement on all states $s \in ss$. Again however, because of Lemma 2, this yields an equivalent noninterference formulation.

For states s and t and sequences of actions as and bs and domain d, let $\mathsf{uwr\text{-}equiv}\ s\ as\ t\ bs\ d$ denote when the contents of domain d is identical after executing as from s and bs from t in all resulting pairs of states. When $\mathsf{uwr\text{-}equiv}\ s\ as\ t\ bs\ d$ is true, domain d is unable to distinguish the cases in which as is executed from s, and bs is executed from t. Recall that we assume that every event is always enabled and that divergence never occurs on any individual execution step, under the assumption that noninterference is proved only after proving that a system's execution is always defined. This is why uwr-equiv and the following noninterference formulation are termination-insensitive.

$$\mathsf{uwr\text{-}equiv}\ s\ as\ t\ bs\ d \equiv$$
$$\forall\, s'\ t'.\ s' \in \mathsf{execution}\ A\ s\ as \wedge t' \in \mathsf{execution}\ A\ t\ bs \longrightarrow s' \overset{d}{\sim} t'$$

Traditionally [16, 21] this property is defined using a projection function $\mathsf{out} :: \mathit{domain} \Rightarrow \mathit{state} \Rightarrow \mathit{output}$ so that, rather than testing whether $s' \overset{d}{\sim} t'$ for final states s' and t', it tests whether $\mathsf{out}\ d\ s' = \mathsf{out}\ d\ t'$. However, these traditional formulations invariably require the unwinding condition of *output consistency* which asserts that $\mathsf{out}\ d\ s' = \mathsf{out}\ d\ t'$ whenever $s' \overset{d}{\sim} t'$, and construct the remaining unwinding conditions to establish precisely this latter relation. We avoid this indirection by discarding out entirely. One could re-phrase the noninterference formulation here in terms of out if necessary, in which case the addition of output consistency to the unwinding conditions presented here would be sufficient to prove the resulting noninterference property.

We now have the ingredients to express our noninterference formulation, which we derive as follows. Given two action sequences as and bs, a domain d, and an initial state s from which each sequence is executed, if $\mathsf{ipurge}\ d\ as\ \{s\} = \mathsf{ipurge}\ d\ bs\ \{s\}$ then, when all events that are not allowed to affect d are removed from each sequence, they are both identical. So if none of these removed events can actually affect d, we should expect that d cannot distinguish the execution of one sequence from the other, i.e. that $\mathsf{uwr\text{-}equiv}\ s\ as\ s\ bs\ d$ should hold.

However, the only domains that should be able to affect d when as executes here are those in $\mathsf{sources}\ as\ s\ d$. So if s were modified to produce a state t from which bs was executed instead, we should expect $\mathsf{uwr\text{-}equiv}\ s\ as\ t\ bs\ d$ to hold so long as: (1) s and t agree on the state of all domains in $\mathsf{sources}\ as\ s\ d$, i.e. $s \overset{\mathsf{sources}\ as\ s\ d}{\approx} t$, and (2) the same domain is currently active in both, i.e. $s \overset{\mathsf{s\text{-}dom}}{\sim} t$. This is our formulation of von Oheimb's *noninfluence* [21], denoted noninfluence.

$$\mathsf{noninfluence} \equiv$$
$$\forall\, d\ as\ bs\ s\ t.$$
$$\quad \mathsf{reachable}\ s \wedge \mathsf{reachable}\ t \wedge s \overset{\mathsf{sources}\ as\ s\ d}{\approx} t \wedge s \overset{\mathsf{s\text{-}dom}}{\sim} t \wedge$$
$$\quad \mathsf{ipurge}\ d\ as\ \{s\} = \mathsf{ipurge}\ d\ bs\ \{s\} \longrightarrow \mathsf{uwr\text{-}equiv}\ s\ as\ t\ bs\ d$$

Note that, as a consequence of Lemma 1 introduced later, replacing the term $\mathsf{ipurge}\ d\ bs\ \{s\}$ by $\mathsf{ipurge}\ d\ bs\ \{t\}$ here yields an equivalent property.

