Dynamic Intransitive Noninterference

Rebekah Leslie
Portland State University

***Abstract*— Existing noninterference frameworks for reasoning about system security assume a fixed configuration of domains governed by a security policy that does not change over time. A static security policy, however, cannot express the domain interactions present in many modern system designs, which allow users to configure the set of active domains at run-time. In this paper, we generalize Rushby's framework for static noninterference to support reasoning about the security of systems with a dynamic policy. In particular, we show that the security of a given dynamic policy system can be established by supplementing Rushby's unwinding theorem with the new notion of *policy respect*.**

I. Introduction

Reasoning about computer systems is a persistent challenge that is increasingly relevant as computers become a ubiquitous part of our lives. To verify that a system behaves correctly, designers must control how information flows through the system. Informal arguments suffice in some situations, but high-security applications require a mathematically rigorous treatment.

A standard method for controlling information flow is to divide a system into *protection domains* that represent distinct components of the system. Typically these domains correspond to processes or address spaces. Permissible interactions between domains are expressed as a *security policy*. Myriad kinds of security policies exist, including access control [1], multilevel security [2], channel-control [3], and noninterference [4]. The noninterference interpretation, introduced by Goguen and Meseguer [4], is the most general of these, because it is capable of expressing the other three. A noninterference policy specifies which domains may not *interfere* with each other, where a domain, u, interferes with a domain, v, if v can observe the effects of u's execution.

Protection domains and security policies allow us to express the desired information flow properties of a system, but to guarantee that a system is secure we must prove that the system enforces a particular policy. Noninterference formulations—which provide a framework for reasoning about policy enforcement—address this need. There are many such formulations, which vary widely in their power and generality.

The foundational work of Rushby [5] is the basis of the noninterference framework described in this paper. Rushby developed two formulations of noninterference: transitive noninterference, which assumes a transitive security policy, and intransitive noninterference, which does not. He showed how an access control interpretation of security—specifically that of Bell and La Padula [1]—could be specified using transitive noninterference, and showed that intransitive noninterference could express channel-control policies. Rushby also proved that transitive noninterference is a special case of intransitive noninterference.

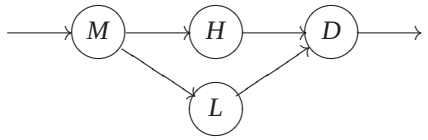
A common characteristic of the existing formulations is that they assume the security policy does not change over time. This assumption is incompatible with modern system designs, which rely on the ability to install and remove components of the system at run-time. A static security policy is incapable of expressing the interactions between domains in such a setting. Even when the set of active domains does not change, a system administrator may wish to alter the policy during execution in response to changing security requirements.

Consider the example of a simplified webserver in Figure 1(a). Domain M receives encrypted packets from the network and distributes them to domains H and L based on the header information. Domain H processes classified packets and domain L processes unclassified packets. Once the processing is complete, domains H and L send a response to the client via domain D. To ensure that no sensitive information is leaked to unclassified clients, there must be separation between domains H and L. Intransitive noninterference formulations are designed to describe the security of systems like this, but what if we wish to upgrade domain H without disrupting service to the clients? We achieve this by introducing two new domains: H' replaces H and C is a configuration task that copies state information from H to H'. During the configuration of H', classified packets are still processed by H, but the policy is changed to allow information flow from H to C and from C to H', as shown in Figure 1(b). We complete the installation of H' by modifying the policy to reflect the replacement of H with H'. Figure 1(c) depicts the final policy.

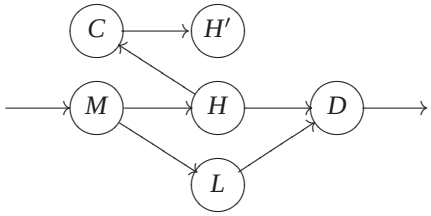
Note that the policy edge from H to C is removed in the final policy, but C might still contain classified information it gained access to under the policy of Figure 1(b). If the same configuration task is later used to upgrade L, then there is a path for information to flow from classified clients to unclassified ones. Unfortunately, we cannot reason about how information flows across the policy change using the existing noninterference formulations.

In this paper, we generalize the intransitive noninterference formulation of Rushby [5] to systems whose security policy changes over time. The result is a framework for reasoning about security that accounts for the dynamic nature of modern systems.

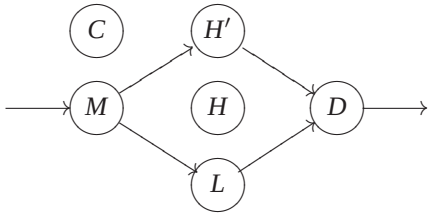
Our interest in dynamic policy systems stems from our efforts to reason about high-level executable models of operating systems as part of the Programatica project [6]. The principal focus of this work is the development of a Haskell [7] implementation of the L4 microkernel [8], [9], with ver-



(a) Initial policy.



(b) Policy during the installation of H'



(c) Alteration of the policy to reflect replacement of H with H'.

Fig. 1. Example of dynamic policy changes in a simplified webserver. We represent policy using a graph where nodes correspond to protection domains. An edge between domains *M* and *H* indicates that *M* interferes with *H*.

ified separation between domains. Many situations requiring dynamic policy changes arise in the development of L4-based systems. For example, the (potentially temporary) installation of monitors for debugging, logging, and security purposes is common practice in L4 [10].

We organize the remainder of the paper as follows. Section II reviews Rushby's intransitive noninterference framework [5]. In Section III, we develop dynamic intransitive noninterference by generalizing this framework to systems with a dynamic policy. Section IV introduces two refinements of noninterference for fine-grained interference control. Section V describes related work. Section VI presents our conclusions and plans for future work. The appendix contains the proofs of our main results, which we omit from the body of the paper to increase readability.

II. Static Noninterference

This section summarizes Rushby's framework for reasoning about system security [5]. The basis of this framework is an abstract model of computer systems and a representation of noninterference policies. Rushby uses these foundational elements to define security, to formalize assumptions, and to prove that a system satisfying these assumptions is secure.

A. System Model

Rushby [5] models computer systems as state machines. An action is a state transformer that also produces some output. The behavior of the system is specified by the functions *step* and *output*; *step* performs a single state transition whereas *output* extracts the result of the action.

Definition 1: A *system* (*machine*), M, consists of

- a set of *states*, S, with an initial state $s_0 \in \mathrm{S}$,
- a set of *actions*, A, and
- a set of *outputs*, O,

together with execution functions,

- *step* :: $\mathrm{S} \times \mathrm{A} \to \mathrm{S}$, and
- *output* :: $\mathrm{S} \times \mathrm{A} \to \mathrm{O}$. □

To express security constraints on the system, Rushby supplements the basic execution model with protection domains and a security policy. Actions in the system are partitioned into a set of domains, *D*, as specified by the function $dom :: A \to D$. A reflexive binary relation on domains, $\rightsquigarrow$ (pronounced *interferes*), expresses the security policy of the system.

$$\rightsquigarrow :: D \times D \to Bool$$

A domain *u* interferes with a domain *v* if information is allowed to flow from *u* to *v*, meaning that *v* can observe the effects of *u*'s actions. The symbol $\not\rightsquigarrow$ (pronounced *does not interfere*) represents the complement relation.

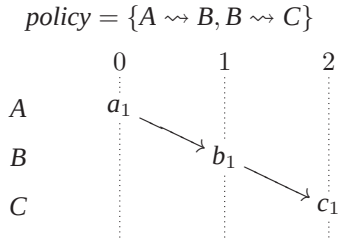
While the specific nature of states is abstract, the security definition requires some mechanism for determining what information is observable to a particular domain in each state. To this end, Rushby introduces an equivalence relation, $\sim :: S \times S \times D \to Bool$, called the *view-partitioning relation*. Two states, *s* and *t*, are equivalent from the perspective of a domain *u*, written $s \overset{u}{\sim} t$, if *u* cannot distinguish *s* and *t*, The precise meaning of $\sim$ is a parameter to the formulation. This relation extends naturally to an equivalence, $\approx$, over a set of domains, *C*.

$$s \overset{C}{\approx} t \equiv \forall u \in C.\ s \overset{u}{\sim} t$$

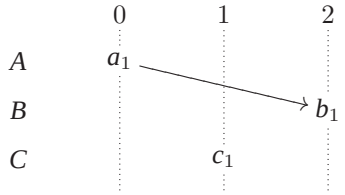
B. Characterizing Domain Interactions

A security policy specifies which domains are allowed to interfere, but does not capture all information flow between domains. For example, consider a system with three domains, *A*, *B*, and *C*, and a policy $\{A \rightsquigarrow B, B \rightsquigarrow C\}$, as pictured in Figure 2. Though the policy does not state that *A* interferes with *C*, information can flow from *A* to *C* indirectly through actions in *B*. If $\rightsquigarrow$ is transitive, then *A* interferes with *C* in all situations. However, we are concerned with systems where $\rightsquigarrow$ is not assumed to be transitive, so we need a strategy for determining if information flow from *A* to *C* occurs in a particular sequence of actions. *A* only interferes with *C* through *B*, so an action in *A* only interferes with an action in *C* when an intervening action in *B* is present.

Rushby captures information flow between domains formally with a function called *sources*. Given a domain, *u*, and a sequence of actions, α, $sources(\alpha, u)$ is the set of domains that might leak information to *u* when α executes. In this paper, we use the notation A^* to represent the set of finite sequences whose elements belong to the set *A*. The empty sequence is denoted by $[]$ and the sequence obtained by prepending *a* to *as* is denoted by $(a{:}as)$. The notation $\wp(D)$ represents the set of possible subsets of the set *D*.



(a) a_1 interferes with c_1 via intervening action b_1



(b) a_1 cannot interfere with c_1

Fig. 2. Intervening actions allow indirect interference between domains. *A*, *B*, and *C* are domains in a system. The policy contains $A \rightsquigarrow B$ and $B \rightsquigarrow C$. In the diagram, columns represent execution steps, where the numeric label corresponds to the state an action executes in (e.g., a_1 executes in s_0 producing s_1). An arrow between two actions indicates interference.

Definition 2: We define a function, *sources*, which determines via what domains information can flow to a particular domain *u*.

$$
\begin{array}{lcl}
\mathit{sources} & :: & A^* \times \mathrm{D} \rightarrow \wp(D) \\
\mathit{sources}([],\, u) & = & \{\mathrm{u}\} \\
\mathit{sources}(a{:}as,\, u) & = & \\
\left\{ \begin{array}{l} \mathit{sources}(as, u) \cup \{\mathit{dom}(a)\} \\ \quad \text{if } \exists v.\ v \in \mathit{sources}(as, u) \\ \qquad \wedge\ \mathit{dom}(a) \rightsquigarrow v \\ \mathit{sources}(as, u) \\ \quad \text{otherwise} \end{array} \right. &&
\end{array}
$$

Given a non-empty sequence of actions, (*a*:*as*), *sources* checks whether *a* directly interferes with an action in $\mathit{sources}(as, u)$. This condition is true if an intervening action in *as* allows information to flow from $\mathit{dom}(a)$ to *u*, or if the security policy allows direct interference between $\mathit{dom}(a)$ and *u*. □

Security under a noninterference policy means that actions not allowed to interfere with a domain, *u*, are truly unobservable to *u*. Rushby captures this interpretation of security by simulating system execution. Executing a sequence of actions should produce the same results, from the perspective of *u*, as executing the same sequence with non-interfering actions removed.

The function *ipurge* removes all actions that do not interfere with a particular domain from a sequence of actions. The purge process relies on *sources* to analyze whether each action in the sequence may leak information to the domain in question.

Definition 3: We define a function, *ipurge*, which, given a list of actions, α, and a domain, *u*, returns a list containing the actions in α that interfere with *u*.

$$
\begin{array}{lcl}
\mathit{ipurge} & :: & A^* \times \mathrm{D} \rightarrow A^* \\
\mathit{ipurge}([],\, u) & = & [] \\
\mathit{ipurge}(a : as,\, u) & = & \\
\left\{ \begin{array}{ll} a{:}\mathit{ipurge}(as, u) & \text{if } \mathit{dom}(a) \in \mathit{sources}(a{:}as, u) \\ \mathit{ipurge}(as, u) & \text{otherwise} \end{array} \right. &&
\end{array}
$$

When $\alpha = a{:}as$, *ipurge* uses *sources* to determine if *a* interferes with *u*. If not, *a* is purged. □

Rushby models the execution of a sequence of actions with *run*, which is a natural extension of *step* from Section II-A.

$$
\begin{array}{lcl}
\mathit{run} & :: & \mathrm{S} \times A^* \rightarrow \mathrm{S} \\
\mathit{run}(s,\, []) & = & \mathrm{s} \\
\mathit{run}(s,\, (a{:}as)) & = & \mathit{run}(\mathit{step}(s,a),\, as)
\end{array}
$$

We often wish to run a sequence of actions from the initial state, s_0, or to extract the output produced by an action following such an execution. The functions *do* and *test* are shorthands for these two operations.

$$
\begin{array}{lcl}
\mathit{do} & :: & A^* \rightarrow S \\
\mathit{do}(\alpha) & = & \mathit{run}(s_0,\, \alpha) \\
 & & \\
\mathit{test} & :: & A^* \times \mathrm{A} \rightarrow O \\
\mathit{test}(\alpha,\, a) & = & \mathit{output}(\mathit{do}(\alpha),\, a)
\end{array}
$$

We now use these functions to state Rushby's formulation of system security:

Definition 4 (Security Property): Let *s* be the state that results from running an arbitrary sequence of actions, α, from the initial state, and let *t* be the state that results from running the corresponding purged list. A system is secure if the output produced by executing any action is the same in *s* and *t*.

$$\mathit{test}(\alpha, a) = \mathit{test}(\mathit{ipurge}(\alpha, \mathit{dom}(a)), a)$$

□

C. Establishing Security

Rushby employs a technique called unwinding to prove that a system satisfies the security property. Many noninterference formulations use this approach [11], [12], [13]. Unwinding establishes that a system with well-behaved state transitions is secure. A set of *unwinding conditions* specify the expected properties of state transitions, and the *unwinding theorem* links these conditions to the security property.