noninfluence might be too strong a property for systems with a pre-determined static schedule that is fixed for the entire lifetime of the system and known to all domains. If every domain always knows the exact sequence of events that

must have gone before whenever it executes, then purging makes less sense. For these kinds of systems, an analogue of von Oheimb's weaker notion of *nonleakage* might be more appropriate. We denote this property nonleakage.

$$\mathsf{nonleakage} \equiv \forall\, as\ s\ t\ d.\ \mathsf{reachable}\ s \wedge \mathsf{reachable}\ t \wedge s \overset{\mathsf{s\text{-}dom}}{\sim} t \wedge s \overset{\mathsf{sources}\ as\ s\ d}{\approx} t \longrightarrow \mathsf{uwr\text{-}equiv}\ s\ as\ t\ as\ d$$

Naturally, noninfluence implies nonleakage.

2.4 Unwinding Conditions

A standard proof technique for noninterference properties involves proving so-called *unwinding conditions* [16] that examine individual execution steps of the system in question. We introduce two unwinding conditions. The first is sound and complete for nonleakage. The addition of the second to the first is sound and complete for noninfluence.

Both of these conditions examine individual execution steps of the system, and assert that they must all satisfy specific properties. As is usual with noninterference, we would like to conclude that these same properties are true across all runs of the system. However, this rests on the assumption that a run of the system, say in which it performs some sequence of actions *as*, is equivalent to performing a sequence of one-step executions for each of the events in *as* in turn.

This is formalised by the following function Run, which takes a step function *Stepf*, and repeatedly applies it to perform a sequence of actions *as* by executing each action in *as* in turn.

$$\mathsf{Run}\ Stepf\ [] = \{(s, s) \mid \mathsf{True}\}$$
$$\mathsf{Run}\ Stepf\ (a{\cdot}as) = Stepf\ a \circ \mathsf{Run}\ Stepf\ as$$

Like ours, traditional unwinding conditions are predicated on the assumption that $\mathsf{reachable}\ s \longrightarrow \mathsf{execution}\ A\ s\ as = \{s' \mid (s, s') \in \mathsf{Run\ Step}\ as\}$ (assuming naturally that $\mathsf{reachable}\ \mathsf{s}_0$ too). While this is valid for traditional noninterference formulations, in which their execution is defined exactly in this way [21], it is not always true for an arbitrary data-type automaton of the kind introduced in Section 2.1 over which our noninterference properties are defined. However, for most well behaved data types this condition is true, and certainly holds for those that model the seL4 functional specification and its C implementation. Thus we restrict our attention to those automata A that satisfy this assumption and return to our unwinding conditions.

The first unwinding condition is a confidentiality property, while the second is an integrity property. The confidentiality property we denote confidentiality-u, and resembles the conjunction of von Oheimb's *weak step consistency* and *step respect* [21] for deterministic systems; however, we require it to hold for *all* successor states and to take into account the scheduler domain s-dom.

$$\mathsf{confidentiality\text{-}u} \equiv \forall\, a\ d\ s\ t\ s'\ t'.\ \mathsf{reachable}\ s \wedge \mathsf{reachable}\ t \wedge s \overset{\mathsf{s\text{-}dom}}{\sim} t \wedge s \overset{d}{\sim} t \wedge (\mathsf{dom}\ a\ s \rightsquigarrow d \longrightarrow s \overset{\mathsf{dom}\ a\ s}{\sim} t) \wedge (s, s') \in \mathsf{Step}\ a \wedge (t, t') \in \mathsf{Step}\ a \longrightarrow s' \overset{d}{\sim} t'$$

This property says that the contents of a domain d after an action a occurs can depend only on d's contents before a occurred, as well as the contents of the domain $\mathsf{dom}\ a\ s$ performing a if that domain is allowed to send information to d. This condition alone allows d to perhaps infer that a has occurred, but not to learn anything about the contents of confidential domains.

The second unwinding condition is an integrity property, denoted integrity-u, and is essentially Rushby's *local respect* [16] adapted to nondeterministic systems and again asserted for all successor states.

$$\mathsf{integrity\text{-}u} \equiv \forall\, a\ d\ s\ s'.\ \mathsf{reachable}\ s \wedge \mathsf{dom}\ a\ s \not\rightsquigarrow d \wedge (s,\ s') \in \mathsf{Step}\ a \longrightarrow s \overset{d}{\sim} s'$$

It says that an action a that occurs from some state s can affect only those domains that the domain performing the action, $\mathsf{dom}\ a\ s$, is allowed to directly send information to. It prevents any domain d for which $\mathsf{dom}\ a\ s \not\rightsquigarrow d$ from even knowing that a has occurred.