Rushby specifies three unwinding conditions—output consistency, local respect, and weak step consistency—and proves that the security property holds for systems that satisfy these conditions.

Definition 5 (Output Consistency): A system is *output consistent* if the output produced by executing an action in equivalent states is the same in both states.

$$s \overset{\mathit{dom}(a)}{\sim} t \ \Rightarrow\ \mathit{output}(s, a) = \mathit{output}(t, a)$$

□

Definition 6 (Local Respect): A system *locally respects* the security policy if the effect of an action, *a*, which may not interfere with a domain, *u*, is unobservable to *u*. That is, the state produced by executing *a* is equivalent, from *u*'s perspective, to the state before *a* executed.

$$\mathit{dom}(a) \not\rightsquigarrow u \ \Rightarrow\ s \overset{u}{\sim} \mathit{step}(s, a)$$

□

Definition 7 (Weak Step Consistency): A system is *weakly step consistent* if the states that result from executing an action in equivalent states are equivalent.

$$s \overset{u}{\sim} t \wedge s \overset{dom(a)}{\sim} t \;\Rightarrow\; step(s,a) \overset{u}{\sim} step(t,a)$$

□

Theorem 1 (Unwinding): Let $\rightsquigarrow$ be a policy and M a view-partitioned system that is

1) output consistent,
2) weakly step consistent, and
3) locally respects $\rightsquigarrow$.

Then M is secure for $\rightsquigarrow$.

Rushby proves the unwinding theorem using properties of *sources* and $\approx$, the details of which are in his paper [5].

III. Dynamic Noninterference

The key contribution of this paper is the development of a framework for reasoning about security in dynamically configured systems. We analyze how dynamic policies impact information flow between domains, formalize the meaning of interference in the dynamic setting, and revise the definition of security from Section II-B accordingly. We introduce a modified set of unwinding conditions and prove that these conditions are sufficient to establish that our new definition of security holds.

A. System Model

In the dynamic setting, there is no longer a single security policy, rather the policy is potentially different in each state. To account for this, we supplement the model of Section II-A with a set of policies and link policies to states. Each state contains a policy component, which represents the active security policy for that state. Interference between two domains becomes a function of state, so we introduce a dynamic version of $\rightsquigarrow$: *interferes* :: $D \times D \times S \rightarrow Bool$. We refer to the systems described by this model as *dynamic policy systems*.

B. Interference in a Dynamic Framework

Information can flow between domains in intricate ways when the security policy changes over time. In particular, the effects of interference allowed under one policy may be observable long after that policy has been revoked.

Consider the example in Figure 3. Initially, the policy permits A to interfere with B. In s_3, the policy changes such that A may not interfere with B, but B may interfere with C. Although the security policy never contains $A \rightsquigarrow B$ and $B \rightsquigarrow C$ simultaneously, information leakage from A to C is possible through actions in B.

Cross-policy information flow also results from the reflexive property of $\rightsquigarrow$, as Figure 4(a) illustrates. In states s_0–s_2, the policy disallows all interference. In s_3, the policy becomes $A \rightsquigarrow B$. Though action a_1 executes under the original policy—which does not permit information flow from A to B—a_1 can still influence action b_2 through action a_2. Thus, introducing a policy of the form $u \rightsquigarrow v$ permits all actions in u, even

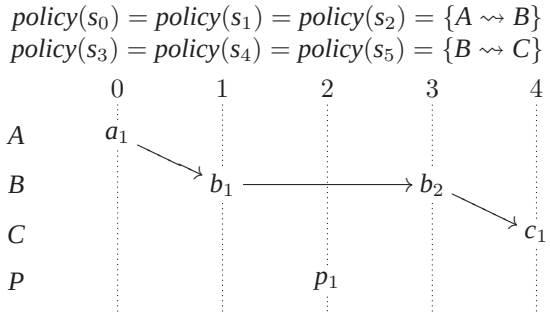
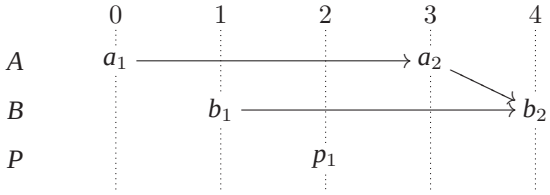
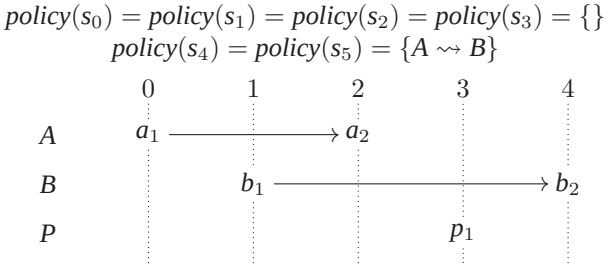


Fig. 3. Indirect interference across policy boundaries. Action a_1 may leak information to B because it executes under the policy $A \rightsquigarrow B$. Action b_2, which has access to information leaked by a_1, executes under the policy $B \rightsquigarrow C$. Thus information can flow from a_1 to c_1 via b_2. P is a domain responsible for changing the policy. Though P interferes with all domains in the system, we omit these relationships from the diagram to avoid clutter.



(a) Role of reflexive policies in indirect interference. Action a_1 influences action b_2 even though a_1 executes under a policy which does not permit information flow from A to B. The interference relationship between a_1 and a_2 enables this cross-policy interference to occur.



(b) Role of state in indirect interference. In the dynamic setting, the policy in effect when an intervening action executes determines whether this action enables indirect interference.

Fig. 4. Impact of policy on interference relationships.

those that occur before the policy change, to influence v. Note that this behavior is different than with a static policy $A \rightsquigarrow B$, because actions in A do not affect actions in B that execute before s_3.

With a static security policy, indirect interference relationships depend only on the presence of an intervening action. In the dynamic setting, the state in which the intervening action executes is also a factor. Figure 4(b) reprises the example of Figure 4(a) with one minor alteration; action a_2 now executes *before* the policy $A \rightsquigarrow B$ takes effect. In this case, no intervening action in A occurs after the policy change, so *no* actions in A influence b_2.