The soundness proofs for these unwinding conditions are slightly more involved than traditional proofs of soundness for unwinding conditions. This is because our sources and ipurge functions are both state-dependent. The following lemma is useful for characterising those states that agree on sources and ipurge, under confidentiality-u, namely those related by $\overset{\mathsf{s\text{-}dom}}{\sim}$.

Lemma 1. $\mathsf{confidentiality\text{-}u} \wedge \mathsf{reachable}\ s \wedge \mathsf{reachable}\ t \wedge s \overset{\mathsf{s\text{-}dom}}{\sim} t \longrightarrow$
$\mathsf{sources}\ as\ s\ d = \mathsf{sources}\ as\ t\ d \wedge \mathsf{ipurge}\ d\ as\ \{s\} = \mathsf{ipurge}\ d\ as\ \{t\}$

With this result, the proofs of the soundness of our unwinding conditions are similar to those for traditional non-state-dependent formulations of noninterference, since (as we explain shortly) confidentiality-u guarantees that the equivalence $\overset{\mathsf{s\text{-}dom}}{\sim}$, asserted by noninfluence and nonleakage, is always maintained. The completeness proofs for these unwinding conditions are straightforward.

Theorem 1 (Soundness and Completeness).

$\mathsf{nonleakage} = \mathsf{confidentiality\text{-}u}$, *and* $\mathsf{noninfluence} = (\mathsf{confidentiality\text{-}u} \wedge \mathsf{integrity\text{-}u})$

2.5 Scheduling

We said that our noninterference formulation requires us to show that the scheduler's choices are independent of the other domains. To see why, consider when the domain d from our unwinding conditions is s-dom. Then confidentiality-u implies that s-dom can never be affected by the state of the other domains:

$$\forall\, a\ s\ t\ s'\ t'.\ \mathsf{reachable}\ s \wedge \mathsf{reachable}\ t \wedge s \overset{\mathsf{s\text{-}dom}}{\sim} t \wedge (s,\ s') \in \mathsf{Step}\ a \wedge (t,\ t') \in \mathsf{Step}\ a \longrightarrow s' \overset{\mathsf{s\text{-}dom}}{\sim} t'$$

Thus confidentiality-u implies that $\overset{\mathsf{s\text{-}dom}}{\sim}$ is always maintained.

When $\mathsf{dom}\ a\ s \neq \mathsf{s\text{-}dom}$, $\mathsf{dom}\ a\ s \not\rightsquigarrow \mathsf{s\text{-}dom}$. So integrity-u implies that the scheduler domain can never be affected by the actions of the other domains:

$$\forall\, a\ s\ s'.\ \mathsf{reachable}\ s \wedge \mathsf{dom}\ a\ s \neq \mathsf{s\text{-}dom} \wedge (s,\ s') \in \mathsf{Step}\ a \longrightarrow s \overset{\mathsf{s\text{-}dom}}{\sim} s'$$

2.6 Refinement

We now show that noninfluence and nonleakage are preserved by refinement. This means we can prove them of an abstract specification A and conclude that they hold for all concrete implementations C that refine it (i.e. for which $A \sqsubseteq C$).

Theorem 2 (noninfluence and nonleakage are Refinement-Closed).
When $A \sqsubseteq C$, $\mathsf{noninfluence}_A \longrightarrow \mathsf{noninfluence}_C$, *and* $\mathsf{nonleakage}_A \longrightarrow \mathsf{nonleakage}_C$

Proof. We will prove that each unwinding condition is closed under refinement, which implies that their conjunction is as well. The result then follows from Theorem 1. Let A and C be two automata, and write Step_A, $\mathsf{sources}_A$ etc. for those respective functions applied to A and similarly for C. Then, when $A \sqsubseteq C$, C's executions are a subset of A's, so $\mathsf{reachable}_C\ s \longrightarrow \mathsf{reachable}_A\ s$ and $\mathsf{Step}_C\ a \subseteq \mathsf{Step}_A\ a$. It is straightforward to show then that $\mathsf{integrity\text{-}u}_A \longrightarrow \mathsf{integrity\text{-}u}_C$ and $\mathsf{confidentiality\text{-}u}_A \longrightarrow \mathsf{confidentiality}\text{-}u_C$, as required. □

As mentioned earlier, a consequence of being preserved by refinement is that our unwinding conditions tolerate very little nondeterminism. Specifically, if the unwinding conditions hold, a system must have no *domain-visible* nondeterminism, which is nondeterminism that can be observed by any domain. This is because any such nondeterminism could abstract from a confidential source of information that is present in a refinement, and so implies the existence of insecure refinements. The following lemma states this restriction formally.

Lemma 2 (No Visible Nondeterminism).

$$\mathsf{confidentiality\text{-}u} \wedge \mathsf{reachable}\ s \wedge (s, s') \in \mathsf{Step}\ a \wedge (s, s'') \in \mathsf{Step}\ a \longrightarrow s' \overset{d}{\sim} s''$$

3 A Proof Calculus for Confidentiality for State Monads

Having explained our noninterference formulation, and in particular its unwinding conditions, we now present a proof calculus for discharging these unwinding conditions across an OS kernel. We have successfully applied this calculus to the seL4 microkernel, as part of ongoing work to prove that it enforces our noninterference formulation.

Our proof calculus operates over nondeterministic state monads, the formalism that underpins the seL4 abstract functional specification. Specifically, the internal steps of the automaton that embodies the seL4 functional specification are formalised as computations of a nondeterministic state monad. The state type of this monad is simply the internal state of the automaton, which for the seL4 functional specification is also identical to its observable state. The unwinding condition integrity-u asserts that the state before a single execution step is related to each final state after the execution step. It is naturally phrased as a Hoare triple, and discharged using standard verification techniques. For seL4, we have used a monadic Hoare logic and its associated verification condition generator (VCG) [6] to discharge this condition [17]. This leaves just the property confidentiality-u. It is this property that our confidentiality proof calculus addresses.

3.1 Nondeterministic State Monad

To prove confidentiality for an entire kernel specification, we need to be able to decompose it across that specification to make verification tractable. It is this challenge that our proof calculus addresses for nondeterministic state monads.

The type for this nondeterministic state monad is

$$state \Rightarrow (\alpha \times state)\ set \times bool$$

That is, it is a function that takes a state s as its sole argument, and returns a pair p. The first component $\mathsf{fst}\ p$ is a set of pairs $(rv,\ s')$, where rv is a return-value and s' is the result state. Each such pair $(rv,\ s')$ represents a possible execution of the monad. The presence of more than one element in this set implies that the execution is nondeterministic. The second part $\mathsf{snd}\ p$ of the pair returned by the monad is a boolean flag, indicating whether at least one of the computations has failed. Since our confidentiality property is termination insensitive, this flag can be ignored for the purpose of this paper.

Our proof calculus for confidentiality properties over this state monad builds upon the simpler proof calculus for Hoare triples [6] mentioned above. In this calculus, a precondition P is a function of type $state \Rightarrow bool$, i.e. a function P such that, a given state s satisfies P if and only if $P\ s$ is true. Since a monad f returns a set of return-value/result-state pairs, a postcondition Q is a function of type $\alpha \Rightarrow state \Rightarrow bool$. Q may be viewed as a function that given a return-value rv and corresponding result-state s', tells whether they meet some criteria. Alternatively, Q may be viewed as a function that, given some return-value rv, yields a state-predicate $Q\ rv$ that tests validity of the corresponding result-state s'. We write such Hoare triples as $\{\!|P|\!\}\ f\ \{\!|Q|\!\}$, and define their meaning as follows.

$$\{\!|P|\!\}\ f\ \{\!|Q|\!\} \equiv \forall\, s.\ P\ s \longrightarrow (\forall\, (rv,\ s') \in \mathsf{fst}\ (f\ s).\ Q\ rv\ s')$$

The proof calculus for Hoare triples of our nondeterministic state monad includes a mechanical rule application engine that acts as a VCG for discharging Hoare triples [6]. Later we discuss how we can apply this same engine to act as a VCG for discharging our confidentiality properties.