C. Characterizing Domain Interactions

As in the static setting, *sources* determines how information flows between domains when a sequence of actions executes. To account for policy changes during execution and for the impact that these changes can have on information flow, we extend *sources* with a state argument. Given a domain, *u*, a sequence of actions, α, and a state, *s*, *sources* calculates the set of domains that may leak information to *u* when α executes from *s*. While the definition is superficially similar to the static version, the meaning of indirect interference that it captures is very different.

Definition 8: We define a function *sources* that determines via what domains information can flow to a particular domain *u*.

$$
\begin{array}{lcl}
\mathit{sources} & :: & A^* \times \mathrm{D} \times \mathrm{S} \to \wp(D) \\
\mathit{sources}([],\, u,\, s) & = & \{\mathrm{u}\} \\
\mathit{sources}(a{:}as,\, u,\, s) & = & \\
\multicolumn{3}{l}{\left\{\begin{array}{ll}
\mathit{sources}(as,u,t) \cup \{\mathit{dom}(a)\} & \\
\quad \text{if } \exists v.\ v \in \mathit{sources}(as,u,t) & \\
\qquad \wedge\ \mathit{interferes}(\mathit{dom}(a),v,s) & \\
\mathit{sources}(as,u,t) & \\
\quad \text{otherwise} &
\end{array}\right.} \\
\multicolumn{3}{l}{\text{where } t = \mathit{step}(s,a)}
\end{array}
$$

Given a non-empty sequence of actions, (*a*:*as*), to be executed from state *s*, *sources* checks whether *a* directly interferes, under the policy of *s*, with a domain in $\mathit{sources}(as, u, \mathit{step}(s,a))$. □

We formalize the expected behavior of *sources* as:

Property 1: The definition of *sources* obeys the following properties.

(a) $u \in \mathit{sources}(\alpha, u, s)$
(b) $\mathit{dom}(a) \notin \mathit{sources}(a{:}as, u, s) \wedge v \in \mathit{sources}(as, u, \mathit{step}(s,a)) \Rightarrow \neg \mathit{interferes}(\mathit{dom}(a), v, s)$
(c) $\mathit{sources}(as, u, \mathit{step}(s,a)) \subseteq \mathit{sources}(a : as, u, s)$
(d) $s \overset{u}{\sim} t \ \Rightarrow\ \mathit{sources}(\alpha, u, s) = \mathit{sources}(\alpha, u, t)$

Properties 1(a) and 1(b) verify that *sources* accurately captures interference relationships between domains. Property 1(c) establishes that the result of *sources* grows with the length of the sequence of actions being evaluated. Property 1(d) captures the relationship between $\sim$ and *sources*, ensuring that *sources* behaves consistently on equivalent states. Each of these properties follow directly from the definition of *sources*.

In formulating noninterference, we frequently use expressions of the form $s \overset{\mathit{sources}(\alpha,u,s)}{\approx} t$, to indicate that the information that *u* may access (through interference by a domain in $\mathit{sources}(\alpha, u, s)$) is the same in states *s* and *t*. Property 2 characterizes the interaction between *sources* and $\approx$ in such expressions.

Property 2: The relation $\approx$ obeys the following properties.

(a) $\forall \alpha. s \overset{\mathit{sources}(\alpha,u,s)}{\approx} t \ \Rightarrow\ s \overset{u}{\sim} t$
(b) $s \overset{\mathit{sources}(\alpha,u,s)}{\approx} t \ \Rightarrow\ s \overset{\mathit{sources}(\alpha,u,t)}{\approx} t$
(c) $s \overset{\mathit{sources}(a:as,u,s)}{\approx} t \ \Rightarrow\ s \overset{\mathit{sources}(as,u,\mathit{step}(s,a))}{\approx} t$

Recall from section II-C that *ipurge* removes non-interfering actions from an action sequence. We name the corresponding function in the dynamic setting *dipurge*, for *dynamic ipurge*. The dynamic framework does not necessitate any fundamental changes to the filtration mechanism, but, of course, the reliance of *sources* on state information requires *dipurge* to be a function of state as well.

Definition 9: We define the function *dipurge* as:

$$
\begin{array}{lcl}
\mathit{dipurge} & :: & A^* \times \mathrm{D} \times \mathrm{S} \to A^* \\
\mathit{dipurge}([],\, u,\, s) & = & [] \\
\mathit{dipurge}(a{:}as,\, u,\, s) & = & \\
\multicolumn{3}{l}{\left\{\begin{array}{ll}
a : \mathit{dipurge}(as,u,t) & \text{if } \mathit{dom}(a) \in \mathit{sources}(a{:}as,u,s) \\
\mathit{dipurge}(as,u,t) & \text{otherwise}
\end{array}\right.} \\
\multicolumn{3}{l}{\text{where } t = \mathit{step}(s,a)}
\end{array}
$$

Given a list of actions, α, to be executed from state *s*, and a domain *u*, $\mathit{dipurge}(\alpha, u, s)$ is a list containing the actions in α that may interfere with *u*. □

Security under a dynamic noninterference policy has the same meaning as security under a static policy—a domain, *u*, cannot observe the effects of actions not allowed to interfere with *u*. The only difference is that we must revise the definition of security to use the dynamic filter function, *dipurge*.

Definition 10 (Dynamic Security Property): A dynamic policy system is secure if, for arbitrary α and any action, *a*:

$$\mathit{test}(\alpha, a) = \mathit{test}(\mathit{dipurge}(\alpha, \mathit{dom}(a), s_0), a).$$

□

Introducing state dependency into the definition of *dipurge* makes verifying the security property more complicated. In particular, we must establish not only that *dipurge* correctly filters a list of actions, but also that *dipurge* behaves consistently given equivalent states.

Property 3: Given a list of actions, α, and a domain, *u*, the filtered lists that *dipurge* produces are identical for equivalent states.

$$s \overset{\mathit{sources}(\alpha,u,s)}{\approx} t \ \Rightarrow\ \mathit{dipurge}(\alpha, u, s) = \mathit{dipurge}(\alpha, u, t)$$

Note that this property requires the assumption $s \overset{\mathit{sources}(\alpha,u,s)}{\approx} t$, rather than $s \overset{u}{\sim} t$, as one might expect. This is because the proof of Property 3 depends on weak step consistency, which requires the stronger assumption.

D. Establishing Security

Output consistency and weak step consistency do not depend on the nature of security policies, so they are still relevant under the new system model. Local respect, on the other hand, is a property of policy enforcement, so it must be modified to account for dynamic policies. The intended meaning remains the same—if an action does not interfere with a domain, *u*, then the effects of the action are unobservable in *u*—but the question of whether the action interferes with *u* is now a function of the state. Thus, the hypothesis of static local respect, $\mathit{dom}(a) \not\rightsquigarrow u$, becomes $\neg \mathit{interferes}(\mathit{dom}(a), u, s)$.