3.2 Confidentiality over State Monads

Observe that the property, confidentiality-u, addressed by our confidentiality proof calculus considers two *pre-states*, s and t, for which some equivalences hold, and then asserts that for all *post-states*, s' and t', another equivalence holds. We formalise this for our nondeterministic state monad, generalising over the pre- and post-state equivalences, as the property $\mathsf{ev}\ A\ B\ P\ f$, pronounced *equivalence valid*. Here, A and B are pre-state and post-state equivalence relations respectively (often called just the *pre-equivalence* and *post-equivalence* respectively), f is the monadic computation being executed and P is a precondition that the pre-states s and t are assumed to satisfy.

$$\mathsf{ev}\ A\ B\ P\ f \equiv$$
$$\forall\, s\ t.\ P\ s \wedge P\ t \wedge A\ s\ t \longrightarrow (\forall\, (r_a,\ s') \in \mathsf{fst}\ (f\ s).\ \forall\, (r_b,\ t') \in \mathsf{fst}\ (f\ t).\ r_a = r_b \wedge B\ s'\ t')$$

Note that $\mathsf{ev}\ A\ B\ P\ f$ also asserts that the return values from both executions of f are equal. This requires that these return-values be derived only from those parts of the system state that are identical between the two executions (i.e. those parts that the pre-equivalence A guarantees are identical). The purpose of the precondition P is to allow extra conditions under which the pre-equivalence A guarantees that confidentiality is satisfied. For instance, if f is a function that reads a region of user memory, the precondition P might include a condition that ensures that this region is covered by the pre-equivalence A.

To decompose this property across a monadic specification, we need to define proof rules for the basic monad operators, return and $\gg\!=$ (pronounced "bind"). $\mathsf{return}\ x$ is the state monad that leaves the state unchanged and simply returns the value x. This means that if A holds for the pre-states, then A will hold for the post-states as well. Also, $\mathsf{return}\ x$ always returns the same value (x) when called. This gives us the following proof rule.

$$\frac{}{\mathsf{ev}\ A\ A\ (\lambda\text{-}.\ \mathsf{True})\ (\mathsf{return}\ x)}\ \textsc{return-ev}$$

Note that this rule restricts the post-equivalence to be the same as the pre-equivalence. As we explain shortly, this turns out not to be a problem in practice.

$f \gg\!= g$ is the composite computation that runs f, and then runs g on the result, and is used to sequence computations together. Specifically, $f \gg\!= g$ runs the computation f to obtain a return value rv and result state s', and then calls $g\ rv$ to obtain a second computation that is run on the state s'. Because f might be nondeterministic, $f \gg\!= g$ does this for all pairs $(rv,\ s')$ that f emits, taking the distributed union over all results returned from each $g\ rv\ s'$.

Because we want to be able to decompose the proof of ev across a specification, we need a proof rule for $f \gg\!= g$ that allows us to prove ev for f and g separately, and then combine the results to obtain a result overall. The following proof rule, BIND-EV, does exactly that.

$$\frac{\forall\, rv.\ \mathsf{ev}\ B\ C\ (Q\ rv)\ (g\ rv) \qquad \mathsf{ev}\ A\ B\ P'\ f \qquad \{\!|P''|\!\}\ f\ \{\!|Q|\!\}}{\mathsf{ev}\ A\ C\ (P'\ \mathsf{and}\ P'')\ (f \gg\!= g)}\ \textsc{bind-ev}$$

Here, $P'\ \mathsf{and}\ P''$ is the conjunction of preconditions P' and P'', i.e. $P'\ \mathsf{and}\ P'' \equiv \lambda s.\ P'\ s \wedge P''\ s$. BIND-EV can be read as a recipe for finding a precondition $?P$ such that $\mathsf{ev}\ A\ C\ ?P\ (f \gg\!= g)$ is true. First, for any return value rv that f might emit, find some state-equivalence B and a precondition $Q\ rv$, which may mention rv, such that $g\ rv$ yields post-states that satisfy the post-equivalence C. Secondly, find a precondition P' such that executing f yields post-states that satisfy the just found state-equivalence B. Finally, find a precondition P'', such that for all return-values rv emitted from executing f, their corresponding result-states satisfy $Q\ rv$. The desired precondition $?P$ is then $P'\ \mathsf{and}\ P''$.