Definition 11 (Dynamic Local Respect): If an action *a*, executed in state *s*, belongs to a domain which may not interfere

with u under the policy of s, then u cannot distinguish the state before a executes from the one afterwards.

$$\neg \mathit{interferes}(\mathit{dom}(a), u, s) \;\Rightarrow\; s \overset{u}{\sim} \mathit{step}(s, a)$$

□

Dynamic policy induces one further unwinding condition, *policy respect*, which is a property of $\sim$ that makes explicit the relationship between policies in equivalent states. Equivalence remains abstract, but—at a minimum—the set of domains allowed to interfere with u must be the same in states equivalent from the perspective of u.

Definition 12 (Policy Respect): If two states, s and t are equivalent from the perspective of domain u, then exactly the same set of domains interfere with u under the policies of s and t. The following formula expresses this requirement.

$$s \overset{u}{\sim} t \;\Rightarrow\; (\mathit{interferes}(v, u, s) \Leftrightarrow \mathit{interferes}(v, u, t))$$

□

Logically, the unwinding conditions extend to equivalences over sets of domains. Modifying output consistency and policy respect in this way is trivial because the properties are implied by the definition of $\approx$. Weak step consistency and dynamic local respect do not follow so naturally. We capture the extended version of these properties in Lemmas 1 and 2 respectively.

Lemma 1: If M is a view-partitioned system that satisfies weak step consistency, dynamic local respect, and policy respect, then

$$s \overset{\mathit{sources}(a:as,u,s)}{\approx} t \;\Rightarrow\; \mathit{step}(s, a) \overset{\mathit{sources}(as,u,\mathit{step}(s,a))}{\approx} \mathit{step}(t, a).$$

Defining step consistency over $\approx$ is straightforward. If states s and t are equivalent from the perspective of all domains in $\mathit{sources}(a{:}as, u, s)$, then the states that result from executing a are equivalent from the perspective of all domains in $\mathit{sources}(as, u, \mathit{step}(s, a))$. While this seems intuitive, intricacies in the proof arise from the presence of $\approx$ on both sides of the implication. Unlike with output consistency and policy respect, we cannot merely apply the definition of $\approx$ and the original property. The state dependent nature of *interferes* is a particular source of difficulty in the proof, which led to the introduction of policy respect as an unwinding condition.

Lemma 2: If M is a view-partitioned system that satisfies dynamic local respect, then

$$\mathit{dom}(a) \notin \mathit{sources}(a : as, u, s) \;\Rightarrow\; s \overset{\mathit{sources}(as,u,\mathit{step}(s,a))}{\approx} \mathit{step}(s, a).$$

Local respect over sets of domains means that no domain that interferes with u can observe the effect of an action that does not interfere with u. This property is surprisingly strong; how does the noninterference relationship between $\mathit{dom}(a)$ and u guarantee dynamic local respect for other domains? Recall the definition of *sources*: if $\mathit{dom}(a)$ interferes with a domain $v \in \mathit{sources}(as, u, \mathit{step}(s, a))$, then information can leak from $\mathit{dom}(a)$ to u via an action in v, so $\mathit{dom}(a) \in \mathit{sources}(a{:}as, u, s)$. Thus, if the hypothesis of the lemma holds, then a does not interfere with any domain in $\mathit{sources}(as, u, \mathit{step}(s, a))$. The conclusion then follows from the original definition of dynamic local respect.

Establishing the correctness of *dipurge* is the crux of the unwinding theorem proof. Informally, we wish to verify that the list of actions not purged from the perspective of domain u contains exactly the actions that may influence u's behavior. Lemma 3 formalizes this interpretation of correctness. We simulate program execution from two equivalent states, s and t. If u cannot distinguish the result of running the original list of actions starting in s from running the purged list starting in t, then *dipurge* behaves correctly. The use of equivalent states, rather than the same state, in the simulated executions guarantees that the behavior of *dipurge* does not depend in any way on data unobservable to u.

Lemma 3: If M is a system that satisfies weak step consistency, dynamic local respect, and policy respect, then

$$s \overset{\mathit{sources}(\alpha,u,s)}{\approx} t \;\Rightarrow\; \mathit{run}(s, \alpha) \overset{u}{\sim} \mathit{run}(t, \mathit{dipurge}(\alpha, u, t)).$$

The impact of state-dependencies in *sources* and *dipurge* is most keenly observable in the proof of Lemma 3 (see the Appendix). While the proof mirrors the overall structure of the corresponding lemma by Rushby [5], many of the intermediate results are not immediately useful due to state disagreements. Rectifying these disagreements relies on the well-behaved nature of *sources*, $\approx$, and *dipurge*. Thus, the key steps in the proof are appropriate applications of Properties 1, 2, and 3.

Note that Lemma 3 bears striking similarity to the noninfluence property of von Oheimb [13]. Noninfluence is a more general version of noninterference, where security is defined in terms of system executions from equivalent states, not the initial state. By adjusting for minor differences between our system model and the model used by von Oheimb, we can express noninfluence as

$$s \overset{\mathit{sources}(\alpha,u,s)}{\approx} t \;\Rightarrow\; (\mathit{test}(\alpha, a) = \mathit{test}(\mathit{dipurge}(\alpha, \mathit{dom}(a), s_0), a)).$$

This property follows directly from Lemma 3 and output consistency.

Our final lemma links the global security property to the correctness of *dipurge*. We use the same notion of correctness expounded in Lemma 3, but specialize it to executions that start in the initial state.

Lemma 4: If M is a view-partitioned, output consistent system such that

$$\forall \alpha.\mathit{do}(\alpha) \overset{u}{\sim} \mathit{do}(\mathit{dipurge}(\alpha, u, s_0)),$$

then

$$\mathit{test}(as, a) = \mathit{test}(\mathit{dipurge}(as, \mathit{dom}(a), s_0), a).$$

The conclusion follows directly from output consistency and from the definition of *test*. In contrast to Lemma 3, the proof of Lemma 4 is completely isolated from the differences between the static and dynamic formulations, and the proof steps are identical to the corresponding lemma of Rushby [5].

We extend the static unwinding theorem in a straightforward manner to incorporate the revised unwinding conditions.

Theorem 2 (Dynamic Unwinding): If M is a view-partitioned dynamic policy system that

1) is output consistent,
2) is weakly step consistent,
3) satisfies dynamic local respect, and
4) satisfies policy respect,

then

$$test(as, a) = test(dipurge(as, dom(a), s_0), a).$$

The static unwinding theorem proof carries over to the dynamic case with only minor changes. We specialize Lemma 3 by substituting $s = t = s_0$. The reflexive property of $\approx$ and the definition of *do* simplify this result to $do(as) \overset{u}{\sim} do(dipurge(as, u, s_0))$, and the conclusion follows from Lemma 4.

Initially, it seems surprising that the proof does not require more significant modification. The insulation of the unwinding theorem proof from formulation changes stems from the abstraction that the lemmas provide. Details concerning interference relationships, purging, and execution are all hidden below the surface.