This rule works because if ev is true for f, we know that both executions of f yield the same return-value, say rv, which means that the two subsequent executions both run the same computation, $g\ rv$. In the rare case that ev cannot be proved for f (say because f returns a value rv derived from confidential state), a more sophisticated rule is required that we introduce later in Section 3.4.

3.3 Automating Confidentiality Proofs

Note that when $C = A$, we may define a simpler variant of BIND-EV, called BIND-EV', in which B and C are both A.

$$\frac{\forall rv.\ \mathsf{ev}\ A\ A\ (Q\ rv)\ (g\ rv) \qquad \mathsf{ev}\ A\ A\ P'\ f \qquad \{\!|P''|\!\}\ f\ \{\!|Q|\!\}}{\mathsf{ev}\ A\ A\ (P'\ \mathsf{and}\ P'')\ (f >\!\!>\!\!= g)}\ \text{BIND-EV'}$$

To apply this rule, we need only compute sufficient preconditions $Q\ rv$, P' and P'' for the relevant obligations. Our ordinary Hoare logic VCG can be applied to compute P'', of course, while BIND-EV' is itself a recipe for computing sufficient $Q\ rv$ and P'. In other words, we may recursively apply BIND-EV' to compute appropriate $Q\ rv$ and P', given appropriate ev rules for the primitive monadic functions.

This is precisely the technique that we have taken to prove these statements across the seL4 functional specification. Specifically, at the top-level, the pre-equivalence of confidentiality-u, asserted for s and t, implies the post-equivalence, $\overset{d}{\sim}$, asserted for all s' and t', because the pre-equivalence includes $\overset{d}{\sim}$. Hence, if we prove that the pre-equivalence is preserved, we can deduce that the post-equivalence must hold after each kernel event. This allows us to reason about a more restricted version of ev in which the pre- and post-equivalences are always identical, using rules like RETURN-EV and BIND-EV' above.

The rule-application engine developed previously [6] that acts as a VCG for Hoare triples over our nondeterministic state monad, can be applied directly without any modification to discharge ev statements by feeding it the appropriate rules. It requires rules like BIND-EV', to decompose these proofs into smaller goals, as well as appropriate rules, like RETURN-EV, to discharge the goals at the leaves of the proof tree. Familiar rules from prior work on proof methods for relational properties of programs [1, 2, 4] may be derived for other monadic functions, such as the one below for monadic while-loops. It establishes confidentiality for the loop under the invariant P when the loop body B maintains confidentiality and the pre-equivalence A guarantees that both executions terminate together. The loop body B and condition C are both parametrised by a loop parameter n, which for subsequent loop iterations is the return-value of the previous iteration.

$$\frac{\begin{array}{c}\forall s\ t\ n.\ A\ s\ t \wedge P\ n\ s \wedge P\ n\ t \longrightarrow C\ n\ s = C\ n\ t \\ \forall n.\ \{\!|P\ n\ \mathsf{and}\ C\ n|\!\}\ B\ n\ \{\!|P|\!\} \qquad \forall n.\ \mathsf{ev}\ A\ A\ (P\ n\ \mathsf{and}\ C\ n)\ (B\ n)\end{array}}{\mathsf{ev}\ A\ A\ (P\ n)\ (\mathsf{whileLoop}\ C\ B\ n)}\ \text{WHILE-EV}$$

3.4 Proving the Functions that Read Confidential State

The approach so far allows very automatic proofs for functions that do not read any confidential state, and so always yield identical return-values rv. Because these functions make up the bulk of seL4, this is what our calculus has been tuned for. However, it is less well suited to functions that operate on confidential state without revealing it to unauthorised domains. Our approach requires confidentiality proofs for these kinds of functions to be performed more manually.

$$\frac{\forall s\ t.\ P\ s \wedge P'\ t \wedge A\ s\ t \longrightarrow R\ x\ y}{\mathsf{ev2}\ A\ A\ R\ P\ P'\ (\mathsf{return}\ x)\ (\mathsf{return}\ y)}\ \textsc{return-ev2}$$

$$\frac{\begin{array}{l}\forall rv\ rv'.\ R'\ rv\ rv' \longrightarrow \mathsf{ev2}\ B\ C\ R\ (Q\ rv)\ (Q'\ rv')\ (g\ rv)\ (g'\ rv') \\ \quad \mathsf{ev2}\ A\ B\ R'\ P\ P'\ f\ f' \qquad \{\!|S|\!\}\ f\ \{\!|Q|\!\} \qquad \{\!|S'|\!\}\ f'\ \{\!|Q'|\!\}\end{array}}{\mathsf{ev2}\ A\ C\ R\ (P\ \mathsf{and}\ S)\ (P'\ \mathsf{and}\ S')\ (f >\!\!>\!\!= g)\ (f' >\!\!>\!\!= g')}\ \textsc{bind-ev2}$$

Fig. 1. VCG rules for ev2

An example is the seL4 function send-async-ipc, which sends a message on an *asynchronous endpoint*. Asynchronous endpoints facilitate unidirectional communication, which implies that the act of sending on an asynchronous endpoint should not leak any information back to the sender. Sending such a message does require the kernel to read state outside of the sending domain (such as state in the endpoint); however, it should not reveal any of this state to the sender.

There is no guarantee, then, that when the two executions of send-async-ipc that ev compares each read the internal state of the endpoint in question, they will get the same result. This means their subsequent executions might behave differently to each other. Proving that ev holds in this case requires comparing two different executions, and showing that they establish the post-equivalence. This suggests that we should reason about a more general property than ev that can talk about two different executions, and allows return-values to differ.

These insights lead to the following property, called ev2.

$$\begin{array}{l}\mathsf{ev2}\ A\ B\ R\ P\ P'\ f\ f' \equiv \\ \forall s\ t.\ P\ s \wedge P'\ t \wedge A\ s\ t \longrightarrow \\ \quad (\forall (r_a,\ s') \in \mathsf{fst}\ (f\ s).\ \forall (r_b,\ t') \in \mathsf{fst}\ (f'\ t).\ R\ r_a\ r_b \wedge B\ s'\ t')\end{array}$$

ev2 takes two computations, f and f', and two associated preconditions, P and P'. It also takes a return-value relation R, that it asserts holds for the return-values of f and f'. ev2 generalises ev, specifically $\mathsf{ev}\ A\ B\ P\ f \equiv \mathsf{ev2}\ A\ B\ \mathsf{op} = P\ P\ f\ f$, where op = is the equality operator. One usually applies this equivalence to rewrite ev goals that cannot be proved by the VCG, into ev2 goals. One then manually applies proof rules like in Figure 1 to discharge these goals.

Applying BIND-EV2 usually requires the human to come up with an appropriate intermediate return-value relation R' that will hold for the return values emitted from f and f'. As with ev, we usually work with a simpler rule in which $(B = C) = A$, which we omit for brevity. We suspect that techniques could be borrowed from other work on automatically proving confidentiality properties of programs [2, 18] to help automatically infer appropriate R'. However, because ev2 proofs are seldom required for seL4, we have not needed to implement them.

4 Related Work

Recently, Barthe et al. [3] presented a formalisation of isolation for an *idealised* model of a hypervisor, and its unwinding conditions. Like ours, their definition

is based on von Oheimb's noninfluence [21]. As in traditional formalisations of noninterference, in their formulation actions are intrinsically linked to domains, and so it cannot reason about information leaks through scheduling decisions.

INTEGRITY-178B is a real-time operating system for which an isolation proof has been completed [15]. The isolation property proved is based on the GWVr2 information flow property [9], which bears similarities to the unwinding conditions for noninterference. Like ours, it is general enough to handle systems in which previous execution steps affect which is the entity that executes next. Unlike ours, it is defined only for deterministic systems. The exact relationship between GWVr2 and our conditions deserves further study.

Our formulation of information flow security is descendant from traditional *ipurge*-based formulations of intransitive noninterference (starting with Haigh and Young's [10]). Van der Meyden [19] argues that ipurge-based formulations of intransitive noninterference are too weak for certain intransitive policies, and proposes a number of stronger definitions. He shows that Rushby's unwinding conditions [16] are sufficient for some of these alternatives. Given the similarity of our unwinding conditions to Rushby's, we wonder whether our existing unwinding conditions may be sufficient to prove analogues of van der Meyden's definitions.