IV. Fine-Grained Interference Control

This section outlines two potential refinements of our dynamic noninterference framework: action-specific interference and time-based interference. The goal of these refinements is to provide system designers with more control over information flow relationships than is possible with traditional noninterference policies.

a) Action-Specific Interference: The abstract nature of actions in our framework leads to an extremely conservative interpretation of interference. We assume that every action that potentially leaks information to another action does so, though this is often not the case in real systems. If we give more structure to actions, we can relax the definition of interference and obtain more accurate knowledge about information flow in the system.

As an example, consider a simple operating system that only permits information flow between domains via an interprocess communication (IPC) system call. An action, a, is either a local action or an IPC message to a particular domain, u. IPC messages leak information from $dom(a)$ to u, while local actions only interfere with $dom(a)$. Figure 5 shows a system configuration with three domains, A, B, and C, running under the policy $\{A \rightsquigarrow B, B \rightsquigarrow C\}$. If b_1 is a local action, then no information flows from A to C. Indirect interference between A and C only occurs if b_1 is an IPC message to C.

To integrate the action-specific interpretation of interference into our framework, we supplement the model of Section III-A with an action classification function *actionInterferes* :: $A \times D \rightarrow Bool$. Given an action, a, and a domain, u, $actionInterferes(a, u)$ is true if a is an action that can cause information flow from $dom(a)$ to u. This is a static property of actions, independent of policy. We alter the definition of *sources* to incorporate *actionInterferes* into the calculation of interference relationships:

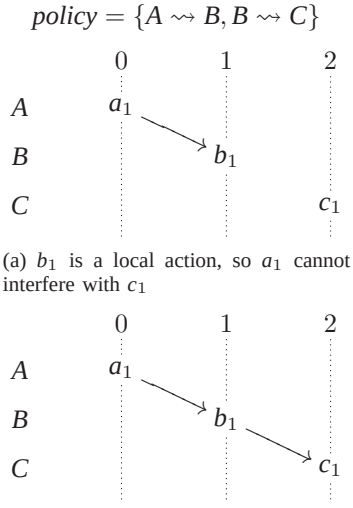


Fig. 5. Action-specific interference. Information flow between A and C depends on the kind of action performed by b_1.

$$
\begin{array}{lll}
sources & :: & A^* \times \mathrm{D} \times \mathrm{S} \rightarrow \wp(D) \\
sources([],\, u,\, s) & = & \{\mathrm{u}\} \\
sources(a{:}as,\, u,\, s) & = & \\
\left\{\begin{array}{l}
\quad sources(as, u, t) \cup \{dom(a)\} \\
\quad\quad \text{if } \exists v.\ v \in sources(as, u, t) \\
\quad\quad\quad\quad \wedge\ interferes(dom(a), v, s) \\
\quad\quad\quad\quad \wedge\ actionInterferes(a, v) \\
\quad sources(as, u, t) \\
\quad\quad \text{otherwise}
\end{array}\right. & & \\
\text{where } t = step(s, a) & &
\end{array}
$$

The signature of *sources* remains the same, so adding the *actionInterferes* constraint does not lead to modifications of *dipurge* or the security property. Lemmas 2 and 4 remain valid, because the proofs of these lemmas do not rely on *sources* in any way. The proofs of Lemmas 1 and 3, on the other hand, must be revised to account for the new definition of *sources*. We expect the revision of these proofs will be straightforward, though it may require an additional unwinding condition, akin to local respect and policy respect, that expresses the expected system behavior when a local action executes.

b) Time-Based Interference: As illustrated in Section III-B, the effects of interference between two domains, u and v, are observable long after the information flow from u to v takes place, even if the policy changes to disallow interference between u and v. This interpretation of interference does not account for situations where the information u shares with v is only valid for a fixed period of time. For example, u might grant v temporary access to classified information by sending v a cryptographic key that will be changed the next day.

We would like to incorporate time-based interference into the dynamic noninterference framework, but this requires extensive changes. In particular, the system model and the interpretation of information flow must be modified to reflect the passage of time. Further investigation is necessary to

determine if such modifications are feasible.

V. Related Work

Bevier and Young [11] provide a state-based model of noninterference as an alternative to I/O-based models, such as the Rushby framework. In the state-based approach, actions are an implicit part of each domain. A function *view*—which returns the portion of the state observable to a domain—replaces *output*. Bevier and Young show that one can express the transitive and intransitive formulations of Rushby in their framework. Conversely, the Rushby framework can be used to express the state-based formulation by instantiating *output* with *view*. Thus, the two modelling styles are equally powerful, and the choice between them is a matter of aesthetics.

The noninterference formulation of von Oheimb [13] is another candidate for extension to the dynamic setting. Von Oheimb extends the Rushby framework to nondeterministic systems and supplements noninterference with the notion of *nonleakage*. Nonleakage complements noninterference by exposing information flow, and these two concepts are integrated into a security definition called *noninfluence*. As mentioned in Section III-D, noninfluence captures essentially the property of Lemma 3.

The system model used by von Oheimb contains several distinctions from the system model presented here, the most interesting being the removal of the requirement that $\sim$ is an equivalence relation. This approach—based on the observation by Mantel that $\sim$ need not be symmetric [14], nor reflexive and transitive [15]—is more general, but it complicates the formulation, and the added power is not necessary in many contexts. Despite the differences between the two frameworks, recasting dynamic noninterference to use von Oheimb's approach seems feasible and is an interesting topic for future work.

Some authors question the relevance of noninterference as an approach to security, suggesting that it is too impractical to be applied to real systems [16]. A major concern is that the view-partitioning relation is overly flexible and that this flexibility might lead one to instantiate the relation in a way that does not capture the intended information flow properties. The freedom to choose the view-partitioning relation does present challenges; however, it is also what makes general noninterference frameworks so powerful. Systems with differing security requirements require different notions of information flow. Noninterference is applicable to a large class of systems, regardless of their definition of information flow, because the view-partitioning relation is a parameter to the formulation.

Recent success employing noninterference in the verification of smart cards provides evidence of its relevance and practicality. Von Oheimb used his framework to analyze the security of the Infineon SLE66 smart card processor [13]. Schellhorn et al. applied Rushby's work [5] to prove security properties of their generic formal model of operating systems for multiapplicative smart cards [17]. While noninterference may not be the right paradigm for all applications, these case studies demonstrate that noninterference formulations can reasonably be applied in practice.

VI. Conclusions and Future Work

In this paper, we extended the intransitive noninterference work of Rusbhy to systems with a dynamic security policy. We provided an analysis of information flow that accounts for policy changes and formalized the results of this analysis. The modified interpretation of interference led us to introduce a new unwinding condition, policy respect, which specifies the relationship between state equivalence and policy. We also introduced two refinements of our framework: action-specific interference and time-based interference.

We proved that the dynamic security property holds for systems that satisfy output consistency, weak step consistency, dynamic local respect, and policy respect. Surprisingly, we were able to migrate Rushby's proof of unwinding to the dynamic setting with little modification, because the impact of our extension is hidden in supporting lemmas. The Appendix contains the full proofs of these lemmas.

In the future, we intend to apply the dynamic noninterference framework to real systems. We are particularly interested in using the framework to support the formal security analysis of separation kernels. We also plan to pursue the refinements of noninterference presented in Section IV.

Acknowledgments

We would like to thank James Hook, Mark Jones, Andrew Tolmach, and Peter White for useful discussion and for their suggestions regarding the presentation of this paper. We would also like to thank the anonymous reviewers for their helpful comments.

References

[1] D. Bell and L. LaPadula, "Secure computer systems: Unified exposition and Multics interpretation," The Mitre Corporation, Tech. Rep. ESD-TR-75-306, March 1976.

[2] ——, "Secure computer systems: Mathematical foundations and model," The Mitre Corporation, Tech. Rep. ESD-TR-73-278-2, November 1973.

[3] J. Rushby, "Design and verification of secure systems," in *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)*, vol. 15, no. 5, 1981, pp. 12–21. [Online]. Available: citeseer.ist.psu.edu/rushby81design.html

[4] J. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[5] J. Rushby, "Noninterference, transitivity, and channel-control security policies," Tech. Rep., December 1992. [Online]. Available: http://www.csl.sri.com/papers/csl-92-2/

[6] "The Programatica Project home page," http://www.cse.ogi.edu/PacSoft/projects/programatica/, 2002.

[7] S. Peyton Jones, Ed., *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[8] *L4 eXperimental Kernel Reference Manual*, L4ka Team, January 2005. [Online]. Available: http://l4hq.org/docs/manuals/l4-x2-20041209.pdf

[9] "The L4 μ-kernel family," http://os.inf.tu-dresden.de/L4/.

[10] T. Jaeger, J. E. Tidswell, A. Gefflaut, Y. Park, J. Liedkte, and K. Elphinstone, "Synchronous IPC over transparent monitors," in *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding Denmark, Sept.17–20 2000.

[11] W. R. Bevier and W. D. Young, "A state-based approach to noninterference." in *Computer Security Foundations Workshop*, 1994, pp. 11–21.

[12] J. Goguen and J. Meseguer, "Unwinding and inference control," in *IEEE Symposium on Security and Privacy*, 1984, pp. 75–86.

[13] D. von Oheimb, "Information flow control revisited: Noninfluence = Noninterference + Nonleakage," in *Computer Security – ESORICS 2004*, ser. LNCS, vol. 3193. Springer, 2004, pp. 225–243.

[14] H. Mantel, "Unwinding Possibilistic Security Properties," in *European Symposium on Research in Computer Security (ESORICS)*, ser. LNCS 1895, F. Cuppens, Y. Deswarte, D. Gollmann, and M. Waidner, Eds. Toulouse, France: Springer, October 4-6 2000, pp. 238–254.

[15] ——, "A uniform framework for the formal specification and verification of information flow security," Ph.D. dissertation, Universität des Saarlandes, Saarbrücken, Germany, July 2003.

[16] J. Millen, "20 years of covert channel modeling and analysis," in *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999, pp. 113–114.

[17] G. Schellhorn, W. Reif, A. Schairer, P. A. Karger, V. Austel, and D. Toll, "Verification of a formal security model for multiapplicative smart cards," in *ESORICS '00: Proceedings of the 6th European Symposium on Research in Computer Security*. London, UK: Springer-Verlag, 2000, pp. 17–36.

APPENDIX

This appendix contains detailed proofs for many of the results given in the body of this paper. For convenience, we repeat the statement of each result in a box at the beginning of the corresponding proof.

A. Properties of dipurge

Property 3. Let M be a view partitioned system that satisfies weak step consistency, local respect, and policy respect. Then,

$$s \overset{sources(\alpha,u,s)}{\approx} t \;\Rightarrow\; dipurge(\alpha,u,s) = dipurge(\alpha,u,t)$$

By induction over $length(\alpha)$. The base case follows directly from the definition of *dipurge*.

$dipurge([],u,s)$
= $[]$ {def. dipurge}
= $dipurge([],u,t)$ {def. dipurge}

In the inductive case, $\alpha = a : as$, we must consider two situations.

Case $dom(a) \in sources(a{:}as,u,s)$: Under the case assumption, the definition of *dipurge* states that

$$dipurge(a : as,u,s) = a : dipurge(as,u,step(s,a)).$$

We apply Lemma 1 to the hypothesis $s \overset{sources(a:as,u,s)}{\approx} t$ to obtain $step(s,a) \overset{sources(u,u,step(s,a))}{\approx} step(t,a)$. Then, the inductive hypothesis provides $a : dipurge(as,u,step(t,a))$. From Property 1(d), we know $dom(a) \in sources(a{:}as,u,t)$, so the definition of *dipurge* yields $dipurge(a{:}\ as,u,t)$.

Case $dom(a) \notin sources(a{:}as,u,s)$: The definition of *dipurge* and the case assumption provide

$$dipurge(a : as,u,s) = dipurge(as,u,step(s,a)).$$

Lemma 1 and $s \overset{sources(a:as,u,s)}{\approx} t$ establish $step(s,a) \overset{sources(u,u,step(s,a))}{\approx} step(t,a)$. Applying the inductive hypothesis to $dipurge(as,u,step(s,a))$ yields $dipurge(as,u,step(t,a))$, which equals $dipurge(a{:}as,u,t)$ by the definition of *dipurge*, the case assumption, and Property 1(d).

This completes the proof. □

B. Unwinding Theorem

This section contains proofs of the unwinding theorem and its central lemmas. We derive much of the proof structure from the corresponding lemmas in Section 4 of Rushby's presentation [5]. The differences stem from our revised definitions of *sources* and *dipurge*, particularly that both functions are state dependent in the dynamic formulation.

Lemma 1. If M is a view-partitioned system that satisfies weak step consistency, local respect, and policy respect, then

$$s \overset{sources(a:as,u,s)}{\approx} t \;\Rightarrow\; step(s,a) \overset{sources(as,u,step(s,a))}{\approx} step(t,a).$$

Let v be an arbitrary domain in $sources(as,u,step(s,a))$. By Property 1(c), $v \in sources(a{:}\ as,u,s)$, from which we derive $s \overset{v}{\sim} t$ using the definition of $\approx$ and $s \overset{sources(a:as,u,s)}{\approx} t$. We now tackle two cases to account for the interference relationship between $dom(a)$ and v.

Case $interferes(dom(a),v,s)$: From the $v \in sources(as,u,step(s,a))$, the definition of *sources*, and the case assumption, we derive $dom(a) \in sources(a{:}as,u,s)$. The definition of $\approx$ and $s \overset{sources(a:as,u,s)}{\approx} t$ then provides $s \overset{dom(a)}{\sim} t$. Combining this result with $s \overset{u}{\sim} t$ allows us to obtain $step(s,a) \overset{v}{\sim} step(t,a)$ by weak step consistency.