Others have presented noninterference conditions for systems with scheduling components. One recent example is van der Meyden and Zhang [20], who consider systems that run in lock-step with a scheduling component that controls which domain's actions are currently enabled. Their security condition for the scheduler requires that the actions of the High domain cannot affect scheduling decisions. Our formulation, in contrast, has the scheduler update a component of the system state that determines the currently running domain. This allows our scheduler security condition to require that scheduling decisions be unaffected not only by domain actions, but also by domain state.

A range of proof calculi and verification procedures for confidentiality properties, and other *relational* properties, have also been developed [1,2,4,5,18]. Unlike many of these, ours aims not at generality but rather at scalability. The simplicity of our calculus has enabled it to scale to the entire functional specification of the seL4 microkernel, whose size is around 2,500 lines of Isabelle/HOL, and whose implementation that refines this specification is around 8,500 lines of C.

5 Conclusion

We have presented a definition of noninterference for operating system kernels, with sound and complete unwinding conditions. We have shown how these latter can be implemented in a proof calculus for nondeterministic state monads with automation support. Our success in applying both of these to the seL4 microkernel, in an ongoing effort to prove that it enforces noninterference, attest to their practical utility and applicability to programs on the order of 10,000 lines of C.

Acknowledgements We thank Kai Engelhardt, Sean Seefried, and Timothy Bourke for their comments on earlier drafts of this paper.

References

1. T. Amtoft and A. Banerjee. Information flow analysis in logical form. In *SAS '04*, volume 3148 of *LNCS*, pages 33–36. Springer-Verlag, 2004.
2. T. Amtoft and A. Banerjee. Verification condition generation for conditional information flow. In *FMSE '07*, pages 2–11. ACM, 2007.
3. G. Barthe, G. Betarte, J. Campo, and C. Luna. Formally verifying isolation and availability in an idealized model of virtualization. In M. Butler and W. Schulte, editors, *17th FM*, volume 6664 of *LNCS*, pages 231–245. Springer-Verlag, 2011.
4. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL 2004*, pages 14–25. ACM, 2004.
5. L. Beringer. Relational decomposition. In *2nd ITP*, volume 6898 of *LNCS*, pages 39–54. Springer-Verlag, 2011.
6. D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In *21st TPHOLs*, volume 5170 of *LNCS*, pages 167–182, Aug 2008.
7. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
8. J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. Security & Privacy*, pages 11–20, Oakland, California, USA, Apr 1982. IEEE.
9. D. A. Greve. Information security modeling and analysis. In D. S. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 249–300. Springer-Verlag, 2010.
10. J. T. Haigh and W. D. Young. Extending the noninterference version of MLS for SAT. *Trans. Softw. Engin.*, 13:141–150, Feb 1987.
11. 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. ACM, 2009.
12. G. Klein, T. Murray, P. Gammie, T. Sewell, and S. Winwood. Provable security: How feasible is it? In *13th HotOS*, pages 28–32, Napa, CA, USA, May 2011. USENIX.
13. D. Matichuk and T. Murray. Extensible specifications for automatic re-use of specifications and proofs. In *10th SEFM*, Oct 2012.
14. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
15. 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.
16. J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, Dec 1992.
17. T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. seL4 enforces integrity. In *2nd ITP*, volume 6898 of *LNCS*, pages 325–340, Nijmegen, The Netherlands, Aug 2011. Springer-Verlag.
18. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS '05*, volume 3672 of *LNCS*, pages 352–367. Springer-Verlag, 2005.
19. R. van der Meyden. What, indeed, is intransitive noninterference? In *12th ESORICS*, volume 4734 of *LNCS*, pages 235–250. Springer-Verlag, 2007.
20. R. van der Meyden and C. Zhang. Information flow in systems with schedulers. In *21st CSF*, pages 301–312. IEEE, Jun 2008.
21. D. von Oheimb. Information flow control revisited: Noninfluence = noninterference + nonleakage. In *9th ESORICS*, volume 3193 of *LNCS*, pages 225–243, 2004.