Case $\neg interferes(dom(a),v,s)$: The case assumption leads to $s \overset{v}{\sim} step(s,a)$ by local respect. Using this equivalence, the fact that $\sim$ is an equivalence relation, and the earlier derivation of $s \overset{v}{\sim} t$, we obtain $step(s,a) \overset{v}{\sim} t$. Furthermore, $s \overset{v}{\sim} t$ and the case assumption imply $\neg interferes(dom(a),v,t)$ by policy respect. Then, $t \overset{v}{\sim} step(t,a)$ by local respect, and $step(s,a) \overset{v}{\sim} step(t,a)$ by transitivity of $\sim$.

Having proved $s \overset{sources(a:as,u,s)}{\approx} t \;\Rightarrow\; step(s,a) \overset{v}{\sim} step(t,a)$ for an arbitrary $v \in sources(as,u,step(s,a))$, we apply $\forall$ introduction and the definition of $\approx$ to reach the conclusion. □

Lemma 2. If M is a view-partitioned system that satisfies local respect, then

$$dom(a) \notin sources(a : as,u,s) \;\Rightarrow\; s \overset{sources(as,u,step(s,a))}{\approx} step(s,a).$$

Let v be an arbitrary domain in $sources(as,u,step(s,a))$. Property 1(b) and the hypothesis provide $\neg interferes(dom(a),v,s)$. We apply local respect to get $s \overset{v}{\sim} step(s,a)$. Then,

$$\forall v \in sources(as,u,step(s,a)).s \overset{v}{\sim} step(s,a)$$

by $\forall$ introduction. Applying the definition of $\approx$ gives the conclusion. □

Lemma 3. If M is a system that satisfies weak step consistency, local respect, and policy respect, then

$$s \overset{sources(\alpha,u,s)}{\approx} t \;\Rightarrow\; run(s,\alpha) \overset{u}{\sim} run(t, dipurge(\alpha,u,t)).$$

By induction on length(α). The base case, $\alpha = []$, is easily established by definition applications.

$$\begin{aligned}
& s \overset{sources([],u,s)}{\approx} t \\
&= \forall v \in sources([],u,s).s \overset{v}{\sim} t && \{\text{def. } \approx\} \\
&= \forall v \in [u].s \overset{v}{\sim} t && \{\text{def. sources}\} \\
&\Rightarrow s \overset{u}{\sim} t && \{\forall \text{ elimination}\} \\
&\Rightarrow run(s,[]) \overset{u}{\sim} run(t,[]) && \{\text{def. run}\} \\
&\Rightarrow run(s,[]) \overset{u}{\sim} run(t, dipurge([],u,t)) && \{\text{def. dipurge}\}
\end{aligned}$$

In the inductive case, $\alpha = a{:}as$, we must consider two situations.

Case $dom(a) \in sources(a{:}as,u,s)$: From Property 1(d) and the case assumption, we derive $dom(a) \in sources(a{:}as,u,t)$. The definition of *dipurge* gives $dipurge(a{:}as,u,t) = a : dipurge(as,u,step(t,a))$. Combining this result with the definition of *run* reduces the proof obligation to

$$\begin{aligned}
& s \overset{sources(a:as,u,s)}{\approx} t \;\Rightarrow \\
& \quad run(step(s,a),as) \overset{u}{\sim} \\
& \quad run(step(t,a), dipurge(as,u,step(t,a))).
\end{aligned}$$

From the hypothesis and step consistency over $\approx$ (Lemma 1), we obtain

$$step(s,a) \overset{sources(as,u,step(s,a))}{\approx} step(t,a),$$

and the conclusion follows from the inductive hypothesis.

Case $dom(a) \notin sources(a{:}as,u,s)$: Similarly to the previous case, we apply the definitions of *dipurge* and *run* to the conclusion. We have $dipurge(a{:}as,u,t) = dipurge(as,u,step(t,a))$ given the case assumption, so the proof obligation becomes

$$\begin{aligned}
& s \overset{sources(a:as,u,s)}{\approx} t \;\Rightarrow \\
& \quad run(step(s,a),as) \overset{u}{\sim} \\
& \quad run(t, dipurge(as,u,step(t,a))).
\end{aligned}$$

There is a discrepancy in the term $run(t, dipurge(as,u,step(t,a)))$: the state argument of *dipurge* does not match the state argument of *run*. The lemma requires these states to be equal, which we establish using the case assumption, the lemma hypothesis, and *dipurge* state equivalence (Property 3). After this step, we are left with

$$run(step(s,a),as) \overset{u}{\sim} run(t, dipurge(as,u,t)).$$

We derive $s \overset{sources(as,u,step(s,a))}{\approx} step(s,a)$ using local respect over $\approx$ (Lemma 2) and $s \overset{sources(as,u,step(s,a))}{\approx} t$ by $sources.\subseteq$. Applying the transitivity of $\approx$ to these properties results in

$$step(s,a) \overset{sources(as,u,step(s,a))}{\approx} t,$$

allowing us to apply the inductive hypothesis to reach the conclusion.

This completes the proof. □

Lemma 4. If M is a view-partitioned, output consistent system such that

$$do(\alpha) \overset{u}{\sim} do(dipurge(as,u,s_0)),$$

then

$$test(as,a) = test(dipurge(as,dom(a),s_0),a)$$

Starting with the assumption and substituting $dom(a)$ for u provides

$$do(as) \overset{dom(a)}{\sim} do(dipurge(as,dom(a),s_0)).$$

Then, by output consistency,

$$\begin{aligned}
& output(do(as),a) \overset{dom(a)}{\sim} \\
& \quad output(do(dipurge(as,dom(a),s_0)),a),
\end{aligned}$$

which is

$$test(as,a) = test(dipurge(as,dom(a),s_0),a)$$

by definition. □

Theorem 2. If M is a view-partitioned system that

1) is output consistent,
2) is weakly step consistent,
3) satisfies local respect, and
4) satisfies policy respect,

then

$$test(as,a) = test(dipurge(as,dom(a),s_0),a)$$

Specializing Lemma 3 by substituting $s = t = s_0$ yields

$$\begin{aligned}
& s_0 \overset{sources(as,u,s_0)}{\approx} s_0 \;\Rightarrow \\
& \quad run(s_0,as) \overset{u}{\sim} run(s_0, dipurge(as,u,s_0)),
\end{aligned}$$

which reduces to

$$run(s_0,as) \overset{u}{\sim} run(s_0, dipurge(as,u,s_0))$$

by reflexivity of $\approx$. We apply the definition of *do* to obtain

$$do(as) \overset{u}{\sim} do(dipurge(as,u,s_0)),$$

and the conclusion follows from Lemma 4. □